Norwegian University of
Science and Technology

Multi-core programming with OpenCL: performance and portability

OpenCL in a memory bound scenario

Olav Aanes Fagerlund

Master of Science in Computer Science

Submission date: June 2010

Supervisor: Lasse Natvig, IDI

Co-supervisor: Hiroshi Okuda, Okuda Laboratory, The University of Tokyo, Japan.

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

With the advent of multi-core processors desktop computers have become multiprocessors requiring parallel programming to be utilized efficiently. Efficient and portable parallel programming of future multi-core processors and GPUs is one of today s most important challenges within computer science. Okuda Laboratory at The University of Tokyo in Japan focuses on solving engineering challenges with parallel machines. A multi-core FEM solver package is under development within this laboratory that utilizes both standard CPUs and GPUs.

This student project, given by Department of Computer and Information Science (IDI) at NTNU in cooperation with Okuda Laboratory at The University of Tokyo, seeks to explore the promising path towards more platform independent parallel programming given by the OpenCL library, runtime system and language.

The main goals of the project are;

OpenCL as a multi-core programming tool and its inherent performance and portability properties is of interest. On background of code developed within this project, we wish to explore this area.

Some relevant and agreed upon sub-parts of the FEM solver package will be written/ported to OpenCL. This code will be used as basis for the performance and portability experiments needed for the project.

Experiments with one or several tools used for performance measuring and profiling of OpenCL code. Nvidias performance measuring and profiling tools should be included here.

If time permits;

For the study of performance tools as mentioned above; include one or more from another vendor; Intel, AMD/ATI or Nvidia.

Based on the experiments, suggest ways to tune portions of the OpenCL code for efficient multi-core/GPU execution.

Study how performance is affected when porting programs between different platforms.

Provide estimates for some OpenCL programs as a function of the number of cores/compute units used.

Compare the performance of benchmark programs implemented in OpenCL with comparable implementations in other languages. Such benchmark programs can be suggested both from the Okuda laboratory and Natvigs research group at NTNU.

Study the interplay of current OpenCL implementations and the operating systems they run on with respect to performance.

A focus on debugging tools for OpenCL is of interest.

Okuda Laboratory is expected to facilitate the project with a relevant focus area that will be agreed upon (via a research plan), as well as infrastructure such as a multi-core/GPU system for the experiments to the extent it is needed. IDI at NTNU provides an 8-way Intel Xeon processor system with Nvidia and ATI OpenCL compatible GPUs.

"A developer interested in writing portable code may find that it is necessary to test his design on a diversity of hardware designs to make sure that key algorithms are structured in a way that works well on a diversity of hardware. We suggest favoring more work-items over fewer. It is anticipated that over the coming months and years experience will produce a set of best practices that will help foster a uniformly favorable experience on a diversity of computing devices."

— OpenCL 1.0 specification [12], Appendix B – Portability

Abstract

During this master's thesis work, the CUKr library has been given additional support for running the Cg Krylov solver on all hardware supported by OpenCL implementations. This includes selected BLAS 1 and BLAS 2 kernels. Changes were made to the CUKr source-code infrastructure to accommodate the use of OpenCL. This implementation has been measured up against the C for CUDA based implementation already a part of the library. The results of the work strongly indicate that there are OpenCL performance issues in Nvidias Computing SDK 3.0, relative to the same SDKs C for CUDA performance. This is to an expected degree, as OpenCL implementations are still not as mature as some older technologies, for instance C for CUDA.

A BLAS 1 kernel considerably more suitable for the CPU memory access pattern was written, and compared against the Intel MKL Library. Simple changes to the memory access pattern demonstrated far superior performance. It was observed that a GPU friendly kernel had problems utilizing the cache when running on the CPU due to the unsuitable memory access pattern. The issues of producing portable code that performs adequately in a High Performance Computing scenario, for memory bound problems, has been explored. The author believes, as a result, that the place for OpenCL within High Performance Computing is as a powerful system for heterogeneous computing. Maintainability and ensuring performance in the kernels, in the mentioned scenario, does not call for a least common denominator, so to speak, with mediocre performance on all hardware. A kernel written to run "unbiased" on both GPU and CPU devices will most certainly have a hard time competing with other libraries targeting a certain device. OpenCL gives good flexibility and portability. However, when considering the performance aspects, and especially for memory bound problems, special care is crucial — as it always has been. Each device has its own ideal memory access pattern that cannot be ignored. Writing efficient BLAS kernels for a certain device in of itself can be a challenge. Making this perform well on a completely different architecture without degrading the performance on the first architecture considerably complicates the task. And it can be argued if this should be done, due to the unnecessary complexity of the code it introduces, from the standpoint of maintainability.

The GPU kernels are expected to run with reasonable efficiency on other recent OpenCL-ready GPUs too, such as those from AMD/ATI. The work has resulted in a more future-ready library, and can enable other interesting topics and focus areas that build upon this added foundation.

Contents

1 Introduction **1**
- 1.1 Thesis problem description 1
- 1.2 Research plan 3
- 1.3 Interpretation of the thesis problem description 3
- 1.4 Thesis structure and overview 4

2 Background for software technologies and tools **5**
- 2.1 Multi-core programming state-of-the-art 5
  - 2.1.1 OpenMP 7
  - 2.1.2 Intel Threading Building Blocks (TBB) 8
  - 2.1.3 Apple Grand Central Dispatch (GCD) 9
- 2.2 OpenCL 9
  - 2.2.1 Inspiration from the computer graphics scene 10
  - 2.2.2 Execution 11
  - 2.2.3 The Low Level Virtual Machine (LLVM) Compiler Infrastructure 11
  - 2.2.4 GPU execution 12
  - 2.2.5 CPU execution 13
  - 2.2.6 The memory hierarchy 14
  - 2.2.7 OpenCL CPU support status 14
- 2.3 Cmake build system for platform independent builds 15

3 Background for the implementation **17**
- 3.1 Solvers 17
- 3.2 Krylov solvers 18
- 3.3 Important compute kernels for the Cg Krylov solver 20
  - 3.3.1 AXPY 20
  - 3.3.2 AYPX 20
  - 3.3.3 DOT 20
  - 3.3.4 SCAL 20
  - 3.3.5 SpMV 21
- 3.4 Sparse Matrix Vector Multiplication (SpMV) on GPUs 21
- 3.5 Data formats of relevance for use with SpMV 22

3.5.1 Compressed sparse vector format (CSV) 22
3.5.2 Compressed sparse row storage format (CSR) 22
3.5.3 Block compressed sparse row storage format (BCSR) 23
3.5.4 ELLPACK 24
3.5.5 Block ELLPACK storage format (BELL) 24
3.5.6 Hybrid (HYB) 25
3.6 The CUDA Krylov (CUKr) software version 1.0 26
3.6.1 The structure of CUKr 28
3.6.2 The BLAS level 28
3.6.3 The data structure level 28

4 Background for relevant hardware 33
4.1 Nvidia OpenCL capable graphics hardware 33
4.1.1 Nvidia Tesla architecture 33
4.1.2 Nvidia Fermi architecture 34
4.1.3 Ideal global memory access pattern 36
4.2 AMD/ATI OpenCL capable graphics hardware 37
4.2.1 Architectural overview 37
4.2.2 Ideal global memory access pattern 39
4.3 A more CPU-ideal global memory access pattern 39
4.3.1 Memory access on the CPU 40

5 Implementing OpenCL support in CUKr 45
5.1 At the build level 45
5.2 Additions to the CUKr infrastructure and data-structure level 46
5.3 Additions to the BLAS level — the set-up of the OpenCL kernels 47

6 Kernel implementations 51
6.1 CUKr OpenCL kernels ideal for the GPU 51
6.1.1 Common structure 52
6.2 Differences between the OpenCL and CUDA kernels 58
6.2.1 BLAS 1 functions 58
6.2.2 SpMV functions 58
6.3 CUKr OpenCL kernels ideal for the CPU 59

7 Results 61
7.1 Performance evaluation 61
7.2 Performance measuring 63
7.3 Results BLAS 1 GPU-friendly kernels — individual benchmarks 64
7.3.1 Nvidia GTX 280 under Linux, Nvidia OpenCL 65
7.4 Results AXPY CPU-friendly kernel on CPU 70

7.5 Results Cg Krylov solver and its GPU-friendly kernels — real-world problems . . . 73
7.5.1 Nvidia GTX 280 under Linux, Nvidia OpenCL 3.0 SDK 73

8 Conclusions 79

9 Further work 83

A Hardware specifications 87

B OpenCL devices under different implementations 93
B.1 Apple Mac Pro, OS X 10.6.4 . . . 93
B.2 Apple Mac Pro, OS X 10.6.3 . . . 94
B.3 Apple Macbook Pro, OS X 10.6.4 . . . 96
B.4 Apple Macbook Pro, OS X 10.6.3 . . . 97
B.5 Nvidia CUDA SDK 3.0 Linux . . . 98
B.6 ATI Stream SDK 2.1 Linux . . . 100
B.7 ATI Stream SDK 2.01 Linux . . . 100

C Matrix properties 103

D Benchmark graphs 105

E Code listings 117
E.1 AXPY CPU Single . . . 118
E.2 AXPY GPU Single . . . 119
E.3 AXPY GPU Double . . . 120
E.4 AYPX GPU Single . . . 121
E.5 AYPX GPU Double . . . 122
E.6 DOT GPU Single . . . 123
E.7 DOT GPU Double . . . 124
E.8 SCAL GPU Single . . . 125
E.9 SCAL GPU Double . . . 126
E.10 SPMV CSR GPU Single . . . 126
E.11 SPMV CSR_B0 GPU Single . . . 128
E.12 SPMV CSR_A1 GPU Single . . . 129
E.13 SPMV CSR_A1_B0 GPU Single . . . 130
E.14 SPMV CSR GPU Double . . . 132
E.15 SPMV CSR_B0 GPU Double . . . 133
E.16 SPMV CSR4 GPU Single . . . 135
E.17 SPMV CSR4_B0 GPU Single . . . 136
E.18 SPMV CSR4_A1 GPU Single . . . 137
E.19 SPMV CSR4_A1_B0 GPU Single . . . 138
E.20 SPMV CSR4 GPU Double . . . 140
E.21 SPMV CSR4_B0 GPU Double . . . 141

E.22 SPMV ELL GPU Single 142
E.23 SPMV ELL GPU Double 143
E.24 Kernels GPU single-double (quasi-double) 144
E.25 Kernels GPU single set-up 164
E.26 Kernels GPU single set-up, header 182
E.27 Kernels GPU single-double (quasi-double) set-up 183
E.28 Kernels GPU single-double (quasi-double) set-up, header . . 204
E.29 Kernels GPU double set-up 205
E.30 Kernels GPU double set-up, header 218
E.31 OpenCL Initialize 220
E.32 OpenCL Initialize, header 233
E.33 OpenCL devices probing 235

List of Figures

2.1 An application under execution builds and initiates an OpenCL kernel, which is thereby executed on a selection of devices. . 12
2.2 The OpenCL Memory Hierarchy adopted from [12]. A compute device has N compute units, and each compute unit handles M work-items (or threads). 15

3.1 Compressed sparse vector layout. 22
3.2 Compressed sparse row layout. 23
3.3 BCSR layout. 23
3.4 ELLPACK/ITPACK layout. 24
3.5 Blocked ELLPACK steps. Figure adopted from [4]. 25
3.6 The HYB format. Figure adopted from [7]. 26
3.7 The layers of CUKr, adopted from [6]. 29
3.8 The block-layout of CUKr. Red boxes shows existing and new areas where work will take place during the implementation phase. The block-layout is adopted from a CUKr labmeeting note by Serban Georgescu, with additions from the author to illustrate the new state. 30

4.1 The Nvidia Geforce GTX 280 architecture overview. Illustration style is inspired by the Geforce GT 8800 figure in [15]. . 35
4.2 The Nvidia Geforce GTX 280 TPC. Illustration style is inspired by the Geforce GT 8800 TPC illustration in [15]. 36
4.3 The R700 architecture figure adopted from [16]. OpenCL Compute Units marked, in addition. 42
4.4 Illustration showing the SIMD element (Compute Unit) and the Stream Core. Partly adopted from [17]. 43
4.5 GPU coalesced read. The red circle indicates the memory requests that gets coalesced into one transfere. 43
4.6 CPU read with GPU kernel. The chaotic memory access pattern arising when using a GPU kernel on the CPU is shown. CPU memory-bandwidth badly utilized. 43
4.7 CPU ideal read with CPU kernel. Each core reads a large sequence of data in memory. 44

7.1 AYPX, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. Partitioning sizes are also adjusted to suit. 66
7.2 AYPX, OpenCL kernels uses local memory, as the CUDA kernel also does. Similar partitioning sizes as to the CUDA kernels are used. 67
7.3 AYPX with large vector sizes — up to 21 million elements, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. Partitioning sizes are also adjusted to suit. 68
7.4 AYPX with large vector sizes — up to 21 million elements, OpenCL kernels uses local memory, as the CUDA kernel also does. Similar partitioning sizes as to the CUDA kernels are used. 69
7.5 DOT; OpenCL vs. CUDA implementation. 70
7.6 DOT with large vector sizes — up to 21 million elements; OpenCL vs. CUDA implementation. 71
7.7 SCAL with large vector sizes — up to 21 million elements, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. 72
7.8 AXPY CPU-friendly kernel on Intel Core 2 Quad processor. . 73
7.9 Cg HYB single precision benchmark result. 74
7.10 Cg HYB qdouble precision benchmark result. 75
7.11 Cg HYB double precision benchmark result. 75
7.12 Cg CSR4 single precision benchmark result. 76
7.13 Cg CSR4 qdouble precision benchmark result. 76
7.14 Cg CSR4 double precision benchmark result. 77
7.15 Cg CSR single precision benchmark result. 77
7.16 Cg CSR qdouble precision benchmark result. 78
7.17 Cg CSR double precision benchmark result. 78

D.1 AXPY, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. 106
D.2 AXPY, OpenCL kernels uses local memory, as the CUDA kernel also does. 107
D.3 AXPY with large vector sizes — up to 21 million elements, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. 108
D.4 AXPY with large vector sizes — up to 21 million elements, OpenCL kernels uses local memory, as the CUDA kernel also does. 109
D.5 AYPX, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. Partitioning sizes are also adjusted to suit. Bandwidth utilization is illustrated. 110

D.6 AYPX, OpenCL kernels uses local memory, as the CUDA kernel also does. Similar partitioning sizes as to the CUDA kernels are used. Bandwidth utilization is illustrated. 111
D.7 AYPX with large vector sizes — up to 21 million elements, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. Partitioning sizes are also adjusted to suit. Bandwidth utilization is illustrated. 112
D.8 AYPX with large vector sizes — up to 21 million elements, OpenCL kernels uses local memory, as the CUDA kernel also does. Similar partitioning sizes as to the CUDA kernels are used. Bandwidth utilization is illustrated. 113
D.9 DOT; OpenCL vs. CUDA implementation. Bandwidth utilization is illustrated. 114
D.10 DOT with large vector sizes — up to 21 million elements; OpenCL vs. CUDA implementation. Bandwidth utilization is illustrated. 115
D.11 SCAL with large vector sizes — up to 21 million elements, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. Bandwidth utilization is illustrated. 116

List of Tables

3.1 Solver classification, adopted from [7], page 4. 19
3.2 CUKr BLAS object. 31
3.3 CUKR_VECTOR_SP data structure. The data members are pointers to arrays of scalars (float, double or int). This is also compatible with CUDA, as the kernels directly accepts pointers to the arrays where the data is stored on the device. 31
3.4 CUKR_MATRIX_SP data structure 32

5.1 CUKR_VECTOR_SP data structure with new additions for OpenCL support; cl_mem object pointers for referencing vectors for use with OpenCL added. Note that OpenCL cannot use ordinary pointers that references arrays on the device, therefore cl_mem objects are used to store the data. 48

7.1 Maximum achievable theoretical peak performance for the memory bound BLAS 1 kernels (single and double precision given here, respectively), in GigaFlop/s. 64

A.1 Intel CPU characteristics 88
A.2 ATI Radeon HD 4870 characteristics 89
A.3 ATI Radeon HD 5870 characteristics 90
A.4 Nvidia GTX 280 characteristics 91
A.5 Nvidia GTX 480 characteristics 92

C.1 Matrix properties table. The divisions shows the 3 groups used. From top to bottom; small – medium – large, respectively. The last four matrices are from subsequent structural problems. CFD is short for Computational Fluid Dynamics. All matrices are **2D/3D**. 104

Acknowledgements

There are quite a few people I have gratitude towards directly related to this thesis and the fact that I could work on it in Japan. For making it easier for me coming to Japan and answering a lot of questions for me, I would like to thank Rune Sætre. His help has been remarkable. He put me in touch with Serban Georgescu, at that time still at the Okuda Laboratory, who was very helpful and discussed with me possible areas I could come and work on. I would also like to thank Serban Georgescu for all the questions he has answered during my work. That was truly helpful. I would deeply like to thank Professor Hiroshi Okuda for making this stay possible by accepting me as a Research Student at his Laboratory, and making it considerably easier for me to come. I would also like to thank him for his feedback during our meetings. I owe many thanks to Professor Lasse Natvig for open-mindedly encouraging me when I suggested such a stay, and being a good support in form of video meetings and feedback while at the Okuda Laboratory here in Japan. I would like to thank the members of the Okuda Laboratory for making my stay pleasant, and for receiving me in the way they did. Especially I would like to thank Yohei Sato, Tatsuru Watanabe, Masae Hayashi, Masaaki Suzuki, Yasunori Yusa and Tairo Kikuchi. Tatsuru Watanabe was of big help for a lot of technical issues, thanks for that.

Last but not least, I would like to thank my parents Brita Aanes and Tore Hind Fagerlund, and my sister Silje Aanes Fagerlund. For always being there.

Chapter 1

Introduction

This thesis originated out of two desired objectives; **(1):** the wish to take a look at OpenCL as a high performance parallel programming tool from a portability aspect, and **(2):** in the process contribute to a piece of software called the **CUKr** (**CU**DA **Kr**ylov), developed by Serban Georgescu [7], at the Okuda Laboratory at The University of Tokyo, Japan — making the software able to utilize a broad range of parallel hardware through the use of the OpenCL runtime and library, and still be portable.

1.1 Thesis problem description

The decided thesis problem description, as of November the 5th 2009, follows:

> With the advent of multi-core processors desktop computers have become multiprocessors requiring parallel programming to be utilized efficiently. Efficient and portable parallel programming of future multi-core processors and GPUs is one of today's most important challenges within computer science. Okuda Laboratory at The University of Tokyo in Japan focuses on solving engineering challenges with parallel machines. A multi-core FEM solver package is under development within this laboratory that utilizes both standard CPUs and GPUs. This student project, given by Department of Computer and Information Science (IDI) at NTNU in cooperation with Okuda Laboratory at The University of Tokyo, seeks to explore the promising path towards more platform independent parallel programming given by the OpenCL library, runtime system and language. The main goals of the project are;
>
> - OpenCL as a multi-core programming tool and its inherent performance and portability properties is of interest. On background

of code developed within this project, we wish to explore this area.

- Some relevant and agreed upon sub-parts of the FEM solver package will be written/ported to OpenCL. This code will be used as basis for the performance and portability experiments needed for the project.
- Experiments with one or several tools used for performance measuring and profiling of OpenCL code. Nvidias performance measuring and profiling tools should be included here.
- If time permits;
  - For the study of performance tools as mentioned above; include one or more from another vendor; Intel, AMD/ATI or Nvidia.
  - Based on the experiments, suggest ways to tune portions of the OpenCL code for efficient multi-core/GPU execution.
  - Study how performance is affected when porting programs between different platforms.
  - Provide estimates for some OpenCL programs as a function of the number of cores/compute units used.
  - Compare the performance of benchmark programs implemented in OpenCL with comparable implementations in other languages. Such benchmark programs can be suggested both from the Okuda laboratory and Natvigs research group at NTNU.
  - Study the interplay of current OpenCL implementations and the operating systems they run on with respect to performance.
  - A focus on debugging tools for OpenCL is of interest.

Okuda Laboratory is expected to facilitate the project with a relevant focus area that will be agreed upon (via a research plan), as well as infrastructure such as a multi-core/GPU system for the experiments to the extent it is needed. IDI at NTNU provides an 8-way Intel Xeon processor system with Nvidia and ATI OpenCL compatible GPUs.

1.2 Research plan

The research plan was formed in collaboration with Okuda Laboratory, and describes in more detail the actual implementation work to be performed at the laboratory, as part of the thesis.

> CUDA Krylov (CUKr) is a package created at the Okuda Laboratory as part of Serban Georgescu's PhD thesis [7]. This is defined as an Accelerated Krylov Solver Interface implementation (AKSI) in the same thesis. CUKr is, by construction, able to use multiple BLAS libraries to accommodate both GPUs and CPUs. When utilizing GPUs, the CUDA programming language, runtime and library is used in combination with Nvidia hardware.
>
> This research aims to utilize the new OpenCL (language, runtime and library) technology and its inherit strength with respect to device independence to target a number of different parallel architectures. This will result in software with CUKr's capabilities that in addition is capable of utilizing all hardware supported by OpenCL implementations with small or no changes to the source code. Rather than using multiple BLAS libraries, the software should now have a common abstraction (codebase/source code) for all architectures. A goal is to investigate if the common abstraction can reach competitive performance on both CPU and GPU devices, compared to other specific implementations targeting a certain device (is this possible with this kind of memory bound problems?). This project includes porting/rewriting BLAS1 functions and SPMV, which should allow for different data formats, at least CSR, CSR4, ELL and HYB. 3x3BCSR and 3x3BELL if time allows.
>
> The OpenCL based software will be constructed for platform portability (support different OS'). An aim, if time allows, is to make it utilize several compute devices, and harvest the resources of a heterogeneous system; specifically, benefit from different types of compute devices. It should be benchmarked against the CUDA based version. What performance can OpenCL give, and still provide portable parallel code?

1.3 Interpretation of the thesis problem description

When mentioning *"OpenCL as a multi-core programming tool and its inherent performance"* it implies that *OpenCL* means its implementations available today implementing the 1.0 version of the specification. As OpenCL is a new technology it is expected that the implementations available today will improve over time, as with all new technologies of a certain complex-

ity. Such improvements will have an effect on the performance seen when executing the kernels written in the language previously.

GPUs available in the Apple Mac Pro at NTNU is one ATI 4870, as the model can not house two cards due to power needs (actually lack of enough power connectors needed by the cards at the PSU). It has later been found that the ATI 4870 is not a good OpenCL performer, as the card was designed before the specification work took place and not with OpenCL directly in mind. However, it is said that careful programming can get the card perform, something that may make the code less suitable for other architectures from a performance viewpoint.

1.4 Thesis structure and overview

This *first* chapter contains the introduction. Following, chapter *two* contains the background of software technologies and tools. The *third* chapter also contains background material; everything that is of relevance for the implementation work. Chapter *four* is the last background-chapter, covering the relevant hardware.

About the implementation itself is covered in chapter *five*, continuing with the kernel implementations in chapter *six*. Chapter *seven* covers the results, and chapter *eight* the conclusions of the work. Finally, chapter *nine* looks at further work that would be of interest after the completion of this thesis work. Appendixes contains hardware specifications, OpenCL device-information under different implementations, matrix properties, benchmark graphs and finally code listings.

Chapter 2

Background for software technologies and tools

This chapter will visit the current state of parallel programming on commodity hardware to give an overview. The highlight is on new and important trends contributing to easier and scalable parallel programming suitable for high performance computing applications both in science and mainstream consumer applications - for instance games. OpenCL will, of course, be covered in more depth as it is of focus in this thesis.

2.1 Multi-core programming state-of-the-art

Shared memory multi-core programming has in the last decade moved towards a trend where the programmer is relived from the details of having to administrate individual threads. Letting the programmer create and administrate threads in-code is an error prone process, and at the same time makes it more difficult to scale the application as processors with increasingly more cores are introduced to the market. Libraries and runtimes that do this heavy lifting are the way of the future, and a high-level coverage of some of the most important in this category is given here. These technologies handle the low-level threading, so the programmer does not have to. The trend is that the programmer can rather think in tasks that can be parallelized and state this by proper syntax, and leave the low-level job of administrating the actual threads needed for the parallelization to the library and/or runtime. In this approach, of course, the programmer still has to know what should be parallelized. Administrating threads "by hand" is not getting easier with increasing number of cores. It is clear that these newer approaches do not attempt to solve the still standing problem of having the compiler automatically see all the parallelism itself, without requiring the programmer to express parallelism. But these technologies do make life considerably easier for the programmer, and will make parallel

programming more accessible for the vast majority of programmers as they have to adjust to the new reality of increasingly more parallel machines. It is of benefit not only for the lifecycle of the application, by making it more scalable and future proof, but also for the programmer in regard of ease of programming. One of the latest attempts in this regard is Apple's GCD (Grand Central Dispatch) introduced in OS X 10.6 Snow Leopard in August 2009. Intel's Threading Building Blocks and the latest OpenMP efforts are other good examples in this category.

The above-mentioned trend is valid for parallel programming of the CPU. These technologies are used in ordinary programs of the kind that previously required threads by either utilizing system specific threading mechanisms or pthreads and alike. However, programming a parallel chip that is not a CPU (rather any kind of accelerator or a special co-processor), like a modern GPU (Graphics Processing Unit), DSP (Digital Signal Processor) or FPGA (Field Programmable Gate Array), requires other approaches and is usually at a lower level and thus more details to take care of is required of the programmer. Examples here includes Nvidia's CUDA (Compute Unified Device Architecture) and OpenCL (Open Compute Library). These technologies are developed for making programming of such mentioned massively parallel modern chip designs easier and much more accessible than previous. Traditional threading on the CPU is thus very different, it does not deliver the same massively parallel performance that a modern GPU can. OpenCL is unique in the sense that it also can target the CPU cores in a system for its computations as well. The CPU is ideal for task-parallel kernels, while the GPU is ideal for the execution of data-parallel ones.

A third and older (but still necessary and useful) way of parallel programming is with some sort of message passing library. This is useful when different compute nodes or workstations needs to cooperate to solve a problem. Modern supercomputers consists of computer nodes connected together in a high-speed network, to minimize communication costs. It is traditionally on such computers message passing has been a common choice. A good example here is the industry embraced MPI (Message Passing Interface) standard. A quite popular implementation in widespread use is OpenMPI. Such technologies are useful for spreading out work to the nodes, who themselves of course can be highly parallel heterogeneous systems. Each machine solves their subpart, and may be utilizing one of the other two above-mentioned paradigms - some sort of a threading library or OpenCL / CUDA. When the assigned task is done the node returns the result to a root node. Modern MPI implementations also work solely on shared memory machines, in which case each CPU core in this one machine is a "node" (and the communication done, in this case, does not enter a network at all). A good example of a project utilizing OpenMPI, OpenGL and OpenCL is the "Hybrid Parallel Gas Dynamics Code" ("HYP-

GAD") project [1]. This is the implementation of a solver for compressible gas dynamics.

To sum it up, the three popular parallel programming categories of importance today:

- Technologies to program and utilize massively parallel chips. Examples include Nvidias CUDA and the widely industry-embraced OpenCL standard.
- A library/technology relieving the programmer of tedious and error prone thread management, making parallel programming easier. Examples include Apple's GCD, Intel's TBB and OpenMP 3.0.
- Message passing libraries for distributing work to networked nodes, such as the MPI standard and its many implementations that exist. As pure shared memory parallel programming is of focus in this thesis, this category will not be covered.

A short overview of OpenMP, Intel Threading Building Blocks and Apple Grand Central Dispatch follows. This should explain at a high level what they offer and their differences.

2.1.1 OpenMP

OpenMP is a standard for multi-platform shared-memory parallel programming, supported by a wide range of platforms. It is used on shared memory systems of different scales, also single socket multicore systems. The specification of version 3.0 can be found at the URL given in [3]. As explained in the specification, OpenMP consists of compiler directives (pragmas), library routines, and environment variables. These are used in combination to specify shared-memory parallelism. The compiler directives adds single program multiple data (SPMD), work-sharing, tasking and synchronization constructs. In relation to the memory model used by OpenMP they give support for sharing (among threads) and privatizing (private for a thread) data. Library routines and environment variables gives the programmer the functionality to manage the runtime environment. The common scenario when programming in OpenMP is that a compute intensive loop is parallelized by the use of pragmas. When this code runs the main thread is forked into a number of threads (number of threads can be decided at runtime), and different portions of the loop is mapped to different cores running each of their own thread. When the compute intensive

[1] Please see the project page at http://hypgad.sourceforge.net. At Supercomputing 2009 this project was demonstrated with computation tasks being distributed to nodes consisting of different hardware (Intel Nehalem, IBM CELL, AMD Opteron and Nvidia GPU node). At each node the processing was done with the exact same OpenCL kernel, illustrating the portable advantage and flexibility OpenCL can give.

parallel region is complete, the threads join and the program continues as a ordinary sequential one. With OpenMP the forked threads can themselves again be forked, thus support more than one level of parallelism — also called nested parallelism. Nested parallelism was introduced with the NESL parallel programming language [2] in 1993.

With OpenMP 3.0 a higher level of abstraction was introduced, a *task*. *Tasks* allows a wider range of applications to be parallelized. The *task* is a piece of code that can be executed independently of other tasks. It is the programmers responsibility to make sure of this. The OpenMP runtime will schedule the defined tasks in parallel. OpenMP 3.0 support will be found in all major compilers in the near future, and is today fully supported by Sun Microsystems in their Sun Studio programming environment.

OpenMP gives the programmer the tools to write scalable and portable parallel programs. The programmer explicitly specifies the parallelism, through the compiler directives and library routines (thus telling actions to be taken by the compiler and runtime system so the program is executed correctly in parallel). OpenMP does not provide any automatic parallelization — it is all up to the programmer. Neither does OpenMP check for deadlocks, data conflicts, race conditions or data dependencies. As a conclusion; OpenMP can give portability and flexibility. It is widespread and popular, and will continue to evolve. The latest specification introduces modern features for easier parallel programming.

2.1.2 Intel Threading Building Blocks (TBB)

Intel TBB is a portable C++ library for multi-core programming. It can be used with Windows, Linux, OS X and other Unix systems. As it is only a library that is used with standard C++ code, no special compiler or language is required. It is a platform independent abstraction above the thread level that lets *tasks* to be defined and scheduled by a runtime that ensures good load balancing of these *tasks*. This makes TBB and OpenMP 3.0 somewhat similar in capability. Though, TBB's focus is purely on *tasks*, blocks of code that are run in parallel. TBB is, arguably, simpler to use for a programmer coming from the "sequential world" than OpenMP. Templates are used for common parallel iteration patterns, so programmers do not have to be highly skilled in synchronization, cache optimization or load balancing to get good performance. The programs written with TBB are scalable, and runs on systems with a single processor core or more. The tasks specified with TBB are mapped onto threads running on the cores. This is done efficiently by a runtime, either if you run on, say, two or twelve cores. This is much more efficient if you want a scalable parallel program, than using native threads or a threading library. The runtime has "work-stealing" capability, resulting in a more balanced execution of the task where less

busy cores can "steal" tasks originally give another core, that might be overworked at the moment. This can be the result of uneven scheduling seen from a system wide perspective. TBB thus compensates for this resulting in faster completion of the TBB based program. The MIT Cilk [1] system first introduced "work-stealing" capabilities. Another important property of TBB is the support of nested parallelism, also found in OpenMP. As a comparison with OpenMP; TBB is a infrastructure simpler for the average C++ programmer to utilize. It is used with success both within consumer applications and game engines relying on good and portable performance. As it is a C++ library, it is designed to be easily adopted by C++ programmers.

2.1.3 Apple Grand Central Dispatch (GCD)

GCD is similar to the two above-mentioned technologies in that the use of threads is abstracted away from the programmer. It introduces new language features and runtime libraries to provide support for parallel execution on multicore processors under OS X 10.6. The library providing the runtime services (`libdispatch`) is open source, and a port exists for FreeBSD. The GCD runtime works at the BSD-level of the OS X operating system, running above `pthreads`. GCD eases the programming of task-parallel applications. Under the hood there is a dynamic pool of threads executing the blocks of code handed over to GCD by the programmer. The blocks, or tasks, are queued by the programmer and routed. Here one can imagine parallel train-tracks, where train cars are routed to the appropriate tracks with the least amount of traffic (load). In a sense, this is analogous to packet routing on the internet — not *one* hardwired route is set up and always used. Where the packet goes is chosen dynamically (in GCD by the GCD runtime). Once a programmer has to deal with 4 threads or more things will easily get too complex. GCD tackles this problem. GCD significantly eases programming of multi-core processors, in a scalable fashion. It is easy to show that much less code is needed do multi-core programming with GCD than traditional threads. GCD is a software layer preparing for the future of multi-core processors, and among the new tools made available to tackle the multi-core era much more elegantly than what has been possible with traditional threads.

2.2 OpenCL

OpenCL is an open standard originally emerging from Apple Inc., who handed it over to the Khronos group as a suggestion to the industry summer of 2008. The OpenCL 1.0 specification was ratified in December 2008. The Khronos group is a non-profit organization with the goal to maintain a

variety of different open standards related to graphics, performance computing, and data exchange — with members from the industry contributing and agreeing upon the standards. All to benefit the industry, acknowledging the importance of such open standards. These standards then benefit the software developers, making the software they create a better and more future-proof investment. This is important, to secure freedom of the developer one should not have to be dependent on a certain company. OpenCL is a runtime-system, API and programming language enabling programmers to write data- and task-parallel programs that can target different kinds of processors; CPUs, GPUs and DSPs. The peculiarities of the underlying hardware is abstracted away from the programmer, who only needs relate to the API to get the work done. This is regardless of processor kind being targeted for execution. At the same time the programming is at a low enough level to give the programmer power and control, such as the possibility to optimize for speed depending on the processor kind being targeted (i.e. optimize memory transfers and problem partitioning). It is important to note that the OpenCL 1.0 specification [12] specifies the OpenCL API a programmer can use, and what OpenCL implementations must comply to in order to be OpenCL 1.0 compatible (a good example is IEEE754 based compliance). It does not specify how a working OpenCL implementation in itself is to be implemented, and how it should map kernels to different architectures. The bibliography in the OpenCL 1.0 draft specification [9], however, shows the sources the creators of the draft specification used as inspiration.

2.2.1 Inspiration from the computer graphics scene

With OpenCL the parallel programming environment has been inspired by the computer graphics scene[2]. OpenCL brings novel techniques that has been well developed in the computer graphics scene related to compilation and targeting for a specific device. Computer graphics hardware and the diversity in unique hardware implementations available has forced the use of fast Just-In-Time (JIT) compilers integrated into the graphics card drivers and runtime. The exact same philosophy is brought over to OpenCL implementations, to enable the massive support on different hardware. As expressed by Timothy G. Mattson, author of the book "Patterns for Parallel Programming" and employee at Intel working with parallel technology; the computer graphics-stack engineers had "a thing or two" to learn the

[2]In fact, the initial persons behind the draft specification had roots from computer graphics work (i.e. previously employed by ATI, or working with graphics driver or general graphics programming at Apple). Rumors has it IBM thought the OpenCL specification included to many ties to graphics (as in, amongst others, image objects as possible memory objects), and uttered opinions related to this during the standardization work process.

parallel software tool-chain developers. An OpenCL compute kernel is just pure source code before the program setting it up is executed. As analogy, this is exactly the same for a shader used with OpenGL. Both the OpenGL shader and the OpenCL kernel are compiled for the targeted architecture on the fly during program execution. This is done in this way because of the variety of hardware it should be able to run on. It is not known before program execution what kind of chip the kernel or shader will run on. Setting up a OpenGL shader the programmer has to go through certain steps, very similar to the approach taken when setting up a OpenCL kernel for execution; The shader must be loaded, compiled and linked, from the main program. Also, the vertex buffer objects that holds the shapes must be set up, and the variables to be passed into the shader. One can here switch the word "shader" with "kernel" to get something that almost completely describes the process of setting up a OpenCL kernel for execution. The only difference is that the memory object you operate on might not only be constrained to a vertex buffer object, as OpenCL can do much more than just processing graphics. OpenCL brings along advanced and smart use of a runtime and compiler, inspired by the way it has been done in the computer graphics stack for almost a decade or so, to the world of parallel computing.

2.2.2 Execution

A program utilizing OpenCL starts life as an ordinary program executing on the CPU, and includes OpenCL header files to gain access to the Platform and Runtime API. The Platform API is used to set up and prepare devices for execution by creating compute contexts, as explained in [12]. Kernel source programmed in the OpenCL programming language is built as executables for the target devices during main program execution (host program running on the CPU), and thereby executed on the selected devices. For this part the Runtime API calls are used, and the compilation of the kernel by an OpenCL runtime compiler. An overview of this sequence is shown in figure 2.1. In most implementations the OpenCL source code is first compiled into an intermediate representation which is device independent. This intermediate code is optimized as much as possible, before the final code for the selected device is generated by the device's code generator (as part of the device's OpenCL driver/runtime infrastructure).

2.2.3 The Low Level Virtual Machine (LLVM) Compiler Infrastructure

The way OpenCL is specified to work requires the use of a just-in-time (JIT) compiler that can target a given architecture. Most, if not all, OpenCL implementations released to this date makes us of a JIT compiler devel-

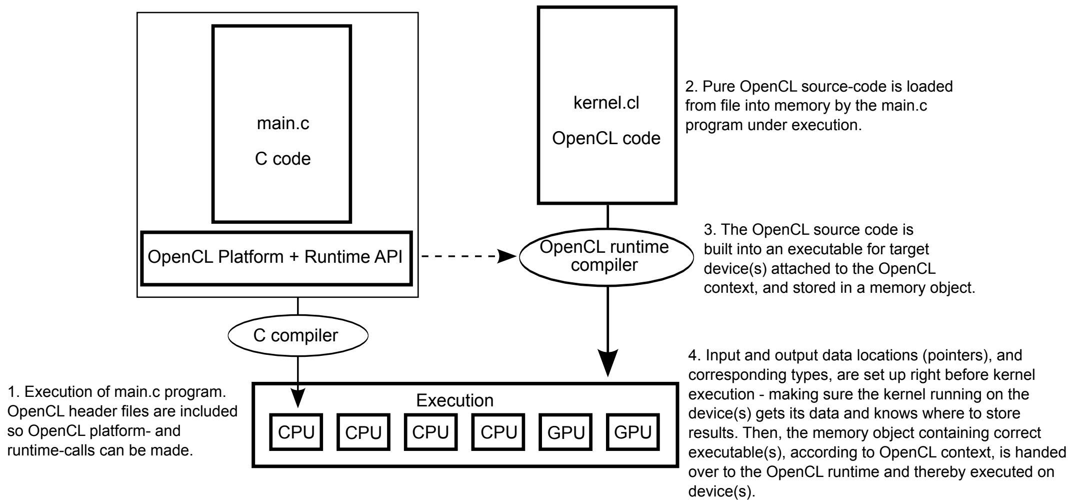

Figure 2.1: An application under execution builds and initiates an OpenCL kernel, which is thereby executed on a selection of devices.

oped with the LLVM open source project. LLVM is a compilation strategy, a virtual instruction set and a compiler infrastructure. It enables the construction of highly efficient JIT compilers, and also traditional static compilers. It is a modern and new compiler infrastructure. JIT compilers have become more and more demanded the last decade or two (both for general code targeting the CPU, and in the graphics pipeline for compilation of shaders that will run on a GPU). For an account of the ideas behind LLVM please see [14] and [13].

2.2.4 GPU execution

The JIT compiler targets the GPU when it is selected as a compute device with OpenCL. At kernel launch, the memory object containing the executable, the compiled kernel, is uploaded to the GPU itself. Data it works upon is by this time already in place in the device' global memory. Execution starts.

Due to the massively parallelism found in modern GPUs, data-parallel execution of kernels is ideal. GPUs are massive data-parallel handling-devices, well suited for performing the same tasks on large amounts of data in parallel. GPUs are not suitable of task-parallelism, as compute units must follow the same uniform operation.

Each compute unit of the GPU are assigned work-groups for execution. All the compute units process work-groups simultaneously until all the work-groups are processed. The exact same kernel is executed for each work-item, the data operated upon differ. The data-parallel execution per-

formance by far exceeds that of the current day CPU.

2.2.5 CPU execution

When the CPU is targeted the kernel is compiled for the CPU, where it is executed. The CPU is ideal as a main target for task-parallel execution under OpenCL. Single work-item performance is much higher on the CPU than the GPU due to higher clock-speeds and more powerful individual cores found in the CPU. The share number of concurrent threads or independent compute cores (compute-units consists of many of these) in the GPU makes it better for data-parallel execution, although each compute core is weaker. For CPU execution command queues can be used to build a dependency graph, containing information about the kernel dependencies. This enables advanced control, and the possibility of using one kernels output as input to another kernel. Under the task-parallel model different compute units of the CPU (CPU cores) can run different compute kernels simultaneously.

Also data-parallel execution can be done on the CPU. Each core will get work-groups assigned for processing, and executes each work-item in succession until the work-group is done. For every work-item being processed the instructions will then be the same (unless there is some branching taking place), but the data worked upon differs. At completion the next work-group in line is assigned to the core. All cores work in this manner until all work-groups of the problem domain are completed. If optimal; the compute kernel is running in loop on the cores while being feed with the right data for each work-item. This continues until all the data of the domain is processed (i.e. all work-groups are processed). Obviously, this takes longer (in most practical cases) than if the execution was done on a GPU which can execute hundreds of kernel-instances simultaneously(threads following the kernel instructions), and thus complete the work-groups much faster because of the share parallel throughput offered by the GPU.

For data-parallel execution it shows most optimal to let the number of *work-grups* equal the number of physical cores (or logical cores when this is available) available, and each have the size of one *work-item*. This is intuitive, as it is then known that the runtime will not make many instances of the data-parallel kernel run in succession on each core, giving some overhead. Rather each core runs its instance of the kernel until the complete task is done. As implementations improve over time this might be optimized by the runtime/compiler so it works in this manner even though each *work-group* contains many *work-items*. Task-parallel executions runs independent kernels, each set up by a domain of one *work-grup* containing one *work-item*. These are assigned to the CPU cores available.

2.2.6 The memory hierarchy

The memory hierarchy of OpenCL is seen in figure 2.2. The main entity seen here is the compute device, which represents a GPU, a CPU, a DSP (Digital Signal Processor), or any other kind of OpenCL capable chip. The compute device memory is typically this device's off-chip dedicated memory. In OpenCL this is mapped to the *Global memory* pool — a memory accessible to all compute units of the chip. The *Global memory* is the larges memory available, and also the slowest. Before a computation commence the necessary data is stored here, where it is reachable from the compute kernel. The compute units are cores or collections of computational elements inside the compute device chip itself. A modern graphics card has several of these compute units (the ATI 4870 has 10), each capable of running several hundreds of threads simultaneously. When mapped to the CPU the compute unit is a CPU core that may be able to execute two threads at once (via Intels HyperThreading or similar techniques). Such a core can thus only execute at most two threads concurrently. We say it has a max work-group size of 2 work-items. In comparison the ATI 4870 has a max work-group size of 1024 work-items. Each compute unit has access to a local memory, which is shared among all of its work-items (its work-group). This memory is an order of magnitude faster than the global memory, as it resides on-chip. Furthest down in the memory hierarchy is the private memory; private to each work-item. No other work-item can access this. It has the speed comparable to registers. Thus, the fastest memory work-items in the same work-group share is the local memory. There is no similar and equally fast way for work-groups to share data with eachother. While programming an OpenCL data-parallel kernel one keeps in mind that the kernel is ran as an instance by each work-item. The kernel defines how each work-item behaves as a piece of the whole, and how it interacts in relation to the memory hierarchy. So, the contribution of all the executed kernel instances gives the final result.

2.2.7 OpenCL CPU support status

ATIs (AMD) Stream SDK 2.0, as of November 5th 2009, supports targeting all x86 SSE (SIMD Streaming Extensions) 3.x CPUs. Wether from Intel or AMD. SIMD (Single Instruction Multiple Data) instructions are implemented in most modern CPUs, and allows for the same mathematical operations to be performed on a series of data in parallel. For example, multiplying four float values with another value in one instruction. The ATI Stream SDK also supports all ATI graphics cards from the Radeon HD 4350 and upwards. This OpenCL implementation is certified by The Khronos group at the time, November 5th 2009. It was the first OpenCL SDK available for multiple platforms that both supported targeting CPUs and GPUs,

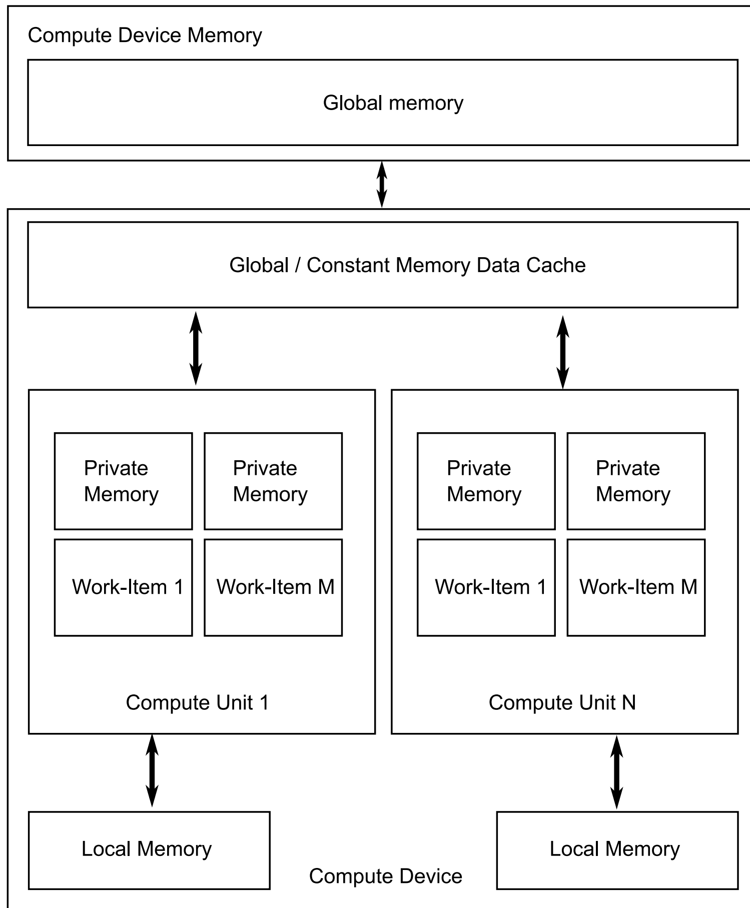

Figure 2.2: The OpenCL Memory Hierarchy adopted from [12]. A compute device has N compute units, and each compute unit handles M work-items (or threads).

enabling easy utilization of that interesting aspect of OpenCL. As Nvidia is not a producer of CPUs, their SDK does not, as of February 1st 2010, support targeting CPUs. The Apple OpenCL implementation runs on both Intel Nehalem CPUs and older Intel Core based CPUs (Core and Core 2), both CPUs found in all of their recent machines.

2.3 Cmake build system for platform independent builds

CUKr uses cmake to help build the CUKr library. Cmake is a system for generating build files for a specific platform, from cmake configuration files and cmake modules. As it works on many platforms, this significantly aids platform-independent software projects. With CUKr and the

new OpenCL support part of the library in mind, cmake will find both OpenCL libraries and header files, either building on a Linux machine or a Mac.

Chapter 3

Background for the implementation

This chapter will provide the background material for everything relevant for the implementation itself, explaining key concepts and ideas the implementation depends upon. The implementation is at the data-structure and BLAS level, the latter is where vital functions used by the CUKr Krylov solvers are implemented. Thus, none of the Krylov solvers themselves are extended or coded, but critical parts they depends upon. Therefore, we will start by a *high* level explanation of what the Krylov solvers are and why they are important in this domain of applications; FEM (Finite Element Method) and CFD (Computational Fluid Dynamics) kinds of problems. Krylov solvers are not the main focus of this thesis, but an area that can benefit of the implementations to be done at the BLAS level of the CUKr library. For a more detailed explanation about solvers and Krylov solvers, please see Chapter 1 and 2 of [7], which is one of the sources for this background material. As the matrix-vector and vector-vector operations further covered here (BLAS functions) are important for a wide range of engineering problems, providing efficient implementations utilizing OpenCL has a wide area of appliance, extending beyond Krylov solvers. And, as OpenCL is platform independent, open and supports parallel hardware, the implementations are highly future-proof.

3.1 Solvers

A solver is a machine implementation of a method used to arrive at a solution for a system of equations. There exists different kinds of solvers, each with their benefits and limitations. Depending on the domain, or kind of problem, the matrices can *dense*, or *sparse*. In *sparse* matrices most of the values are zeros (often more than 99% - 99.9%), and the rest are non-zeroes. The order of the matrices can be in the order of millions. This amounts to a

large amount of data. Data formats to store these in an efficient manner will be looked upon in a following section of this chapter (*Data formats of relevance for use with SpMV*). The use of these formats are vital to achieve performance when working with *sparse* matrices. The *sparse* matrices arise in areas such as computational fluid dynamics and structural analysis. Here, only the local interactions are of interest, which is the direct cause of the sparsity seen in the matrices. *Dense* matrices contains a small number of zero elements, and as no compression is a practical requirement they are easier to work with.

Solvers exists in two different kinds; *direct* and *iterative* solvers. The *direct* solvers produces exact solutions, but can be too time consuming when the order of the matrix is large enough — even impossible to use by the fastest computers available. They solve the system in an algebraic manner, by the use of substitution. Because of the restraints, *iterative* solvers are of interest in many cases, especially when an approximate solution is good enough (the approximation can be quite good so this is quite often true). For large and *sparse* matrices are *iterative* solvers much used. As they find an approximation through iterations, the answer keeps improving. It is an optimization approach. At one point the solution is judged good enough, the measure of error is acceptable (the residual).

An overview of the most popular solvers and their classifications can be seen in table 3.1.

3.2 Krylov solvers

Krylov subspace solvers are iterative solvers that are used with sparse matrices, as reflected in table 3.1. They are much used with large systems of linear equations. They work with matrices solely utilizing the matrix-vector product. So, the matrix is not affected, which other solvers can do by incurring something called *fill-in*; previous zero elements are turned into non-zeros, thus affecting the result. They are preferred because of the small memory foot-print, required computations, and the ability to handle unstructured problems. There exists several Krylov solvers, amongst others *Generalized Minimal Residual Method* (GEMRES)[19] and *Conjugate Gradients* (CG)[8]. These two are the most used ones. Both of these are part of the CUKr library. The time it takes to find an acceptable solution, convergence, is improved by the use of a *preconditioner*. This is often in the form of a *direct* solver. The performance of Krylov solvers is often limited by the memory bottleneck, as will be touched upon later. All kernels used by Krylov solvers are memory-bound. The most important ones includes SpMV, AXPY, AYPX and DOT, which we will visit shortly. When the CG Krylov solver is running, most of the time is spent in the SpMV kernel. This underlines the importance of a fast SpMV routine, as it greatly affects

	Dense matrices	**Sparse matrices**

Direct solvers	`Gaussian elimination`
	`Gauss-Jordan elimination`
	`LU decomposition`
		`Frontal`
		`Multifrontal`
		`Supernodal`

Iterative solvers	`Preconditioned iterative solvers`	`Jacobi`
		`Gauss-Seidel`
		`SOR, SSOR`
		`Krylov solvers (preconditioned)`
		`MG, AMG`

Table 3.1: Solver classification, adopted from [7], page 4.

the overall efficiency of the solver.

3.3 Important compute kernels for the Cg Krylov solver

Both AXPY and DOT are part of the BLAS level 1 functions, which consists of vector-vector operations, and no matrix-vector operations. The SpMV is part of BLAS level 2, which is containing matrix-vector operations.

3.3.1 AXPY

AXPY is defined by the function $\boldsymbol{y} \leftarrow \alpha * \boldsymbol{x} + \boldsymbol{y}$. The values of vector $\boldsymbol{x}$ are multiplied with the scalar α, and then the values of corresponding elements in vector $\boldsymbol{y}$ are added. The result is written to vector $\boldsymbol{y}$, replacing the old element values. The two vectors are of size n. The ratio between *computation* and *io* (double precision) for this operation is 2 *flop* / (3 x 8 Bytes).

3.3.2 AYPX

AYPX is similar to AXPY. Here vector $\boldsymbol{x}$ and $\boldsymbol{y}$ have taken the others place in the calculation. It is defined by the function $\boldsymbol{y} \leftarrow \alpha * \boldsymbol{y} + \boldsymbol{x}$. The values of vector $\boldsymbol{y}$ are multiplied with the scalar α, and then the values of corresponding elements in vector $\boldsymbol{x}$ are added. The result is written to vector $\boldsymbol{y}$, replacing the old element values. The two vectors are of size n. The ratio between *computation* and *io* (double precision) for this operation is 2 *flop* / (3 x 8 Bytes).

3.3.3 DOT

DOT is defined by $res = \sum \boldsymbol{x} * \boldsymbol{y}$. The corresponding elements in the two vectors of size n are multiplied with each other. Then all the resulting values are added together and stored in *res*. The result of the operation is thus one scalar value. The ratio between *computation* and *io* (double precision) for this operation is 2 *flop* / (2 x 8 Byte).

3.3.4 SCAL

SCAL is defined by $\boldsymbol{y} \leftarrow \alpha * \boldsymbol{y}$. Every element of the vector $\boldsymbol{y}$ of size n are multiplied with a scalar value α. Then all the resulting values are added together and stored in *res*. The result is written back to vector $\boldsymbol{y}$. The ratio between *computation* and *io* (double precision) for this operation is 1 *flop* / (2 x 8 Byte).

3.3.5 SpMV

SpMV is defined by $\boldsymbol{y} \leftarrow \alpha * \boldsymbol{A} * \boldsymbol{x} + \beta * \boldsymbol{y}$. Here $\boldsymbol{y}$ and $\boldsymbol{x}$ are vectors of size n. $\boldsymbol{A}$ is a $n \times n$ symmetric matrix, supplied in packed form as explained in the next two sub-chapters. α and β are scalars. As we will see later, performance on a given architecture is highly dependent on the format of $\boldsymbol{A}$ — the data-structure. The ratio between *computation* and *io* depends on the data-structure used and the parameters of the matrix, such as number of *non-zeroes* and dimensions of the matrix.

3.4 Sparse Matrix Vector Multiplication (SpMV) on GPUs

Untuned Sparse Matrix-Vector Multiplication (SpMV) implementations has historically not performed much more than 10% of system peak performance on cache-based superscalar microprocessors, as accounted for in Chapter 1 and 2 of [21]. It is a highly important computational kernel for use in many fields within engineering, and is defined as part of the BLAS level 2 specification. The limited performance is in great part due to the memory bottle-neck found in computers. It depends on streaming data to the kernel — data that is hardly reused afterwards. This becomes a limiting factor because the algorithm is highly data intensive. So, as means of improving the situation the matrices are stored in formats having less of a memory footprint. Formats that optimize performance and minimize memory usage [7]. The fact that sparse matrices contains mostly 0-elements is exploited; these formats only stores the non-zero elements and the indexing information needed for each of those. With potentially millions of elements in a matrix this has a big impact on the memory usage. A good example of such a storage format is the *Compressed sparse row* storage format (CSR). However, the problem of data intensity still prevails. Storing the indexing information does not help in that regard, but is of course vital for the kernel and much better than the alternative in terms of memory footprint. The format should also suit the architecture that is to execute the kernel. When optimizing for speed this is also of utmost importance, not just taking care of the memory footprint alone. Therefore, even if OpenCL is used for the implementation, the format should suit whatever processor that is being targeted. It is obvious and anticipated that the same format will not be the best performer on both architecture-types found in CPUs and GPUs - architectures with big fundamental differences.

As a conclusion; for running SpMV on GPUs the obvious strategy would be to look at ways that can enable a decrease in data intensity, and at the same time arrange the data in a manner suiting the architecture of the chip (is it a vector processor, or a scalar processor — and so on). This is also

applicable to CPUs. If it is possible to exchange communication with computation on the GPU, to keep it busy and hiding the latency, this should be investigated. Secondly, by looking at blocking formats it should be possible to achieve another speed increase. This is shown in previous works; amongst others in [4].

3.5 Data formats of relevance for use with SpMV

In this chapter the layout of the matrix data formats to be used with the SpMV kernel is explained. All figures are adopted from [21], which also describes all formats, except the block version of the ELLPACK/ITPACK format (BELL).

3.5.1 Compressed sparse vector format (CSV)

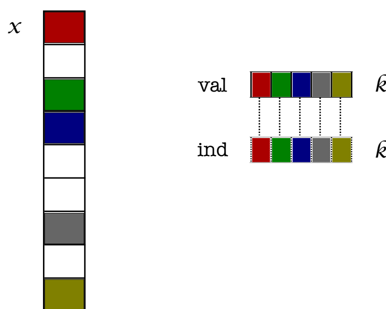

Figure 3.1: Compressed sparse vector layout.

A sparse vector consists of non-zero elements. In the compressed sparse vector format they are stored contiguously in an array. We call this array `val`. Further, the integer index for each non-zero is also needed, so that the whole original vector can be described. This is stored in the array `ind`. The layout of the *Compressed sparse vector* format is illustrated in figure 3.1.

3.5.2 Compressed sparse row storage format (CSR)

Here each row is stored as a compressed sparse vector. Three arrays are used. `val` stores the sparse row vector values, and `ind` stores the integer index, as in the compressed sparse vector format. In addition the third array `ptr` contains pointers to the first non-zero element of each row, indicating where each sparse vector begins in the `ind` and `val` arrays. The last element of `ptr` is equal to the number of non-zeroes. The layout of the *Compressed sparse row* format is illustrated in figure 3.2.

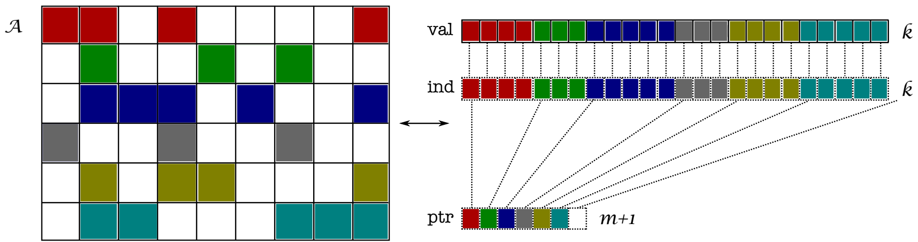

Figure 3.2: Compressed sparse row layout.

3.5.3 Block compressed sparse row storage format (BCSR)

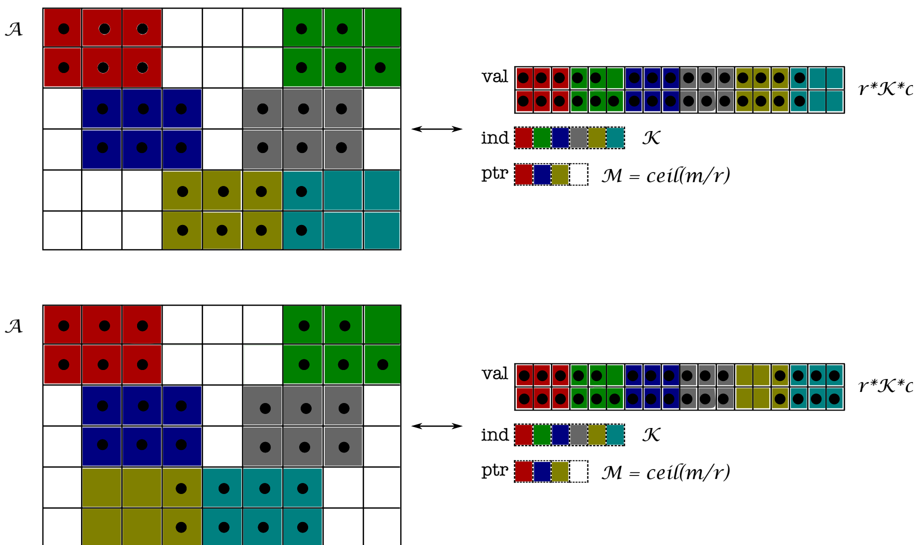

Figure 3.3: BCSR layout.

The layout of the *Block compressed sparse row* format is illustrated in figure 3.3. Block compressed sparse row storage (BCSR) is a further improvement of the CSR format. Here dense $r \times c$ sub-blocks contains the non-zeroes. In the CSR format they were stored individually. In BCSR a CSR matrix is, as described in [4], statically divided into $\left(\frac{m}{r}\right) \times \left(\frac{n}{c}\right)$ sub-blocks. These blocs are explicitly padded with zeroes as needed. In figure 3.3 the non-zeroes are indicated with black dots. Now, each block is stored in sequence, beginning with the upper left block, in the array `val`. The figure shows 6 blocks, which corresponds to the value of K. The array `ind` contains the column index of every (0, 0) element of each block. The array `ptr` contains the offset for the first block in a given block row, where first element contains offset for first block row and so on. Figure 3.3 shows

two different blockings, both with origin from the same matrix A. As [21] explains, blockings are not unique.

3X3 BCSR

Figure 3.3 illustrates a 3×2 BCSR. A 3×3 BCSR would simply be to use 3×3 blocks instead.

3.5.4 ELLPACK

The ELLPACK format is described in [21], as the other formats above. Figure 3.4 illustrates the format. The structure of it is quite straight forward. Two arrays are used, `val` and `ind`. The arrays have the same dimensions, m x s. Here m is the number of elements in the original matrix in the vertical direction, and s is the maximum number of elements in any row. Now each non-zero at the matrix in a row i is stored consecutively in `val`, also at row i. Are there less than s non-zeros in any row, the rest of the row is filled with zero values. This is also done in the `ind` array, which holds the index position of each value `val`[i, j] in the corresponding `ind`[i, j] location. The optimal case from a flops and data movement perspective is when each row has a number of elements close to s.

3.5.5 Block ELLPACK storage format (BELL)

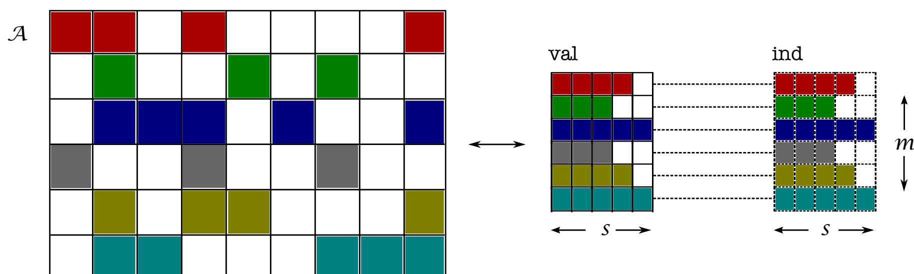

Figure 3.4: ELLPACK/ITPACK layout.

This is an further improvement of the ELLPACK format, which originally was developed to suit vector processors. As explained in [4], a blocked version adds the advantages of the dense subblock storage found in BCSR contributing to reduced index-data size. All while still being in a format suitable for a vector processor, something [20] argues the modern GPU can be looked upon as. The BELL format is not described in [21]. The format is introduced in [4], which is the source for the description in this text.

The steps taken to transform a matrix into the BELL format is illustrated in figure 3.5. Say we have an input matrix A. Organizing this into dense subblocks of size $r \times c$ gives us matrix A′. Then A′ is reordered in a descending order in respect to the number of blocks per row, which gives us A″. At the final step shown in the figure, the rows of A″ is partitioned into $\frac{m}{R}$ non-overlapping submatrices. Each such matrix is of size $R \times \frac{n}{c}$. Now the sub-matrix is stored in a $r \times c$ blocked ELLPACK format, or in the ELLPACK format described above.

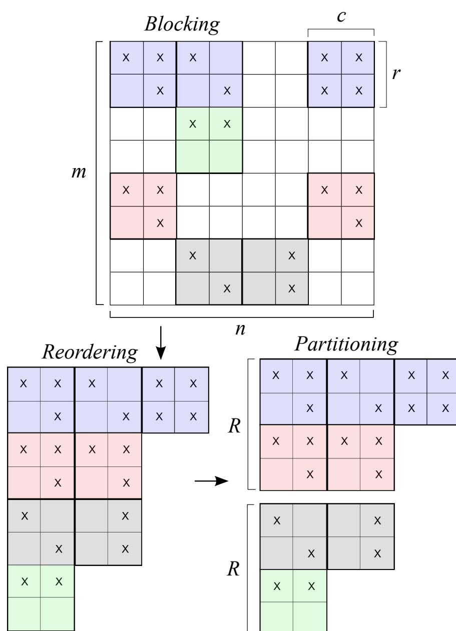

Figure 3.5: Blocked ELLPACK steps. Figure adopted from [4].

3X3 BELL

Figure 3.5 illustrates a 2×2 blocked ELLPACK. A 3×3 blocked ELLPACK would simply be to use 3×3 blocks instead.

3.5.6 Hybrid (HYB)

The hybrid format is a combination of the ELL and CSR formats. It is illustrated in figure 3.6. It is a custom format developed for the original CUKr implementation. Here ELL is used to store the regular parts, and CSR is added to take care of the few overshooting rows. This results in a format suitable for the GPU, as it is arguably a vector processor with SIMD(Single Instruction Multiple Data) processing, that still can take care of the irregularities by also utilizing CSR.

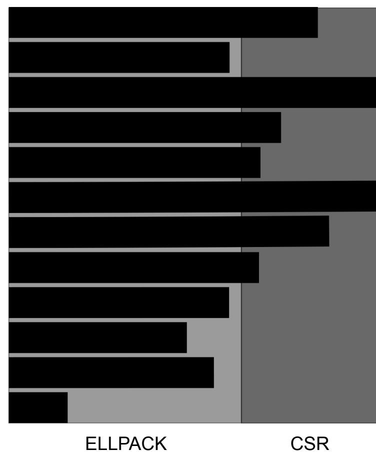

Figure 3.6: The HYB format. Figure adopted from [7].

3.6 The CUDA Krylov (CUKr) software version 1.0

In [7] the CUKr library is described as a prototype AKSI (*Accellerated Krylov Solver interface*) implementation. An overview of the software components and their relations can be seen in figure 3.8. CUKr is a library for writing Krylov solvers. It contains the building blocks required by these solvers, and supports execution on both CPUs and Nvidia GPUs through CUDA. The Krylov iterative solver is, as stated in the CUKr User's Guide ([6]), popular for the use in the field of finite element computation. It is also used in other areas where the matrix of the system to be solved is of such size that direct methods (which gives a precise solution) do not work. Iterative solvers can give good enough solutions with less computational work than in direct solvers. Krylov solvers on the computer are based on sparse-matrix vector multiplications (SpMV), dot products and vector updates [6]. All of these are to a high degree memory bounded. The actual computations needed to be done takes much shorter time than bringing the needed data from memory to the processor. One can say the nature of the sub-problems do not fit ideally with the actual ratio of computation to communication that is the ideal for these systems in order to utilize the processing power of the processor best. This is the reason why Krylov solvers on the CPU have a difficulty, reaching 10% system peak can be a challenge. GPUs are known for much higher bandwidth than current generation CPUs, an order of magnitude. This is why running the Krylov solver on a GPU is of high interest — and thus the goal for the CUKr library. The library makes it easy to construct a Krylov solver for use on the GPU, without any knowledge of GPU programming, or the construction of the parts needed for the Krylov solver, such as SpMV.

A good point stated in [6] is that researchers today within a given field that requires high performance computing are usually stopped by the lack of easy to use software or libraries. This is especially true for GPU computing, which is still in its infancy when it comes to application support and ease of use. Although they can easily have the budget to build a system that a few years ago were considered a supercomputer, for which to run their computations on, the needed software is missing or overly hard for them to develop.

CUKr is a scalable framework, and solvers written using the library can remain unchanged either if its used on one or multiple nodes. On each node it can utilize one or more GPUs or cores on CPUs, or a combination of the two. Any desired combination of data formats, BLAS libraries (BLAS routines that target certain hardware / uses a certain BLAS implementation) and precisions can be used. The precisions supported are single, quasi double and double. In quasi double mode two single precision values (floats) are used to store a double, here the mantissa is represented with 48 bits while a double does this with 53 bits, hence the term quasi, as described in [6]. This can be used to get higher precision on hardware that only supports single precision, such as older architectures.

Still, most commodity hardware available today runs much faster in single than double precision. Single precision ALUs are cheaper from a transistor perspective than double ones, and are thus outnumbering ALUs capable of doing double precision operations. This makes single precision operations faster (higher throughput). And, as especially true in these kinds of problems that are memory bound, faster because 50% less data needs to be used, also implying more data fits in cache. In computer graphics single precision is enough, but for scientific computing double precision is preferred. One can use *mixed-precision* and *quasi-double arithmetic*, or only one of them, to get a decent level of accuracy. The *mixed-precision* technique has to be applied with care at the right places, in order to give a good result (i.e. the effect of the usage is as wanted).

Mixed-precision uses the fact that in some cases most parts of the iterative loops can be done in a lower precision, without affecting the result. The parts sensitive for the final result and its accuracy are run in double precision. The result will be as if the higher precision was used all along in the computation. The use of *mixed-precision* in a Krylov solver can be implemented as *iterative-refinement*. Here, a high-precision correction loop runs outside a lower-precision solver.

Both *quasi-double arithmetic*, used to provide quasi double accuracy on single precision hardware, and *mixed-precision*, used to speed up the computation without considerable loss in precision, are supported in the CUKr library.

3.6.1 The structure of CUKr

In [7] the requirements of an AKSI implementation is stated as at least provide the following functionalities:

1. The possibility of using various types of many-core hardware, both CPUs and accelerators, as easy and transparent as possible.

2. Transparent data movement and coherency.

3. The emulation of higher precision and iterative refinement.

4. The possibility of scaling up to multiple accelerators and accelerated clusters.

In order to implement the CUKr library in a comprehensive manner that is expandable, the implementation is divided into different layers with each their responsibilities. A figure of the layout of these layers is shown in figure 3.7. The first requirement above is achieved with the use of multiple BLAS implementations, each for utilizing a kind of hardware or certain vendor delivered library optimized for their hardware (CPU or GPU). This is the bottom level layer seen in figure 3.7, the level communicating directly with the hardware through a library for it or custom code. It is called the BLAS level, and is the BLAS implementation for the particular kind of hardware, be it a CPU, GPU, or a kind of accelerator card.

3.6.2 The BLAS level

The BLAS level implements the BLAS functions for the certain targeted device and should exploit its potential performance as well as possible. Because of this, it is device dependent, and it hides this complexity from the other layers above, seen in figure 3.7. It gets its inputs and provides an output, or result — after a given period of time. This level provides wrappers for the various BLAS libraries or BLAS function implementations. This is the BLAS object, which enables the use of *abstract BLAS call*s, where *what* to be done is specified but not *how*. The latter is encapsulated inside a BLAS object, which knows which device to use, BLAS library, and precision for the operation. The information encapsulated in the BLAS object is shown in table 3.2.

3.6.3 The data structure level

The level above the BLAS level, as seen in figure 3.7, is the data structure level. Here the data structures needed by the Krylov solver are implemented. The structures include vector an matrix types. When matrices are stored in a compressed format they are represented as collections of

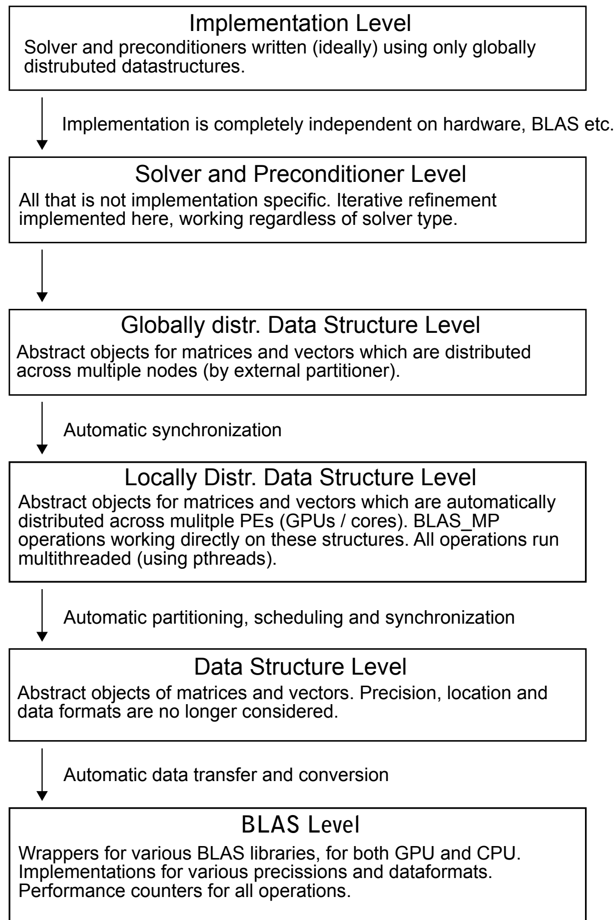

Figure 3.7: The layers of CUKr, adopted from [6].

vectors, as explained in [7]. In addition a mathematical Krylov solver also requires *scalars*. Information about data precision and data location (device location) has been abstracted out, so the data structure level is the highest level to deal with such. Description of these follows.

CUKR_VECTOR_SP

Table 3.3 shows the structure of CUKR_VECTOR_SP. The structure contains pointers to a vector, that can exist in different precisions and at different locations. For instance a double precision vector that resides in GPU memory, or a single precision vector that resides in system memory (i.e. on the CPU side).

Status contains information about where the vector exists and in which precisions. If the vector is needed in a computation but required precision

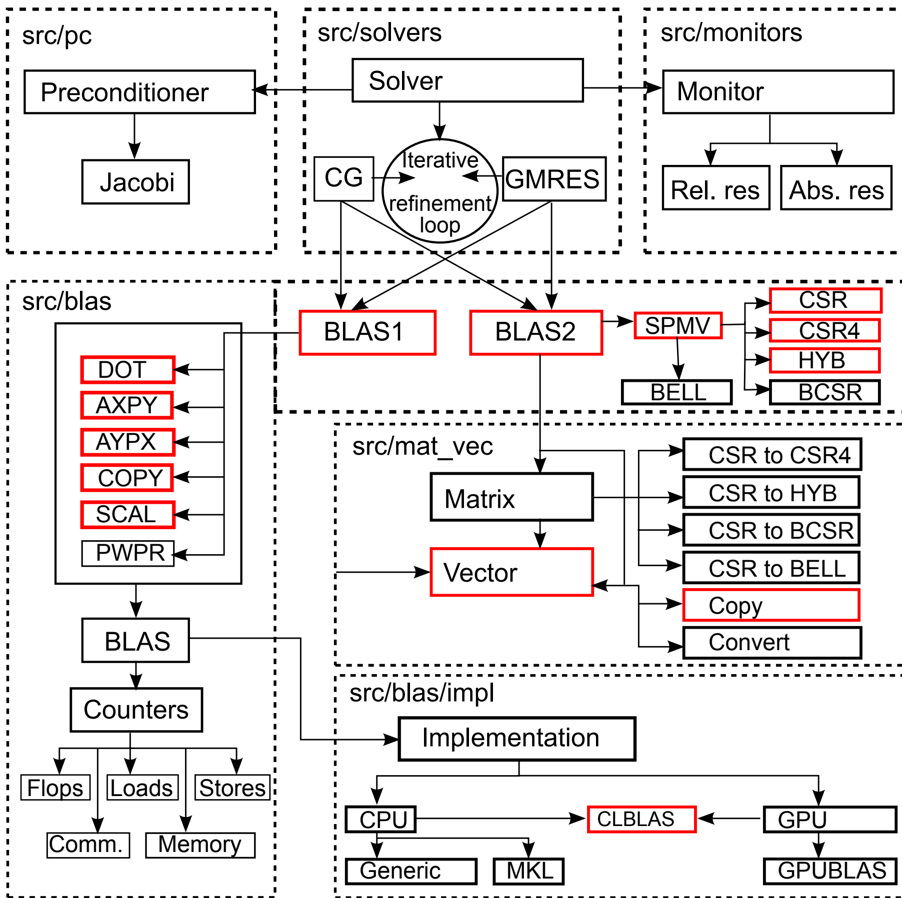

Figure 3.8: The block-layout of CUKr. Red boxes shows existing and new areas where work will take place during the implementation phase. The block-layout is adopted from a CUKr lab-meeting note by Serban Georgescu, with additions from the author to illustrate the new state.

does not exist at the required location, the data structure level makes sure a new vector in the required location an precision is created. For instance the GPU might need the double precision version, which already resides on the CPU. Then this value is copied over to GPU memory, and pointed to by `pd_dval`. If the needed vector is already in place nothing needs to be done. If there is no value at a location in a given precision, the pointer is a NULL pointer to indicate the non-existence. The *status* field is constantly updated to reflect the state (existence of the vector at certain location in a given precision).

Properties	Contains
Name	Blas name
Counters	Performance counters
Location	CPU or GPU
Precision	double, qdouble or single
Operations	DOT, AXPY, COPY, SpMV etc.

Table 3.2: CUKr BLAS object.

Properties	Contains	
n	Vector size	
name	Vector name	
status	CUKR_STATUS_CPU_DOUBLE	
	CUKR_STATUS_GPU_DOUBLE	
	CUKR_STATUS_CPU_QDOUBLE	
	CUKR_STATUS_GPU_QDOUBLE	
	CUKR_STATUS_CPU_SINGLE	
	CUKR_STATUS_GPU_SINGLE	
	CUKR_STATUS_CPU_INT	
	CUKR_STATUS_GPU_INT	
Data members	**CPU**	**GPU/CUDA**
Double	ph_dval	pd_dval
Quasi-Double	ph_qval(tail)	pd_qval(tail)
Single	ph_sval(head)	pd_sval(head)
Integer	ph_ival	pd_ival

Table 3.3: CUKR_VECTOR_SP data structure. The data members are pointers to arrays of scalars (float, double or int). This is also compatible with CUDA, as the kernels directly accepts pointers to the arrays where the data is stored on the device.

Properties	Contains
rows	No. of rows
cols	No. of columns
nz	No. of nonzeros
format	Matrix format
variation	Matrix format variation
Formats	**Member**
CSR	csr_mat
HYB	hyb_mat

Table 3.4: CUKR_MATRIX_SP data structure

CUKR_MATRIX_SP

Table 3.4 shows the structure of CUKR_MATRIX_SP. This structure holds the matrix in a given format. The matrix can automatically be converted to other formats if requested, when needed in a computation. Because of the share size of the matrices, once a matrix is converted to another format, the old format is deleted. If not the data would take up too much space. Thus, the matrix only exists in one format at the time, unlike the vector structure which can hold all precisions and locations. Since the matrices are built up of the vector structures, they exist in the precisions and at the locations their vectors exist in.

Chapter 4

Background for relevant hardware

In this chapter some of the current generation of programmable graphics hardware will be covered. We will look at the main-lines between the differences in hardware, and how the devices best utilize `global` memory — which is of importance for the tasks at hand given the memory bound nature they possess. The evolution of the graphics hardware leading up to todays generation will not be explained. For the interested reader please see [5] [1].

The first sections presents some current OpenCL capable graphics hardware. Tables listing each GPUs characteristics is found in Appendix A. Note that the performance listings is peak theoretical performance, real world applications will not fully achieve these speeds(given that they are not memory bound). There are two related reasons:

- Speed is based on multiply-add instructions or operations, which vendors count as two operations (all though in graphics hardware this is done in one instruction).
- All operations in a kernel are rarely *only* multiply-add operations.

A modern CPU of relevance will also be looked upon, the Intel Nehalem — and how to best utilize memory with this processor.

4.1 Nvidia OpenCL capable graphics hardware

4.1.1 Nvidia Tesla architecture

The Nvidia Tesla architecture was designed to be capable of not only graphics computations. An overview of the architecture is shown in figure 4.1.

[1] The project work leading up to this masters thesis.

The TPC (Texture/Processor Cluster) units consists of processing cores called SMs (Streaming Multiprocessors). They share a Texture unit and a texture L1 cache. The design is highly modular, and different chips based on this architecture has different number of TPCs — the number of these is directly related to the chips' performance level (both in frame-rates for graphics and general computing power), and the power usage of the chip. A laptop chip could sport *two* TPCs, while a high-end desktop chip like the GTX 280 had 10 such. The ROP (Raster Operation Processor) units showed in figure 4.1 are dedicated hardware units for doing rasterization operations, later in the graphics pipeline when the pixels for the screen are determined (rasterization for the screen is performed here), and are thus not utilized in GPU computing. They are implemented in hardware and are fixed function, for the speed it provides. The TPC illustrates the reason for the name Compute Unified Device Architecture (CUDA); it is a unified, or merged, unit that can do both graphics operations and general computations.

Geforce GTX 280

The structure inside the TPC unit in the GTX 280 chip is shown in figure 4.2. Each SM maps to a compute unit in OpenCL. The SM consists of 8 *scalar processors*, and has access to a shared memory as seen in figure 4.2 — the local memory in OpenCL terms. Notice also the DP; a *double precision* floating point unit (FPU). The ratio between the DP and SPs, 1:8, explains the 1/8th *double precision* performance compared to *single precision* performance. The SFUs (*Special Function Unit*) is for(amongst others) transcendental operations; sine, cosine, logarithm and so on. The SM utilizes Single Instruction Multiple Data(SIMD) processing to instruct the cores, the MT issue unit is responsible for this. The characteristics of this card is seen in table A.4, Appendix A.

4.1.2 Nvidia Fermi architecture

Nvidias new Fermi architecture contains ECC cache and memory, and also full IEEE 754 double precision floating point support. The Fermi-based chip made for scientific computing, found in the Tesla[2] M2070 computing module, has a double precision peak performance at about 515 GFlop/s (*billions* of floating point operations per second) – about half of its single precision performance. This is over a threefold the peak double precision performance of the AMD/ATI Radeon HD 4870 chip released sum-

[2]There must be for branding reasons that the Tesla name is still used on Nvidia cards meant for HPC. It can seem confusing that older cards in the Tesla series HPC cards were based on the Tesla architecture, and the newer cards introduced in the same series are based on the Fermi architecture. Nvidia has used the name Tesla for *two* different things — making it easy to mix architecture names with the card series name.

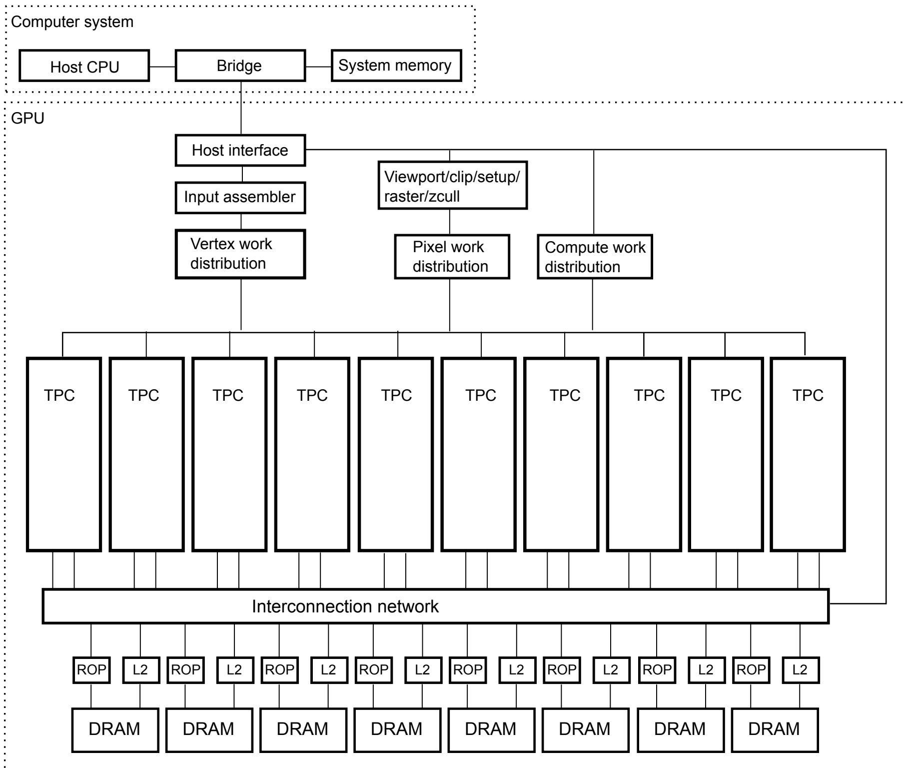

Figure 4.1: The Nvidia Geforce GTX 280 architecture overview. Illustration style is inspired by the Geforce GT 8800 figure in [15].

mer 2008. These additions are definitely showing Nvidias focus on making their GPUs even more suitable for High Performance Computing (HPC), also apparent by their collaboration with CRAY Supercomputers announced by CRAY in October 2009 at a CRAY workshop event in Tokyo.

Geforce GTX 480

The GTX 480, based on the Fermi architecture, has a double precision performance that is 1/8th of the single precision one. The characteristics of this card is seen in Table A.5 in Appendix A. The chip is a natural evolution from the one found in the GTX 280 card(as the Fermi architecture is a natural evolution of the Tesla architecture). Here, each TPC contains 4 SMs, in contrast to 3 found in the GTX 280. The total number of TPCs has also increased up to 15 (chip contains 16 TPCs, *one* is disabled during production to increase the number of usable chips).

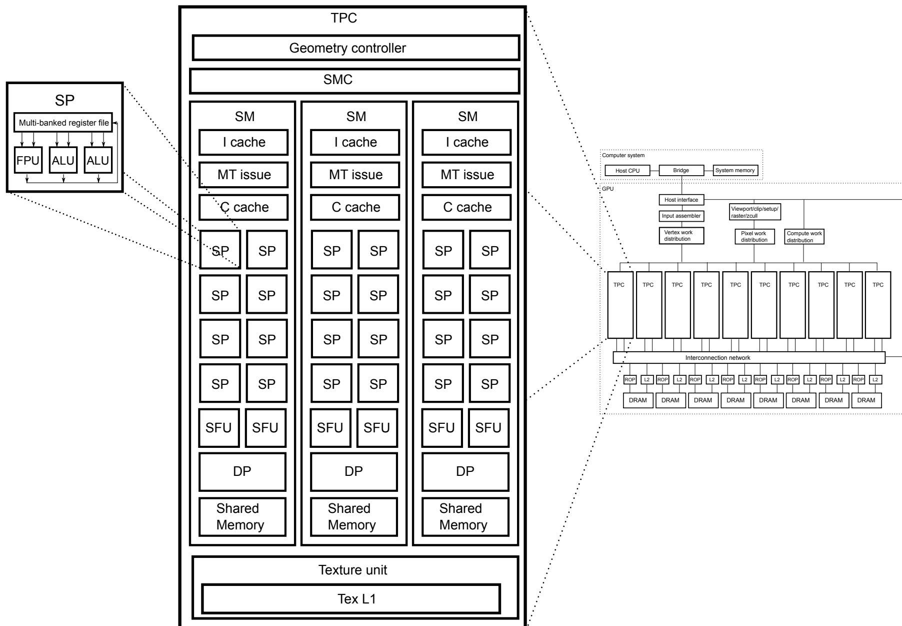

Figure 4.2: The Nvidia Geforce GTX 280 TPC. Illustration style is inspired by the Geforce GT 8800 TPC illustration in [15].

4.1.3 Ideal global memory access pattern

To utilize the memory bandwidth available in the Nvidia cards the memory access must be coalesced. For the memory access to be coalesced some rules must be followed. Coalesced memory access happens when work-items in a work-group accesses the memory in a manner where the addresses increase sequentially for each work-item. They each fetch their needed part of the global-memory. Rather than amounting to as many memory fetch operations as work-items, they all happen in one big memory read operation — the multiple requests are coalesced into one operation by the memory controller. On Nvidia hardware a *warp* is referred to a collection of 32 work-items or threads executing the same instructions on a compute unit (part of a work-group). A *half-warp* consist of 16 work-items, and it is these 16 work-items that can get coalesced memory operations at a time. The total size of the memory transaction is of 32, 64 or 128 bytes. This is further explained in [18]. Nvidia has historically[3] classified their devices according to *compute capability*. Higher version of *compute capability* is better, generally meaning the device gives more memory access flexibility and less restrains or requirements regarding how to access the data — while still providing the utilization of the bandwidth. For *compute capa-*

[3] After the first introduction of CUDA and CUDA-capable devices.

bility 1.2 or higher (both GTX 280 and 480 are in this category) coalesced memory access can happen for any pattern of addresses (sequential access, as described above, is no longer required). Here work-items can even access the same address and still get a coalesced operation. Also substantial improvements in how many memory transactions are needed if a *half-warp* tries to access words in n different memory segments are in place. Cards of a lower compute capability would issue 16 transactions — severely impacting the utilization of the memory bandwidth. In contrast the newer cards only issue one transaction for each segtment (n transactions). More details are found in [18]. Alignment is also required to get coalesced reads. For the built-in types this requirement is already followed. This means that the addresses must be a multiple of 4, 8 or 16.

4.2 AMD/ATI OpenCL capable graphics hardware

4.2.1 Architectural overview

ATI Radeon HD 4870

The recent ATI chips consists of something called SIMD engines by ATI. The 4870 contains 10 such entities. Each SIMD engine consists of 16 Stream Cores (SC), and these each consist of 5 Precessing Elements (PE). This gives 10 x 16 x 5 = 800 PEs, also called Shaders (when used with graphics) or ALUs (Arithmetic Logical Unit). The ATI Radeon HD 4870 GPU chip belongs to ATIs R700-family architecture, as do the AMD FireStream 9270 mentioned in [5]. The 800 PEs gives a vendor supplied theoretical single precision peak compute rate of 1.2 TFlop/s. The theoretical double precision peak compute rate is 240 GFlop/s (*billions* of floating point operations per second) — one fifth of the single precision rate. This is explained if we look at the SC, where only one of the five PEs is capable of double precision computation (also called a *fat ALU*). The 4870 was the first graphics card to make use of GDDR5 memory technology, thus increasing the bandwidth dramatically. The 4870s memory bandwidth is at 115.2 GB/s. The characteristics of the 4870 card can be seen in table A.2 in the Appendix. In an OpenCL centric view the SIMD engines are the compute units. Thus, each compute unit on the 4870 consists of 80 PEs. The 4870 is capable of handling 15872 concurrent threads, in hardware, sharing time on the available resources. Figure 4.4 illustrates a Compute Unit (SIMD engine), and the contents of a Stream Core. As explained in [17], the Stream Core is a five-way Very Long Instruction Word (VLIW) processor. In *one* VLIW instruction as many as five scalar operations are co-issued, keeping the PEs occupied simultaneously.

The 4870 is part of the R700 architecture, which is illustrated in figure 4.3. Notice the blocks marked as Compute Units(SIMD engines). The

16 SCs are seen contained inside each Compute Unit, each consisting of 4 PEs(indicated with thin lines) and 1 T-PE(indicated with a thicker line). The PEs can perform integer or single-precision floating point operations, and the T-PE can in addition perform transcendental operations such as logarithm, sine, cosine, and so on. If *one* double precision operation is to be performed, *two* or *four* of the PEs are connected together to allow this. This explains the 1:5 performance-ration between double- and single-precision operations of the chip, and tells us that only *one* double-precision operation can be performed by the SC at the time. In contrast 5 single-precision operations can be performed at the time by a SC.

The SIMD engines utilizes Single Instruction Multiple Data processing, something that does not imply SIMD instructions (like what is found in modern AMD and Intel processors — the SSE instruction sets). By using the SIMD processing the cost of fetch and decode of instructions are shared across many ALUs, who follow these same instructions for every cycle. This model suits modern graphics well, where many *items* share the same shader processing (performed in a SIMD processing fashion). The 16 Stream Cores processes 64 elements over 4 cycles, this is called a *Wavefront* by ATI. Work-groups have to be a multiple of this amount in size (number of work-items), if not the SIMD engine will be under-utilized and full potential of the hardware is not reachable.

For the 4870 the minimum global size should be[4]:
$10 SIMDs * 2 waves * 64 elements = 1280 elements$

For latency-hiding (which is of essence for efficiently utilize GPUs):
$10 SIMDs * 4 waves * 64 elements = 2560 elements$

ATI Radeon HD 5870

The Radeon 5870 chip was introduced early autumn 2009. The characteristics of the 5870 card can be seen in table A.3 in the appendix. Its design is a continuation of the 4870. Instead of 10 SIMD engines the chip has 20 such. This, of course, gives OpenCL 20 compute unites to utilize — and effectively doubles the amount of ALUs usable with OpenCL compared to the 4870, to a total of 1600. The SIMD engines and Stream Cores are in principle (at a *high* level) similar to that found in the R700 architecture). The memory bandwidth is a 33% improvement over the 4870, making it even more suitable for memory bound tasks. The higher clock-rate makes this chips peak performance more than twice than that of the 4870. Also, ATI has implemented more reliable memory utilization by using EDC (Error Detection Code) in the form of CRC (Cyclic redundancy check) checks

[4]Based on notes from Siggraph Asia 2009 in Yokohama, Japan.

on data transfers. This makes the card more reliable than previous ones for high performance computing where there is no tolerance for errors caused by corrupted memory values. The 5870 can handle 31744 concurrent threads. It is important to keep in mind that such threads running on GPUs are lightweight, and hardware makes sure of extremely fast switching between threads waiting to get processed. The hardware based thread management incurs an almost neglect-able performance overhead.

For the 5870 the minimum global size should be[5]:
$20SIMDs * 2waves * 64elements = 2560elements$

For latency-hiding (which is of essence for efficiently utilize GPUs):
$20SIMDs * 4waves * 64elements = 5120elements$

4.2.2 Ideal global memory access pattern

Like we have seen in the Nvidia based graphics chips, the ATI chips are able to coalesce the read from multiple addresses when requested from work-items — during the same cycle. In this way the cost of the memory read is amortized among all the Stream Cores reading. The data for all the Stream Cores thus gets fetched in one memory access, as explained in [17]. Looking at the graphics processing nature of the GPU this makes perfect sense, as the shaders each need data from different parts to work upon, and there need to be some efficient way of feeding all the Stream Cores(coalesced reads) — analogous to the need of an efficient way of instructing them as previously mentioned (SIMD processing).

To get coalesced reads from the concurrent accesses to memory addresses in global memory, the addresses must increase sequentially from one work-item to the next work-item participating in the read. These work-items are in the same wavefront, as it is within the same wavefronts these coalesced reads can occur. Also, the addresses must start on a 128-byte alignment boundary — as further explained in [17].

4.3 A more CPU-ideal global memory access pattern

While programming in OpenCL the kernels will be able to run on a diversity of hardware. This does not imply one will get equal level of performance relative to each device' performance capabilities; the kernels have to be constructed in a way so they exploit a certain architecture – both in regard of algorithm and data-structures (the latter is especially true when dealing with memory bound problems as in this thesis). This custom manner of programming that works well on one device might not be beneficial

[5]Based on notes from Siggraph Asia 2009 in Yokohama, Japan.

for another device (say a common CPU). For kernels not used in HPC this is less of an issue, and there is more headroom for the programming of the kernel. However, when programming for performance it is expected that attaining competitive performance on both a CPU and a GPU comparable to other implementations each targeting only a certain device (and by only using the same kernel) can be overly hard if not practically impossible. The author notes that GPU devices are the ones with the largest constraints and least flexibility regarding programming for performance. CPUs are somewhat more flexible, but also here it is expected that the access pattern of the GPU will severely impact CPU performance.

The access on the GPU is in a coalesced manner to gain the bandwidth utilization on these architectures. This results in CPU cores attempting to read with the same access pattern (when running these kernels on CPUs), while using a dramatically smaller number of cores (4 or 8 typically on todays CPUs) — in contrast to hundreds on a modern day GPU. Of course, these memory accesses will not be coalesced, even though the access pattern is the same. The CPU architecture greatly differs when it comes to ideal memory access. And, as further described in [11], also here the access pattern is of high importance to utilize the bandwidth. Attaining highest possible bandwidth utilization can be a challenge. As shown in [10]; the changing of the burst size, the channel layout and the internal bank — while leaving the theoretical bandwidth intact — can have dramatic impact on performance. The more memory bound the problem is, the more dramatic this impact can become. This illustrates the problems with different CPU architectures, each having their own ideal memory access pattern — and thus the implementation challenges for memory bound problems. For the CPU(versus the GPU) — here at a more general level of detail, the much more ideal access pattern had been to let each core read a large number of sequential addresses, rather than single words at (seemingly to the CPU) random places, where different CPU cores try to read words next to each other in memory. We predict this access pattern ideal for the GPU to severely underutilize the potential memory bandwidth of the CPU.

The following sub-section explains, at a high-level, why it is necessary to handle memory different on the CPU if one is after performance here.

4.3.1 Memory access on the CPU

As the GPU kernels will not perform well on the CPU relative to other CPU implementations, we will look at the main reasons. We illustrate the memory access on the GPU and CPU if arrays positioned in global memory gets their elements accessed in a manner that should enable coalesced reads on a GPU. Following, simple figures illustrates the difference. How the GPU kernels will read memory while being executed on the CPU, in an un-optimal manner, is illustrated in Figure 4.6. On the GPU the mem-

ory access gets coalesced, as seen in Figure 4.5. The work-groups on the GPU fetches values from memory efficiently in this way. When the operations are done upon the current values fetched, a new set of values are read coalesced by the work-group — and processed in the same way. This continues until the hole vector is processed. Finally, Figure 4.7 shows what is a much more ideal reading pattern for the CPU. This pattern is implemented in a CPU AXPY kernel described in the kernel implementation chapter later.

Another reason for performance decrease is due to partitioning of the problem. On the CPU the ideal `global` and `local` partitioning sizes are much smaller. The ideal is to have *one* work-item per core. And total work-items equal to total number of cores in the system.

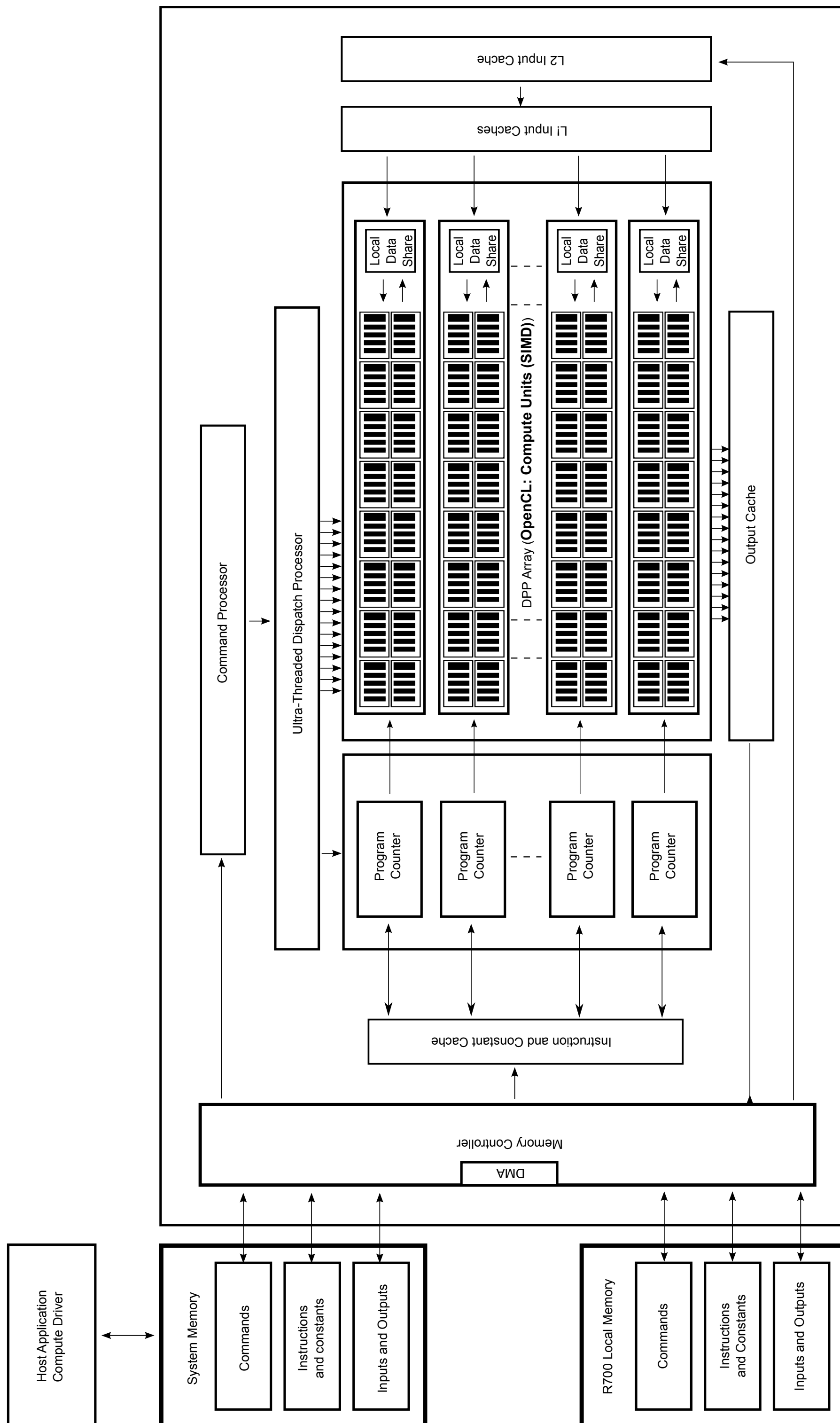

Figure 4.3: The R700 architecture figure adopted from [16]. OpenCL Compute Units marked, in addition.

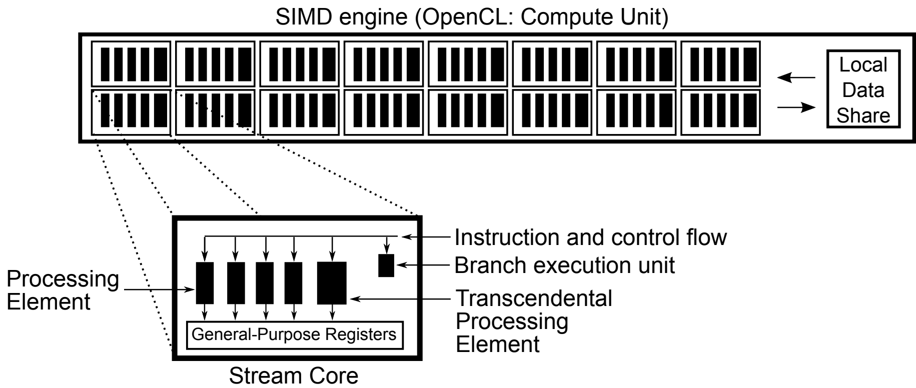

Figure 4.4: Illustration showing the SIMD element (Compute Unit) and the Stream Core. Partly adopted from [17].

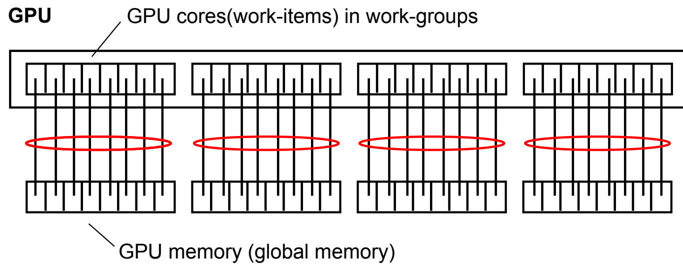

Figure 4.5: GPU coalesced read. The red circle indicates the memory requests that gets coalesced into one transfere.

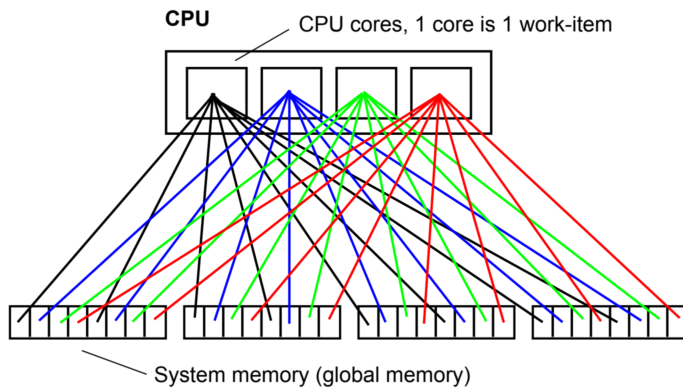

Figure 4.6: CPU read with GPU kernel. The chaotic memory access pattern arising when using a GPU kernel on the CPU is shown. CPU memory-bandwidth badly utilized.

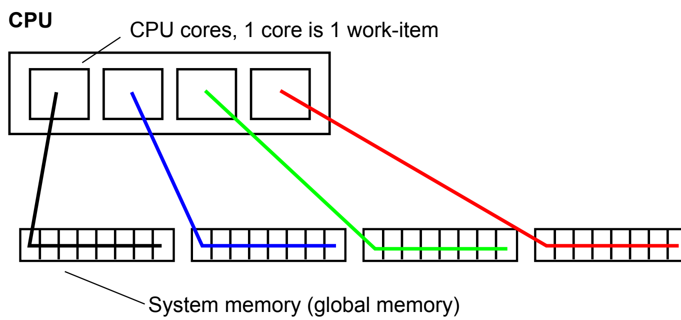

Figure 4.7: CPU ideal read with CPU kernel. Each core reads a large sequence of data in memory.

Chapter 5

Implementing OpenCL support in CUKr

When adding OpenCL support to CUKr there are several important aspects to take into consideration. First of all the addition should gracefully integrate with the philosophy behind the existing software system. It was chosen to add it to the software without replacing the existing CUDA implementation CUKr also can use. They live side by side, and one of the technologies to use must be chosen upon *cmake* configuration. This is covered in the first section.

The next section moves focus to the actual implementation itself. It was important to implement the OpenCL support in a way not interfering with CUKr's philosophy behind data movement and location — vital for the CUKr runtime. Differences in how CUDA and OpenCL deal with arrays as input to the kernels made changes to the `CUKR_VECTOR_SP` data-structure necessary (when OpenCL is used).

The last section looks at the additions to the BLAS level; the set-up the actual OpenCL kernels for each precision. The implementation of the kernels themselves are looked upon in the next chapter.

5.1 At the build level

Cmake is a platform independent system to generate build-files for software projects. This has been used to generate build files for CUKr since the first version, and is a good choice to continue using due to its strengths. Modules can be added to *cmake* so it can be used with a variety of software technologies and build options. For instance modules exist to integrate CUDA with *cmake* and configure its options — like the use of double precision or not, emulation or device mode, and so on. For OpenCL *cmake* should be able to find the include files and libraries when either running the build-file generation on Linux or OS X. On OS X these are always found

Listing 5.1: OpenCL include.

```
#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif
```

at the same place as Apple provides the OpenCL implementation on this platform. Under Linux this can differ as both ATI and Nvidia have their respective implementations. The build-file generation system will find the proper locations for all three of these mentioned configurations. It is not tested, but this should also work under Microsoft Windows. This is of value as we also want the build process to be as platform independent and flexible as possible. When including the header file for OpenCL the include-code shown in Listing 5.1 will suffice in all of the three configurations (Linux - ATI, Linux - Nvidia, OS X).

In the revised CUKr source code both CUDA and OpenCL lives side by side, this is made possible the use of `#ifdef`'s, like "`#ifdef CUKR_USE_OPENCL`" and "`#ifdef CUKR_USE_CUDA`". The pre-compiler can then make sure only the relevant code for the chosen technology (OpenCL or CUDA) is seen/-compiled by the compiler and thereby becomes a part of the CUKr library. The use of OpenCL and CUDA are mutually exclusive, the usage of both for the same build should not be configured.

5.2 Additions to the CUKr infrastructure and data-structure level

At first when CUKr is launched the OpenCL sub-system is initialized — if compiled with OpenCL support. A OpenCL platform is chosen(as a system might contain several OpenCL implementations), and a device supported by this implementation(OpenCL platform) is set up as target device(be it a CPU or a GPU, for instance). All the source-codes of the kernels are loaded into memory and their pointers handed over to OpenCL API-calls in order for the source-code to be compiled and built for the target device, the device that is associated with the OpenCL context just set up. Now memory objects for the kernels exist with their executables, and can be uploaded to the device and executed later, when set-up and called(with proper input data and a domain partitioning; `GLOBAL` and `LOCAL` sizes). Next we will look at the reoccurring input data to the kernels; the vector — how the infrastructure handles the vectors at the OpenCL level.

The vector data-structure is a foundation in CUKr, as explained previously. It stores pointers to the vectors on different devices (CPU or GPU)

and knows where these vectors are and in which precisions. The CUDA kernels accept ordinary pointers to arrays directly, pointers returned by CUDA allocation functions. So, the CUKR_VECTOR_SP data structure can keep these pointers that are pointing to locations on the GPU as they are ordinary `float`, `int` and `double` pointers. OpenCL does not deal with pointers in this manner for the kernels' input arrays; all arrays must be set up with a `cl_mem` memory object, and these objects are passed on to the kernels as arguments. To accommodate for this difference, that breaks with the way CUKr works, the CUKR_VECTOR_SP data structure has been modified to store pointers to `cl_mem` objects that contain the arrays in the different precisions. The revised CUKR_VECTOR_SP data structure is shown in Table 5.1.

Further, CUKr must know how to deal with these `cl_mem` objects, so that the CUKr runtime works properly with these too. A source file part of the software deals with all the vector operations at single GPU/CPU level; either if the vector being handled is in system memory, or device memory (and being used by OpenCL or CUDA), the appropriate handling code is contained here.

The function `CukrVecspMallocGPU` is used to allocate a vector of a certain precision on a device. When OpenCL is used it creates the appropriate `cl_mem` buffer. `CukrVecspFree` frees the buffer when done. Then there are the functions:

- `CukrVecspCopyDataCPU2GPU`
- `CukrVecspCopyDataGPU2CPU`
- `CukrVecspCopyDataGPU2GPU`

They copy the vector from device to device (where devices as read from the function names just mentioned), and possibly to a new precision in the process. The code can be found in the code-listings in the Appendix.

5.3 Additions to the BLAS level — the set-up of the OpenCL kernels

Before any kernel can execute it must be properly set-up(as previously mentioned), with its input data and partitioning(`NDRange` domain). The input-data is set up with pointers to the data object and the size. This is passed on to `clSetKernelArg`. The next step is to set up the `LOCAL` and `GLOBAL` sizes of the kernel domain. This defines the size of each work-group, and how many work-groups in total is to be used.

Now `clEnqueueNDRangeKernel` can be issued, enqueuing the kernel for execution. An event is attached to the kernel launch, and used by

Properties	Contains		
n	Vector size		
name	Vector name		
status	CUKR_STATUS_CPU_DOUBLE		
	CUKR_STATUS_GPU_DOUBLE		
	CUKR_STATUS_CPU_QDOUBLE		
	CUKR_STATUS_GPU_QDOUBLE		
	CUKR_STATUS_CPU_SINGLE		
	CUKR_STATUS_GPU_SINGLE		
	CUKR_STATUS_CPU_INT		
	CUKR_STATUS_GPU_INT		
Data members	**CPU**	**GPU/CUDA**	**OpenCL (cl_mem)**
Double	ph_dval	pd_dval	pcl_dval
Quasi-Double	ph_qval(tail)	pd_qval(tail)	pcl_qval(tail)
Single	ph_sval(head)	pd_sval(head)	pcl_sval(head)
Integer	ph_ival	pd_ival	pcl_ival

Table 5.1: CUKR_VECTOR_SP data structure with new additions for OpenCL support; cl_mem object pointers for referencing vectors for use with OpenCL added. Note that OpenCL cannot use ordinary pointers that references arrays on the device, therefore cl_mem objects are used to store the data.

`clWaitForEvents` to wait until the kernel event is over (it has been executed). This set-up process is performed for every OpenCL kernel needed by CUKr, for all precisions.

Chapter 6

Kernel implementations

This chapter will look at implementations of the OpenCL kernels, at the BLAS-level in CUKr. In the first section we will look kernels ideal for the GPU, and how this OpenCL port implements them. Then we discuss differences between the OpenCL and CUDA kernels that will directly influence their performance. The next section looks at changes that must be made to the memory access pattern to better accommodate the CPU, based on the differences between GPUs and CPUs and how they attain best memory bandwidth utilization. In the next chapter the results of the implementations are covered.

6.1 CUKr OpenCL kernels ideal for the GPU

In this section we will explain the implementation of the actual compute kernels, and how they both differ and are similar compared to the CUDA ones. In order to be able to do a evaluation and performance comparison against the CUDA versions, it is desirable to keep the codes as similar as possible. At certain areas the technologies' differences can prevent this. We will see the different performance characteristics. Similar to all the kernels (except COPY[1]) is the setup of the following variables, and the use of a similar `for`-loop for reading in data, as seen in Listing 6.1. The implication of this is explained in the following sub-section. A later section will explain how this way of reading in data (code in the kernels that is in the *for*-loop) badly affects the performance when running the kernels on the CPU.

For the sake of simplicity we look at the `single` kernels. The `quasi-double` ones differ in that they also handles the tail part, and uses special double-single add, multiply and subtract operations that do not neglect the higher precision given by the tail part. `double` kernels are similar to the `single`

[1]The BLAS COPY-function is actually implemented with a OpenCL API-call to copy a memory object on the compute device, and not with a actual OpenCL kernel, which would have been slower.

Listing 6.1: Common kernel properties.

```
/* Starting point for this work-group */
int ctaStart = get_group_id(0) * get_local_size(0);

/* Total no. of work-items in the kernel */
int totalThreads = get_global_size(0);

/* Get current local work-item id */
int tx = get_local_id(0);

/* Read the data (full lines) */
for (int i = ctaStart + tx; i < n; i += totalThreads)
{
  .
  .
  .
}
```

ones, but uses `double` variables instead of `float` variables. All kernels can be found in the code-listings in the Appendix.

6.1.1 Common structure

Before the code shown in Listing 6.1 is executed as part of the kernel, the kernel itself is set up for execution. This implies setting up the size of the work-groups (how many work-items/threads in each) and the total number of work-items. Since CUDA defines the block-size (OpenCL: work-group size) and and the number of blocks, we use the same for the OpenCL kernels. This is overall easier to deal with as the code inside each kernel works from this knowledge (how many work-groups and their size, rather than number of total and local work-items) — it ensures the comparison with the CUDA version is easier.

When each work-item runs the code in the kernel the value `ctaStart` is set to the global id of the first work-item (the one with local id of 0) of the current work-group (that the executing work-item is in). Note that this is stored in private memory for each work-item. Next the `totalThreads`-value is set to the total number of work-items (global size). Thereafter `tx` stores the local work-item id. We now see that the `for`-loop starts with `i` set to the global id of the current work-item and increments with the number of global work-items, as long as the vector-size `n` is not surpassed for the following iteration.

AXPY (and AYPX)

As AXPY and AYPX are virtually identical, we only cover AXPY. Inside the `for`-loop of Listing 6.1 we have the line

```
y[i] = y[i] + a * x[i];
```

The CUDA version uses local memory (shared memory in CUDA terms) to first read in the data. This was not done in the OpenCL implementation as it degraded the performance, so the arrays worked upon are read right from global memory. The loop will make sure each work-item in the work-group reads in consecutive values of `y` and `x` coalesced, does the addition and multiplication needed, and then writes back to `y`.

DOT

The DOT function is the most complex of the BLAS level 1 functions covered in this work, due to the need of reductions to produce the result. This is also a bottleneck. Inside the `for`-loop of Listing 6.1 we have the line

```
sum += x[i] * y[i];
```

`sum` is a private variable for the work-item, to which the product of each element of `x` and `y` are added. All work-items in each work-group will read the consecutive `x` and `y` values coalesced, put them in work-item level registers, and perform the operations. This will go on until the end of the vectors are reached, and each work-item has the `sum` containing the result of all work assigned to it (DOT operation done on its "responsibility area"). A local memory array for each work-group called `partial_sum` is then used to store each `sum` value from the work-items. All these needs to be added — reduced, at the device. For this a new `for`-loop, used for work-group level reduction, is utilized.

```
for (int i = get_local_size(0) >> 1; i > 0; i >>= 1)
{
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tx < i) {
    partial_sum[tx] += partial_sum[tx + i];
  }
}
```

Each iteration starts with a barrier to make sure all work-items have a coherent view of the local memory array being worked upon. Value `i` starts at local size divided by two (by right bit-shifting of the local size value). For each iteration value `i` is right bit-shifted, until it becomes 0. The loop runs as long as `i` is larger than 0. The body of the loop adds two consecutive elements of `partial_sum`, as long as the local work-item id is less than `i`. At the end of the loop all values are added, and the result is in

`partial_sum[0]` — the first element of the local memory array. At the end of the kernel a simple

```
if (tx == 0) {
  res_m[bx] = partial_sum[tx];
}
```

will make the work-item number 0 at each work-group write the work-groups reduced value to global memory array `res_m[bx]`, `bx` being the group id. This array has as many elements as work-groups, analogous to the local memory array that had as many elements as work-items in the group. The global memory array now contains each work-groups result, and a last summation (reduction) is needed. This happens at the host-side (CPU), after the contents of the global memory array are transferred to system memory where it is reachable by the CPU.

SCAL

SCAL scales every element of the vector `x`. Inside the `for`-loop of Listing 6.1 we have the line

```
x[i] = x[i] * a;
```

The elements are put into registers in a coalesced read, multiplied with `a`, and thereafter written back to `x`. Similar to the AXPY and AYPX operations, the use of local memory is found to degrade the performance.

COPY

As shortly mentioned, the COPY implementation does not use a kernel, rather a OpenCL API-call to copy the memory object. This is more efficient than invoking a kernel to copy data from one buffer (both already set up in global memory) to another. The code for this is as follows

```
err = clEnqueueCopyBuffer(ComputeCommands, *cl_d_x, *
    cl_d_y, 0, 0, n * sizeof(cl_float), 0, NULL, &
    scopy_event);
if (err != CL_SUCCESS)
{
  printf("clEnqueueCopyBuffer failed %d\n", n);
}

// Synchronize for timing
err = clWaitForEvents(1, &scopy_event);
if (err != CL_SUCCESS) {
  printf("clWaitForEvents failed!\n");
}
```

where cl_d_x and cl_d_y are source and destination buffers, respectively. It can be seen with the Nvidia OpenCL profiler that this saves time for the particular operation by a order of magnitude.

CSR

In the CSR kernels (4 variants exists for each precision) all the code is in the body inside the for-loop of Listing 6.1, thus accesses to global memory can be GPU-friendly. As explained previously in the section about the data-formats for use with SpMV, the CSR uses a pointer, index and value array for storing the 2D matrix in a compressed manner. In addition the kernel needs the vector x (being read) and y (being read and written to) as input. The for-loop starts by assigning iRowBeg and iRowEnd variables. These are set to the start and end address of the matrix row to be handled by the work-item (for the current iteration of the loop), respectively. The values of the pointer array will be read in a coalesced manner. Next, the column vectors are read and summed. A variable sum is set to 0 (for each iteration). A new for-loop iterates from iRowBeg to iRowEnd with an increment of 1. For each iteration sum is added the product of the j-th element of the value array (d_val[j]) and the corresponding value of vector x (the index is found in the index arrays j-th element). Note that these reads are irregular and will hardly result in any coalesced access. Especially the reads from vector x is highly scattered. *Kahan*-summation (also called *compensated summation*) can also be used in place of the ordinary one, and is activated by the use of a #define USE_KAHAN_IN_SPMV 1 (or 0 for ordinary summation as described). It has the property of reducing the numerical error when floating-point values are added. The summation is done and finally the value is written to vector y at its i-th element (now in the outer for-loop seen in Listing 6.1). Depending on the *beta*-value, 0 or 1, the y-element gets set to solely the sum or the sum *plus* the previous value of the y-element itself, respectively. If the *alpha*-value is not 1 it gets multiplied with the final sum and the result stored at the current y-element. The different *alpha* and *beta* values can make simplifications possible, and explains the 4 variants of the CSR kernel. The following code shows the Listing 6.1 for-loop contents in the case where *alpha* is 1 and *beta* is 0.

```
/* Read the beginning and end of the row
 * which will be processed by this work-item */
int iRowBeg = d_ptr[i] - 1;
int iRowEnd = d_ptr[i+1] - 1;

/* Read and sum for the column vectors */
#if USE_KAHAN_IN_SPMV
  float sum = d_val[iRowBeg] * d_x[(d_idx[iRowBeg])
     - 1];
  float c = 0.0;
  for (int j = iRowBeg + 1; j < iRowEnd; j++) {
    float y = d_val[j] * d_x[(d_idx[j]) - 1] - c;
    float t = sum + y;
        c = (t - sum) - y;
        sum = t;
  }
#else
  float sum = 0;
  for (int j = iRowBeg; j < iRowEnd; j++) {
    sum += d_val[j] * d_x[(d_idx[j]) - 1];
  }
#endif

/* Write the result to global memory */
d_y[i] = sum;
```

CSR4

The following private variables are declared in the CSR4 kernel:

```
float sum;
float4 val, x;
int4 idx;
```

These are used inside the `for`-loop of Listing 6.1 where we have the following code:

```
  // Read the beginning and end of the row which
  // will be processed by this work-item
  iRowBeg = d_ptr[i] - 1;
  iRowEnd = d_ptr[i+1] - 1;

  // Read and sum for the column vectors
  sum = 0;
  for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {

    idx = d_idx[j];
    idx -= 1;

    x = (float4)(d_x[idx.s0], d_x[idx.s1], d_x[idx.s2],
         d_x[idx.s3]);
    val = d_val[j];

    sum += dot(x, val);
  }

  // Write the result to global memory
  d_y[i] = sum;
```

We can see that the CSR4 variation differs from CSR in that the values from the index and value arrays are read 4 elements at the time by each work-item. This has a good effect on the performance of the kernel relative to the plain CSR. To enable this read of 4 and 4 elements, the index and value pointer arguments are defined the following way; as `int4` and `float4` type, respectively.

```
__kernel void kernel_sspmv_csr4_a1_b0(
      const int rows,
      __global const int* d_ptr,
      __global const int4* d_idx,
      __global const float4* d_val,
      __global const float* d_x,
      __global float* d_y
)
```

Although when these arrays are handed over to the kernel with the `clSetKernelArg` right before the kernel launch, the arrays are of type `int` and `float`. From the construction of CUKr viewpoint it is practically hard to let these arrays be in this format from the start when they are created by the CUKr run-time, so this is a good way of dealing with the issue. The CUDA version uses CUDA texture calls, which also groups the values so they are fetched 4 and 4 inside the kernel. Using texture fetching hardware in the GPU to read the values has a performance advantage. In OpenCL the equivalent would be done, if implemented to do so by the OpenCL implementors,

when reading data stored as `Image2d` memory objects [2] with OpenCL's *image samplers*. As OpenCL implementations keep improving over time, this is a good example of optimizations the compilers could incorporate; the use of texture-fetch hardware if available in the device, in cases where this is appropriate.

ELL

The ELL format is needed for the HYB format. It follows(as also the ELL kernel part of CUKr's C for CUDA code does) NVIDIAs ELL implementation written in C for CUDA.

HYB

The HYB format is a combination of the ELL and CSR4 format. So, the format(or rather its processing) is materialized at the level(or source file) where the functions setting up the kernels for execution resides, as there does not(naturally) exist a HYB version in the form of a single kernel. The ranges to be processed by each kernel, ELL and CSR4 respectively, are already decided by CUKr by the time needed data is handed over to the HYB set-up function. From here the ELL kernel is first set-up and completed, and right after completion the CSR4 is run to cover the remaining parts. Its completion completes the HYB function. The CSR4 kernel is described above.

6.2 Differences between the OpenCL and CUDA kernels

6.2.1 BLAS 1 functions

For SCAL, AYPX and AXPY local memory is not used. As will be seen in the next chapter; it did not increase the performance using it, rather decreasing the performance to some extent.

6.2.2 SpMV functions

For the SpMV functions no `Image2D` memory objects and samplers are used (texture memory and texture fetch, in CUDA terms). This is used for some of the arrays in the CUDA versions. This is partly due to CUKr's design more suiting CUDA memory handling (with ordinary pointers, rather than

[2] Images support was just recently added to the ATI Stream SDK with the new version 2.1 release of early May 2010, the previous version, 2.01, did not support this. This illustrates how new OpenCL actually is, with features important for performance in memory bound applications on th GPU just recently being added.

`cl_mem` objects) making the implementation of this not as straight forward. A work-around however is to implement the use of `Image2D` in the same functions where timing is started and ended. Here the ordinary (and selected) arrays must be converted from their memory buffer `cl_mem`-object to image buffer `cl_mem`-object, before the timing start so the comparison is fair compared to the CUDA version. This is not optimal for the overall CUKr performance, but will give correct and fair measuring, and acceptable as a research study. Though, for a software to be used in the real-world, some deeper modifications to CUKr would be more appropriate. Modification in such a way that the image buffer `cl_mem`-object would be created from the start for the vector array and not have to be converted at a later point (in the function just mentioned, where it is known what array to convert). Another reason is time restraints. The next desired step would be to use texture memory, on the same arrays as the CUDA version. This would give the most accurate comparison. And make it easier to tell true performance differences in OpenCL/CUDA implementations. Now, part of the differences must be assumed to be caused by the lack of use of texture memory and texture fetch. Though, by the differences to be seen in the BLAS1 functions, we can also to a certain degree assume there are OpenCL/CUDA implementation-differences related performance-differences reasons.

6.3 CUKr OpenCL kernels ideal for the CPU

In the hardware-chapter we visited ideal global memory access on both Nvidia and ATI graphics cards, and also what kind of memory access is better for the CPU. The kernels are ideal for exploiting the GPU memory bandwidth, by implementing a memory access pattern that suits the GPUs. It is sought to exploit the fact that the GPUs can deliver dramatically higher bandwidth — something of great need in memory bound problems like these. We mainly want to use the GPU for this sake. However, seeing how this memory access pattern affects performance when running the kernels on the CPU is of high interest. To get better CPU performance we look at a new AXPY kernel developed as part of this thesis where the access pattern better suits the CPU. The code is found in the Appendix, E.1.

Here, the two input arrays are accessed *four and four* elements at the time. This utilizes the memory bandwidth of the CPU better. As a consequence the kernel code gets more complex as there will be special cases depending on the total length of the input vectors. It must be able to handle all vector lengths, not only those divisible by 4. Each *work-item* (maps to one CPU-core) reads a sequential range from these arrays, which it works upon. Best performance on the CPU should be attained when there are as many total *work-items* as there are cores (a 1-to-1 relationship).

The next chapter shows the performance differences when running a CPU and GPU suitable kernel on the CPU, see section 7.4.

Chapter 7

Results

This chapter will cover results of the OpenCL kernels running on GPUs and CPUs, at the BLAS-level in CUKr. In the first section we will look at performance evaluation, describing what experimentation will be done. The next section explains performance measuring in CUKr. Thereby the actual performance results follows in the next sections.

7.1 Performance evaluation

A overall focus of the evaluation follows:

1. We will look at how the OpenCL kernels (port from the CUDA kernels) in CUKr performs on the same hardware, the Nvidia GTX 280 card(unfortunately we do not have access to the new Nvidia GTX 480 card based on the *Fermi* architecture[1]), relative to their CUDA original counterparts. If performance differs we will discuss the reasons for this. Use of a profiler to do analyzation of a running kernel is also included here.

2. A specific OpenCL kernel(running on all available CPU-cores) is written to try exploit a CPU better(basically by only changing the memory access pattern and partitioning of the problem domain, the `GLOBAL` and `LOCAL` sizes, of the kernel). This kernel will be measured up

[1]The new Nvidia GTX 480 card has not especially improved double precision performance as it is limited by Nvidia to segment the market, and the memory bandwidth is not increased much more than 25% over the GTX 280. For significant better double precision performance one must buy the version targeted for the scientific market(the Tesla 20-series), at a considerable increase in cost. But then again, the bottle-neck would be the memory bandwidth due to the nature of these problems and not the theoretical peak double precision performance. However, the author believe these cards will make it easier to utilize a higher percentage of the memory bandwidth available, as this architecture is kinder to irregular memory access due to improvements of the architecture — irregular memory access is something which the SpMV kernels possesses.

against the Intel MKL library(also running on all available CPU-cores).

There will be done two different kinds of benchmarking for the *first* (number 1) evaluation above. For the evaluation the following benchmarking methodic is used:

- The first is BLAS 1 benchmarking routines part of CUKr. This is to test individual AXPY, AYPX, DOT and SCAL performance. Here several consecutive runs are done with the same vector data for each kernel and the results are averaged. For small vector sizes this will give some degree of higher performance as parts of the data needed for the kernel can be found in cache (both on CPU and on modern GPUs). Running these tests with large vector sizes must be done to se the performance without the influence of the caches on the CPU. Differences seen here on small vector sizes can tell something about the ability to use the cache of the device.

- The second testing consists of running the complete Cg Krylov solver. In addition to the BLAS 1 kernels the CSR, CSR4 and HYB(CSR4 + ELL) SpMV kernels are also tested here(*one* SpMV format is chosen for each Cg run). When this is done real-world matrices from *structural analysis* problems are used, and two are from the area of *computational fluid dynamics*. Several different matrices from *The University of Florida Sparse Matrix Collection*[2] are used. They are categorized into medium or large sizes (and for the HYB-format measurement a small size is also used) — depending on their respective amount of *non-zeroe* elements. For every size category the results of solving the matrices in the given category are averaged. This is done in order to a higher degree give a more correct view of the real-world performance[3]. For every matrix the Cg solver is ran both with OpenCL kernels and CUDA kernels, and with different precisions and with different SpMV formats. The performance of the individual kernels involved are measured while running the solver[4] for each matrix, as well as the total Cg performance. From this data graphs are then generated. For the properties of the matrices used, please see Appendix C. Three matrices not in the table(not part of *The University of Florida Sparse Matrix Collection*) are also part of the benchmark groups; for the medium group: poisson3D_64, and for the large group: poisson3D_128 and poisson3D_192. As explained in [7] the Poisson equa-

[2]Please see their site at URL http://www.cise.ufl.edu/research/sparse/matrices/ for more information about the kinds of matrices available there and the repository in general.

[3]Note that this is not of utmost importance, as the goal is to look at OpenCL performance relative to other implementations. However, it is done to add a higher value to the results.

[4]A shell-script runs the solver executable and adds the results to file.

tion is found in many fields, amongst others computational fluid dynamics (CFD), particle based flows in computer graphics and steady-state heat dissipation. The matrix poisson3D_256 is not included, its file size is of 1.87 GB — not fitting in the memory of the GPU where the matrices are stored as 1D-vectors. Comparatively the poisson3D_192 has a file size of 761 MB, fitting well into the memory of current graphics cards. Note that when running all the Cg Krylov solver tests in this thesis, *no* pre-conditioner is used.

7.2 Performance measuring

The benchmarking mechanism in CUKr is straight forward. Before each BLAS invocation a timer is started. The function for launching the correct OpenCL kernel is called (this function is done when the kernel has completed its work). Note that the kernel is already built and ready to be uploaded to the device *before* the timing is started (this happens when CUKr is initialized), and the data to be used by the kernel is also already in device global memory at this point (taken care of by the CUKr runtime and its vector handling functions with the new OpenCL additions to allocate and deallocate appropriate `cl_mem` objects with needed data). The kernel setup function tells the kernel what data to be used and sets up the partitioning of the kernel (global and local sizes — *partitioning* of the problem). The kernel is complete, timing stops, and the elapsed time is accumulated. In benchmarking several runs are done to get more accurate results, and the the sum of time passed of all these runs are therefore stored. It is also known how many operations the particular BLAS operation requires, the size of the vectors being used, and the amount of total loads and stores to global memory — within the same function timing the kernel. By this information the actual performance of the BLAS operation is computed, and also the bandwidth used. At the node level total performance is accounted for(which can include use of several devices), and also at the MPI level if running on a cluster(including several nodes).

Table 7.1 shows the maximum theoretical peak performance that can be reached (in GigaFlop/s) for the relevant BLAS 1 kernels. Note that this is a theoretic scenario, where needed data is not reused and not in cache (as opposed to benchmark routines that run several consecutive times and finds an average, this leaves some data in caches). All data thus has to go through the memory bottle-neck — real world problems being solved will have similar properties. It is also assumed that the amount of data is sufficient(sustained delivery of data over time), and it is being read/written in optimal ways suiting the GPU, to utilize the bandwidth. Under these conditions the numbers represent the peak performances possible. The limit is computed by `Flop / Flio x Bandwidth (GigaBytes/second)`, where

	Nvidia GTX 280	Nvidia GTX 480	ATI 4870	ATI 5870
SAXPY SAYPX	2 / 12 x 141.7 = 23.6	2 / 12 x 177.4 = 29.57	2 / 12 x 115.2 = 19.2	2 / 12 x 153.6 = 25.6
SDOT	2 / 8 x 141.7 = 35.4	2 / 8 x 177.4 = 44.35	2 / 8 x 115.2 = 28.8	2 / 8 x 153.6 = 38.4
SSCAL	1 / 8 x 141.7 = 17.7	1 / 8 x 177.4 = 22.18	1 / 8 x 115.2 = 14.4	1 / 8 x 153.6 = 19.2
DAXPY DAYPX	2 / 24 x 141.7 = 11.8	2 / 24 x 177.4 = 14.78	2 / 24 x 115.2 = 9.6	2 / 24 x 153.6 = 12.8
DDOT	2 / 16 x 141.7 = 17.7	2 / 16 x 177.4 = 22.18	2 / 16 x 115.2 =14.4	2 / 16 x 153.6 = 19.2
DSCAL	1 / 16 x 141.7 = 8.86	1 / 16 x 177.4 = 11.09	1 / 16 x 115.2 = 7.2	1 / 16 x 153.6 = 9.6

Table 7.1: Maximum achievable theoretical peak performance for the memory bound BLAS 1 kernels (single and double precision given here, respectively), in GigaFlop/s.

`Flop` is floating point operations needed per vector element position to process, `Flio` is float loads and stores to global memory needed for each BLAS 1 operation when processing one such element position, and `Bandwidth` is the total bandwidth of the device between global memory to the device chip / processor itself given in billions of Bytes per second. The characteristics of the devices, including their theoretical bandwidth, is found in Appendix A.

7.3 Results BLAS 1 GPU-friendly kernels — individual benchmarks

In this section we show the BLAS level 1 kernel results when running the kernels through benchmarking routines. Of interest is how they perform relatively to other implementations, but also how they(the same kernels) perform on different hardware. These are kernels that are written to take advantage of the GPU memory bandwidth to a higher degree. Note the name `GPUBLAS` in the graphs, which is the CUDA based versions part of CUKr. The CUDA based versions are actually part of the CUBLAS library for all BLAS level 1 kernels, except for quasi-double (single-double) precision — here the kernels are custom and part of the CUKr CUDA sourcecode. The CUDA based AYPX is also custom for all precisions.

7.3.1 Nvidia GTX 280 under Linux, Nvidia OpenCL

For benchmarking the AXPY, AYPX and SCAL functions the testing was done accordingly:

- The benchmark routine was run with two versions of the OpenCL functions; one were local memory was used by the kernel to prefetch the vector values(similarly as done in CUDA versions), and one without the use of local memory at all. Here the vectors are read straight from global memory.
- Custom partitioning sizes were used when no local memory was used, and original sizes similar to those with the CUDA kernels in CUKr when local memory was used. This is to better suit the kernel and get higher performance in these cases. [5]

We want to observe effects of using and not using local memory. Further we will try to se the main-lines in the differences in performance seen in the OpenCL and CUDA kernels.

The AYPX kernels are good examples as the CUDA based kernels are not part of CUBLAS, rather part of the CUKr source-code. The OpenCL based implementation is therefor known to be similar to the CUDA based one when used with local memory, and in all of the three precisions. Looking at figure 7.3 we can see how the OpenCL performance trails the CUDA performance for each precision. Each pair of "precision trails" outlines an area, seen between the lines. The most of this area lies between 10 000(smallest sizes around here) and 1 000 000 in vector sizes. After passing a million in vector size the graphs eventually crosses.

Figure 7.1 gives a more detailed view of up to a million in vector size. We can see that this is the sensitive area for the OpenCL kernels, and that they star at about half the performance for the smallest vector sizes. Looking at figures 7.1 and 7.3 in comparison to figures 7.2 and 7.4 we can see the performance differences between not using and using local memory in the kernels. These observations are also similar for the AXPY and SCAL kernels.

We also take a look at the performance graphs of the DOT and SCAL kernels. AXPY is omitted as this function is virtually the same as the AYPX. Looking at figure 7.5 showing the DOT kernels, for vector sizes less than a million, we see that performance builds up slowly for both OpenCL and CUDA kernels. For single precision the start performance(about 10 000 vector elements) of the OpenCL kernel is about 375 MFlop/s, and about

[5]These sizes are called `global` and `local` sizes (in OpenCL terms, `block` size and number of `blocks` in CUDA terms). The sizes used in OpenCL can be seen in the source-code in the Appendix under Code Listings. `kernel_config_custom.h` and `kernel_config_orig.h` contains the new more optimal sizes and the original sizes, respectively. Note; this is for optimal use with Nvidia OpenCL and the GTX 280 card.

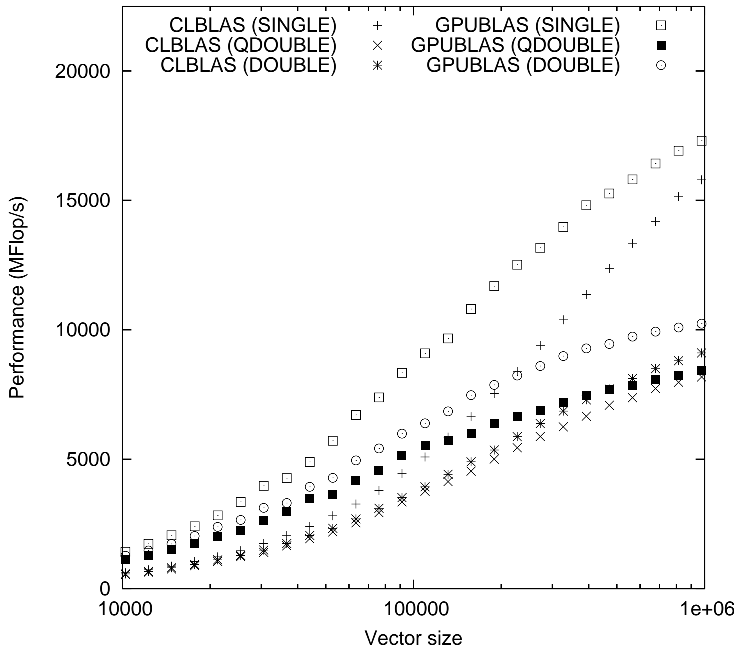

Figure 7.1: AYPX, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. Partitioning sizes are also adjusted to suit.

620 MFlop/s for the CUDA version. Also here the OpenCL version constantly trails behind. Figure 7.6 shows what this looks like for up to 21 000 000 in vector elements size. Notice here how the GPUBLAS graph for single precision reaches a peak and then the performance decreases by the biggest vector size. This can be attributed to GPU caches not being able to keep parts of the needed elements as the vector size is increasing, to the same degree as before when at smaller vector sizes.

Finally, figure 7.7 shows SCAL kernels performing with up to 21 000 000 elements. Notice how the OpenCL trails almost fall together until past size 100 000. The CUDA and OpenCL trails are clearly distinguished here.

From what is seen we can make two main conclusions:

- We have seen a difference in performance characteristics between the OpenCL and CUDA kernels. The OpenCL kernels start off with lower performance for the small vector sizes compared to the CUDA counter parts. This difference has the appearance of a constant cost factor. This can be due to some extra overhead in the OpenCL infrastructure (or implementation; in this case the Nvidia SDK 3.0 and

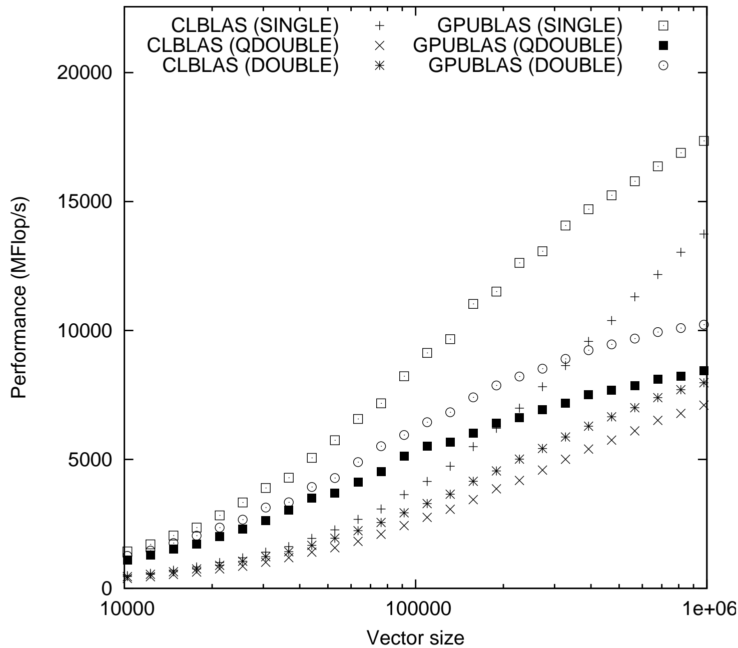

Figure 7.2: AYPX, OpenCL kernels uses local memory, as the CUDA kernel also does. Similar partitioning sizes as to the CUDA kernels are used.

its OpenCL implementation) compared to the mature CUDA C technology. Following the hypothesis that there is some higher initial cost related to OpenCL with this implementation, we can from the graphs see that this gets, to a degree, amortized with increasing vector sizes. Also, an explanation for this observed performance difference can be the use of cache in the GPU. There could be that the CUDA version somehow better utilize the caches in the GPU than the younger OpenCL implementation. As the benchmark routine runs 10 times for every BLAS 1 operation and finds the average(after a single warm-up run), some data would be left in GPU caches. Later when looking at real world problems, where data is hardly reused, we can get some indication. The Nvidia profiler is used in the following sub-section to see if we can analyze more of this difference seen, and maybe the cause.

- It was found through the benchmarking and studying the graphs that with the AXPY, AYPX and SCAL OpenCL kernels the use of no local memory gave the best performance. This was true for all preci-

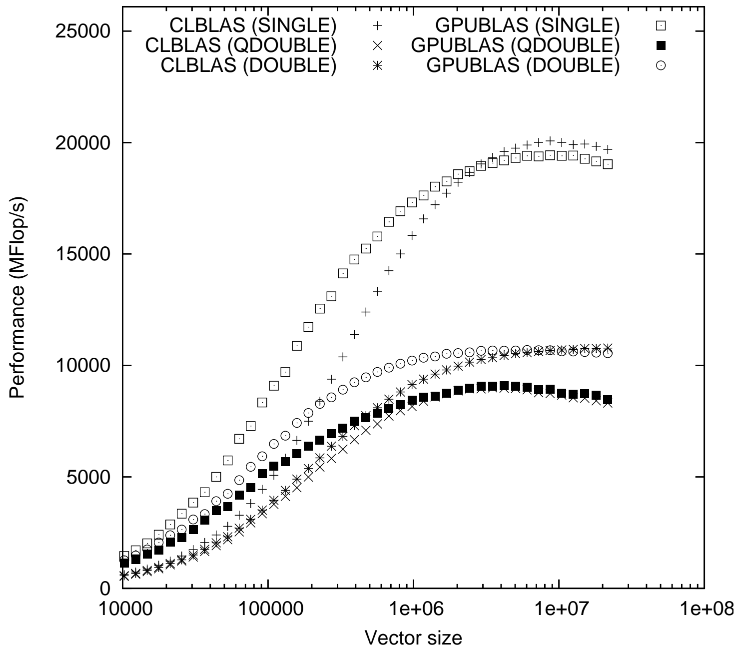

Figure 7.3: AYPX with large vector sizes — up to 21 million elements, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. Partitioning sizes are also adjusted to suit.

sions, and for both small(up to about a million vector elements) and large(up to about 21 million vector elements) vector sizes. It can be asked if this is due to less use of GPU cache when OpenCL is used with local memory, and thus rather than being a benefit the use of local memory then becomes a overhead factor.

Profiling BLAS 1 AYPX with Nvidia CL Profiler and CUDA Profiler

The profiling is done for both the OpenCL and CUDA single precision AYPX kernel. They are good for comparison, as the OpenCL based kernel is a direct port of the CUDA based version. The profiling is run on vector sizes all up to about one million elements. For each size 10 kernel launches are done. When profiling we will look at two important parameters; GPU Time and CPU Time. These are given in microseconds. From the Nvidia Visual Profiler help-menu these parameters are defined accordingly:

- GPU Time: It is the execution time for method on GPU.

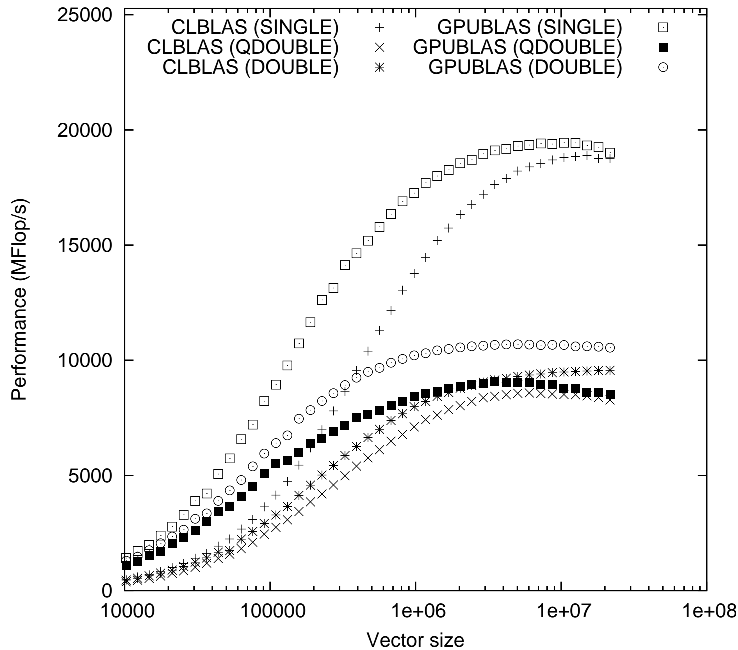

Figure 7.4: AYPX with large vector sizes — up to 21 million elements, OpenCL kernels uses local memory, as the CUDA kernel also does. Similar partitioning sizes as to the CUDA kernels are used.

- CPU Time: It is sum of GPU time and CPU overhead to launch that Method. At driver generated data level, CPU Time is only CPU overhead to launch the Method for non-blocking Methods; for blocking methods it is sum of GPU time and CPU overhead. All kernel launches by default are non-blocking. But if any profiler counters are enabled kernel launches are blocking. Asynchronous memory copy requests in different streams are non-blocking.

The profiling showed that the OpenCL kernel had on average close to 27 microseconds of CPU overhead to launch, per kernel call. In contrast this was 10 microseconds for the CUDA kernel. Overall the CPU overhead with OpenCL was constantly close to 20 microseconds more then for the CUDA kernel. In addition; each kernel call took close to 3 microseconds longer to execute on the GPU than with the CUDA kernel (GPU time). For larger vector sizes this increase in overhead becomes less noticeable as the GPU execution time becomes large. But at smaller sizes this constant factor is very noticeable, and explains the graphs previously seen in this

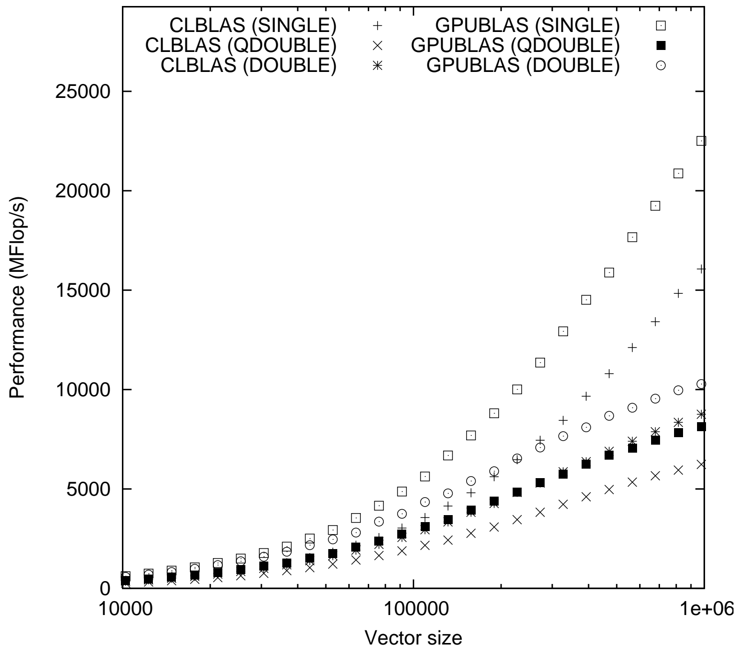

Figure 7.5: DOT; OpenCL vs. CUDA implementation.

section. Each vector size has 10 consecutive kernel launches(for being able to average results). Here it is also noticed a difference in CPU time for the first launch and the remaining 9 launches, in the CUDA version, of 15-20 microseconds. This is only observed for the smallest vector sizes(10 - 20 000 in vector size). For the OpenCL kernel the difference here is only 1-2 microseconds. This indicates that the CUDA version is able to cache the data for the smaller vector sizes and use this data already in cache for the consecutive runs, something the OpenCL version is not doing to the same degree. Similar behavior as seen here is also seen for the other BLAS 1 kernels. This is, in addition, confirmed from the previous performance graphs.

7.4 Results AXPY CPU-friendly kernel on CPU

Figure 7.8 shows the running of the AXPY CPU kernel on the CPU. For comparison, the AXPY GPU kernel is also run on the CPU, a Intel Core 2 Quad processor. CLBLAS_CPU shows the result. Here only one *work-item* per core us used, which should be the most ideal for the CPU(other parti-

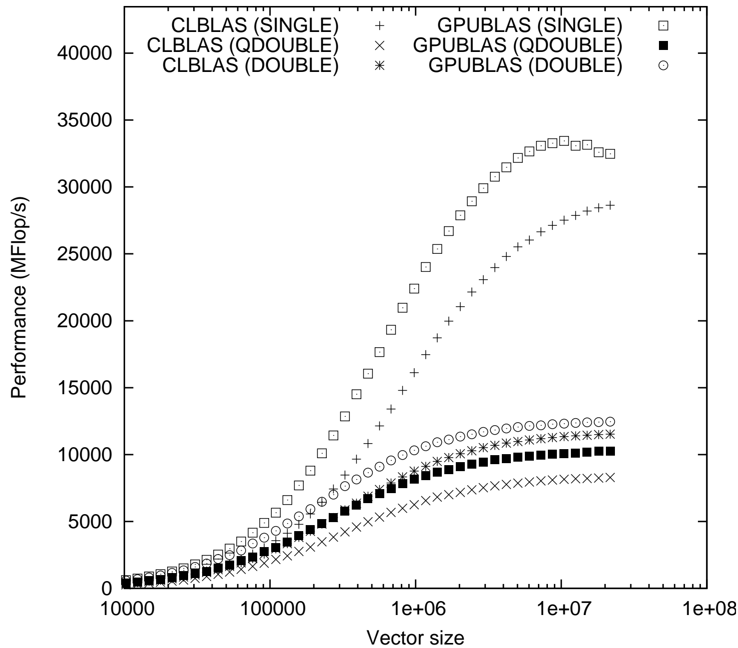

Figure 7.6: DOT with large vector sizes — up to 21 million elements; OpenCL vs. CUDA implementation.

tioning should incur more overhead). CLBLAS_CPU2 is the same kernel, running with "GPU-partitioning" (large `global` and `local` sizes). CLBLAS and CLBLAS_2 are the GPU-friendly version running on the CPU, with partitioning sizes optimal for CPU and GPU, respectively. Here the partitioning ideal for the CPU gives in many cases twice the performance. It is apparent how important proper problem domain partitioning is. Even though the kernel in itself is written to utilize the memory bandwidth well(though; the memory access can be influenced by the partitioning); a wrong partitioning can severely limit the performance of the kernel. Conversely it is seen how a sub-optimal memory access pattern affects performance, and how the use of right partitioning only to a limited degree helps the overall performance. As seen, the performance difference is dramatic as the memory access gets suited to the CPU. Using GPU access pattern severely impacts the performance. The intel MKL library is the top performer, and has a substantial stronger performance at small vector sizes. Once the CPU cache cannot hold the entire vectors, both MKL and OpenCL performance drops dramatically, and they have a sustained almost identi-

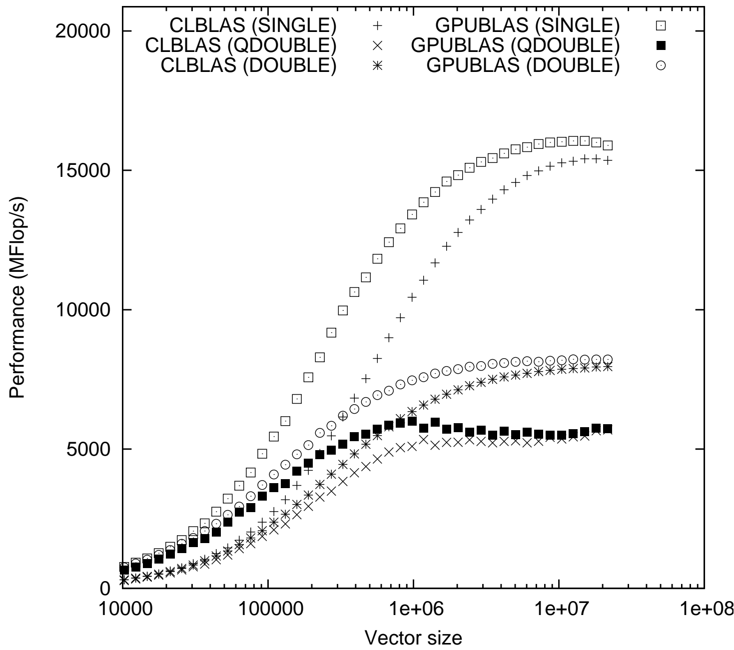

Figure 7.7: SCAL with large vector sizes — up to 21 million elements, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does.

cal performance level. At this point the performance the GPU kernel with partitioning ideal for the CPU performs about 50% of the CPU ideal kernel and Intel MKL. It is clearly seen how the GPU ideal kernel could not utilize the cache because of its memory access pattern, something the CPU friendly kernel does to a much higher degree. Given the overall performance differences it is apparent how the Intel MKL library is highly tuned to exploit the cache. The CPU OpenCL kernel is more agnostic of (other than reading sequential addresses) the cache. Throughout the testing the ATI Stream SDK 2.1 is used. Running this on Intel CPUs is not officially supported by ATI, and what impact this has on performance has not been investigated due to the timeframe of this project. The characteristics of the Intel Core 2 Quad CPU used is seen in Table A.1.

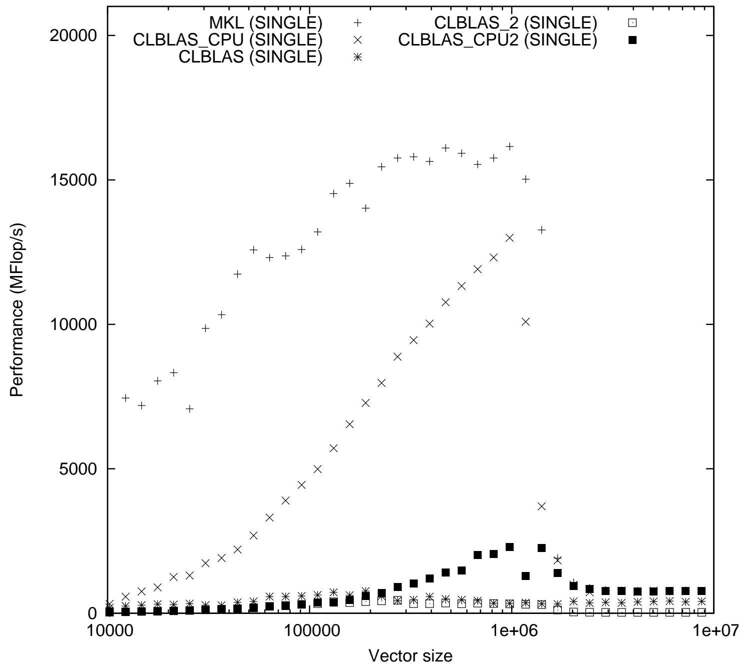

Figure 7.8: AXPY CPU-friendly kernel on Intel Core 2 Quad processor.

7.5 Results Cg Krylov solver and its GPU-friendly kernels — real-world problems

In this section we cover the results of running the actual Cg Krylov solver benchmark routine with real-world matrices as input (see Appendix C). Performances are given in MFlop/s.

7.5.1 Nvidia GTX 280 under Linux, Nvidia OpenCL 3.0 SDK

The benchmark is run on the Nvidia GTX 280 card, both when the CUKr library is compiled to use OpenCL, and CUDA. This is to have the comparison basis. We test for all three SpMV formats and kernels, CSR, CSR4 and HYB, in all three precisions — single, quasi double(single-double) and double. Figure 7.9, 7.10 and 7.11 shows the results for the CG with Hybrid kernels. Also here, as with the BLAS 1 kernels, the reduced OpenCL performance is visible. For the SpMV kernels the gap is even larger, except for when in quasi precision. In quasi precision one double is represented as two floats. This shows a less performance impact in the OpenCL Hy-

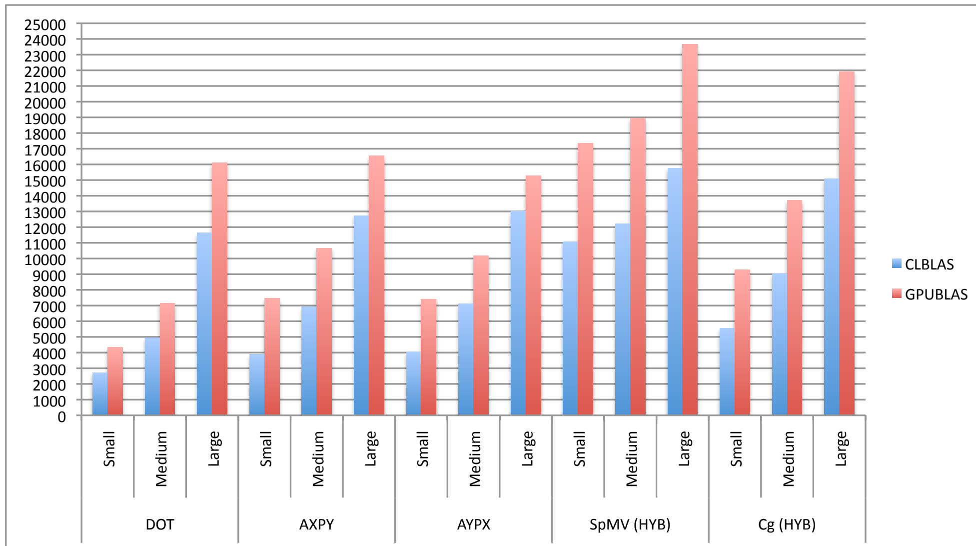

Figure 7.9: Cg HYB single precision benchmark result.

brid kernel than the CUDA based one, and the results here are closer to each-other. The overall lesser performance is mainly due to the non-use of CL-Image memory objects (storing the vectors) to enable efficient texture reads on the vectors, and partly due to the overhead as found for the BLAS 1 kernels.

Figure 7.12, 7.13 and 7.14 shows the results for the CG with CSR4 kernels. Here the CSR4 SpMV gap is quite dramatic, again part of this is caused by no use of CL Image in the OpenCL kernels, for texture reads. The OpenCL kernel will read the arrays in a 4-by-4 elements manner, but not use the texture memory hardware(image samplers) available for this. Also, the worsened effect of not using the ELL format for the non-regular rows is clearly visible.

Finally, Figure 7.15, 7.16 and 7.17 shows the results for the CG with CSR kernels. Interesting to see is how the OpenCL SpMV kernels as fast or even faster than the CUDA version. This tells us that using texture fetch has no benefit when the elements are read in a one-by-on manner, and that it actually might degrade performance a bit in comparison to a ordinary read(not using texture fetch hardware). The CUDA version shows a greatly improved performance when reading 4-by-4 elements with texture fetch(in the CSR4 kernels).

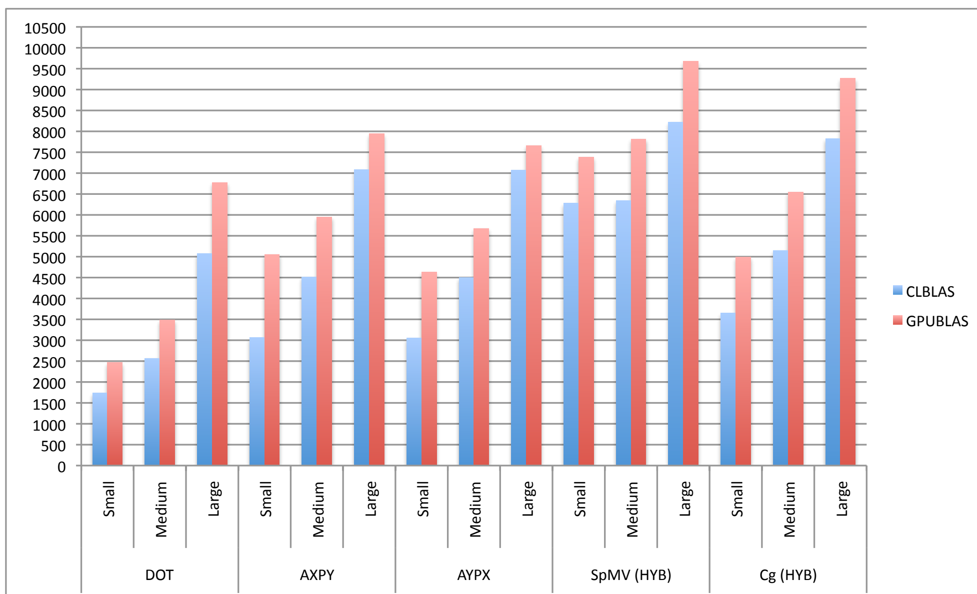

Figure 7.10: Cg HYB qdouble precision benchmark result.

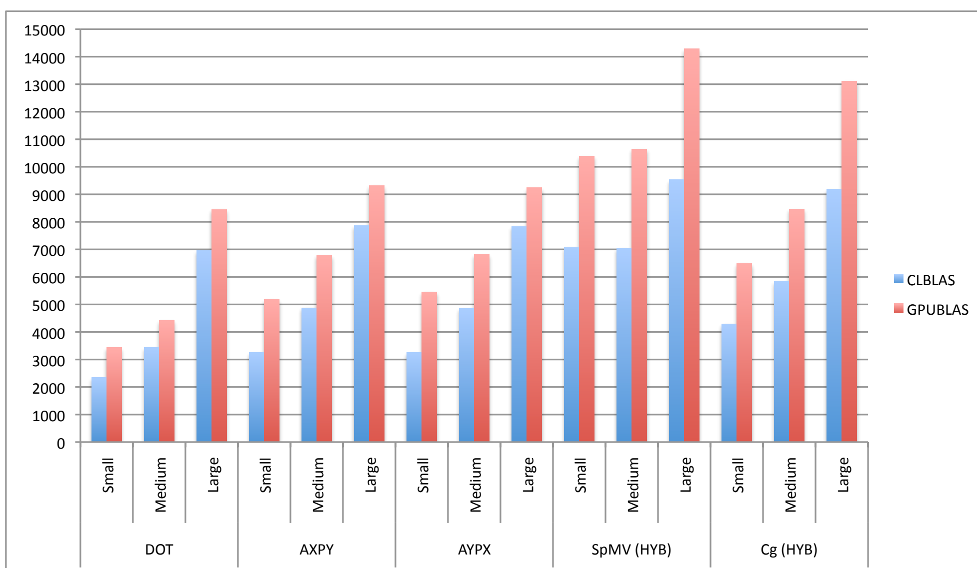

Figure 7.11: Cg HYB double precision benchmark result.

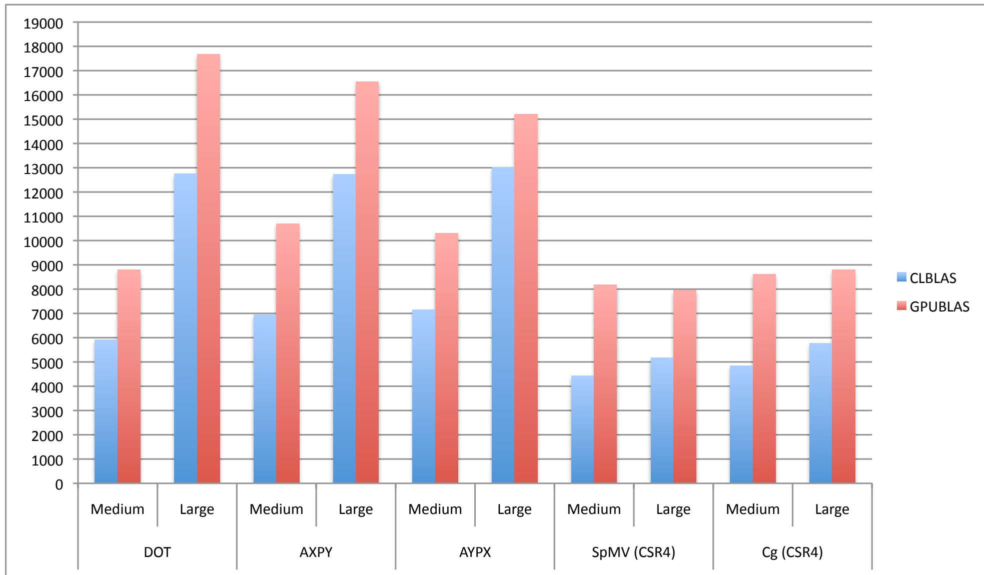

Figure 7.12: Cg CSR4 single precision benchmark result.

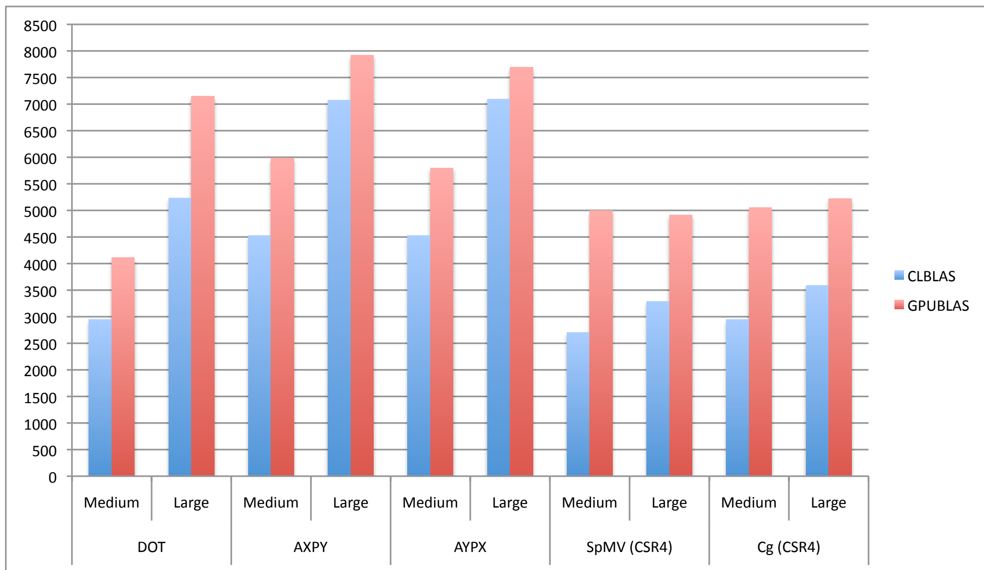

Figure 7.13: Cg CSR4 qdouble precision benchmark result.

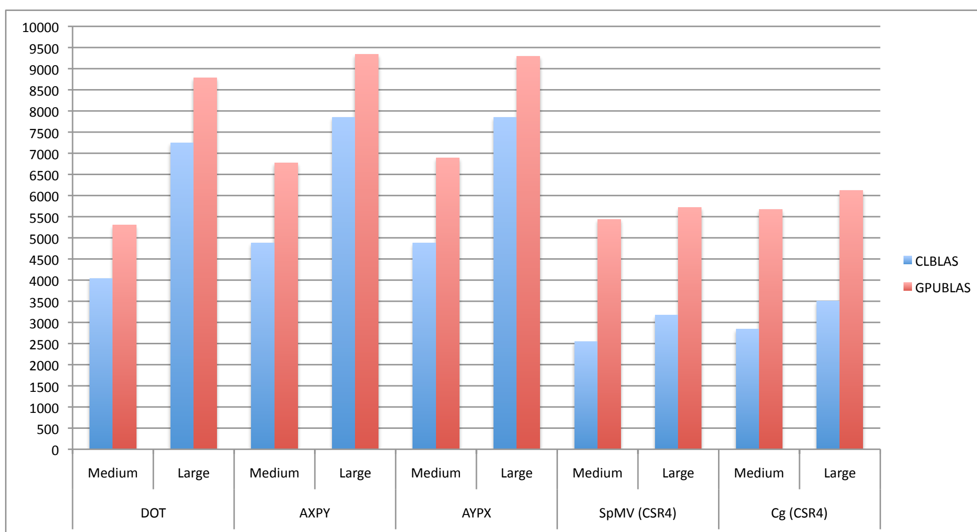

Figure 7.14: Cg CSR4 double precision benchmark result.

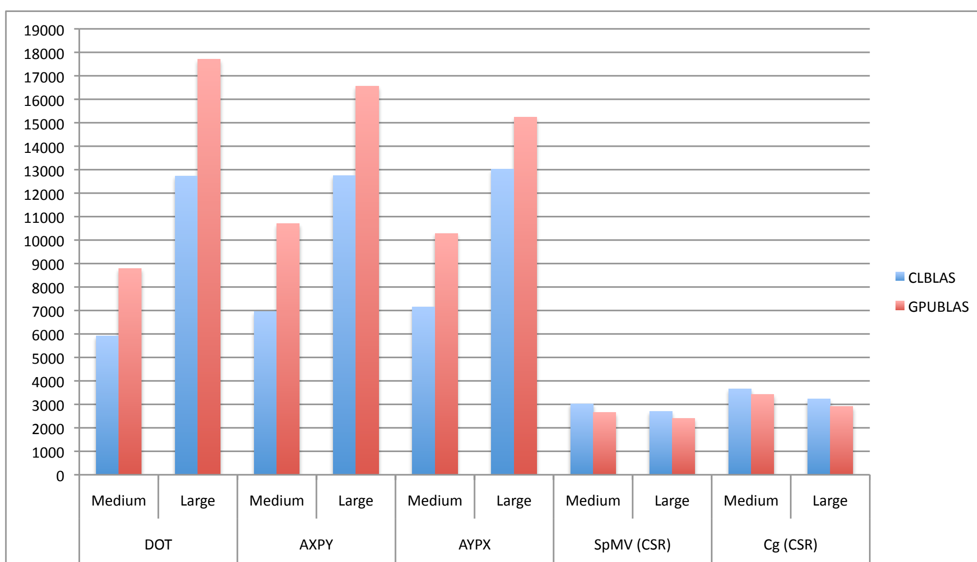

Figure 7.15: Cg CSR single precision benchmark result.

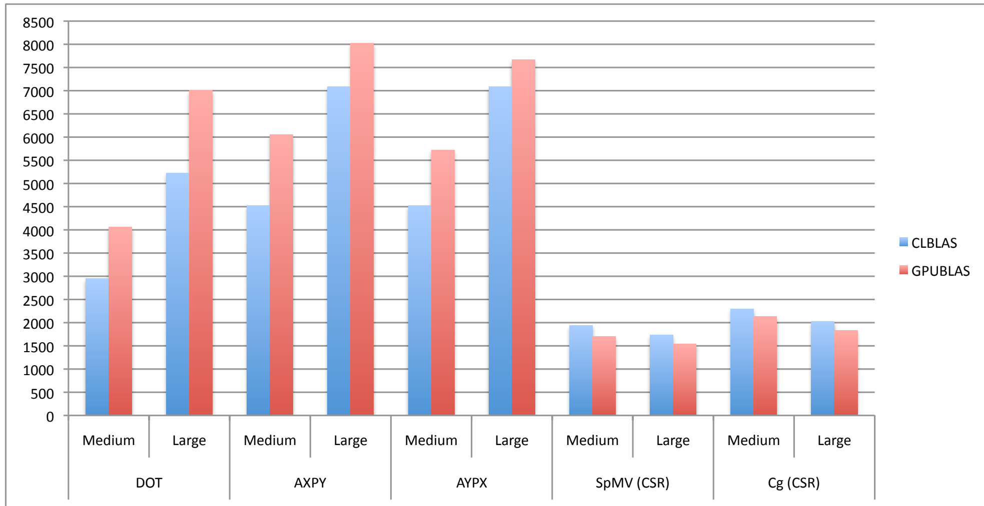

Figure 7.16: Cg CSR qdouble precision benchmark result.

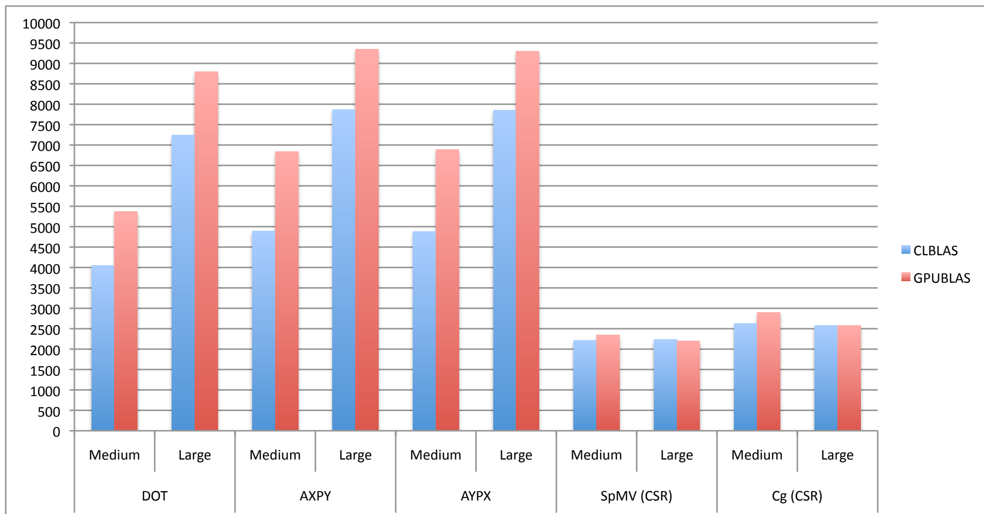

Figure 7.17: Cg CSR double precision benchmark result.

Chapter 8

Conclusions

During this masters thesis work the CUKr library has received additional support for running the Cg Krylov solver on all hardware supported by OpenCL implementations. This includes selected BLAS 1 and BLAS 2 kernels. Existing CUDA kernels in the CUKr library were ported to OpenCL, and changes were made to the CUKr source-code infrastructure to accommodate the use of OpenCL. Here some structural properties of CUKr that did not suit the use of OpenCL well had to be overcome (in essence the way CUKr dealt with references to vectors in device memory). All kernels in CUKr are solving problems that are highly memory bound. The GPU is ideal here as these architectures can deliver an order of magnitude higher memory bandwidth than the common CPU of the present. CUKr with the new additions compiles under both Linux and OS X. It should compile just fine under Windows too, though this is not tested.

OpenCL is a big leap forward. Challenges regarding memory access, especially visible in memory bound problems, are observed in the Nvidia Computing SDK 3.0. There are strong indications that the more mature CUDA technology has an overall better performance. It is believed this gap in the performance observed will close as the OpenCL implementation matures, and the SDK gets revised. It is also highly expected that other implementations of OpenCL (from AMD/ATI and Apple) will improve performance and efficiency over time.

The kernels produced in this work are expected to perform well on latest AMD/ATI GPUs (58xx - series), adjusting the partitioning (global and local sizes) is the only modification that will be needed. Unfortunately, as there is no 58xx-hardware available for the time of the benchmarking, this is not yet tested. Test runs have been done on 48xx-hardware, but performance was not what it should be considering the device's theoretical possibilities for the problems to be solved. [1]

[1] AMD/ATI has reported that this hardware was not designed with OpenCL in mind (the fact that it runs OpenCL at all is a testament to the AMD/ATI engineers forward think-

The difficulty of implementing high performance memory bound compute kernels, to run efficiently on different devices, is clear. This is directly linked to the different devices' ideal memory access pattern for utilizing its memory bandwidth — which is significantly different on a CPU and a GPU. Even within different kinds of CPUs and different types of GPUs, there are different best practices for attaining ideal memory bandwidth utilization. This problem domain underlines the importance of proper memory access patterns, ideal for each device. An ideal AYPX kernel for the GPU was run on a CPU, with performance far away from the Intel MKL library. Then, a AYPX kernel ideal for the CPU reading sequential memory addresses from each core was implemented, resulting in many fold speed improvements as long as the vectors could fit in the CPU cache. Once the vectors could not fit in the cache the performance was very similar to the Intel MKL library (on same vector lengths). At this point the performance of the GPU ideal kernel was about half. This illustrated how the GPU ideal kernel could not utilize the cache because of its memory access pattern.

Maintainable and easily readable code is difficult if not impossible to produce if one tries to make one kernel suitable for both GPU and CPU in this domain — that is to compete with other high performance implementations for a particular device (like the Intel MKL library or CUDA CUBLAS library). If one is after competitive performance, one should still make custom kernels for the architecture type. This is not only for the sake of pure performance, but also for the sake of readable and maintainable OpenCL code.

It is a fact that in High Performance Computing, one does not want to compromise on performance, in general. However, if kernels are easily portable from one architecture to another as OpenCL kernels are, this is of value in itself. Maybe even to such a degree that considerable reduction in performance is acceptable. The question is to what degree.

The author believes that OpenCL 1.0 in the domain of High Performance Computing should be used as a powerful tool for heterogeneous systems, with its orchestrating and scheduling abilities. Thereby; utilizing each device for what it is best suited for with suitable kernels for it, thus not forcing a uniform kernel on vastly different architectures — counterfeiting the high performance computing ideology.

By the end of this thesis work, the OpenCL 1.1 specification was released by The Khronos group — 18 months after the initial OpenCL 1.0 specification was released at ACM Siggraph Asia in December 2008. OpenCL 1.1 is reported to add functionality for increased programming flexibility and performance. Quoting the press release[2] from the Khronos group the

ing when the chip was in design process around 2006). It is said this card can perform with careful and highly device dependent tuning, something that would be undermining the wish to have a kernel performing well on a range of GPUs.

[2]http://www.khronos.org/news/press/releases/khronos-group-releases-opencl-1-1-

major new additions are:

- New data types including 3-component vectors and additional image formats
- Handling commands from multiple host threads and processing buffers across multiple devices
- Operations on regions of a buffer including read, write and copy of 1D, 2D or 3D rectangular regions
- Enhanced use of events to drive and control command execution
- Additional OpenCL C built-in functions such as integer clamp, shuffle and asynchronous strided copies
- Improved OpenGL interoperability through efficient sharing of images and buffers by linking OpenCL and OpenGL events

We underline that OpenCL 1.0 capable devices can still benefit from improved implementations, as the OpenCL 1.0 compatible implementations are maturing — as seen in parts of this work.

---

parallel-computing-standard

Chapter 9

Further work

At the end of this project it is clear that there are many areas part of this project or closely related that are worth exploring in more depth:

- Implementing the use of CL Image memory objects in the OpenCL based SpMV kernels, in order to achieve higher performance.
- Investigate how automatic set-up of Local and Global sub-divisions (partitioning) impacts performance. Also, experimentation with explicit and implicit sub-divisions.
- As this project has required competing with the CUDA kernels already part of CUKr (both from a practical and interest point of view), substantial focus has not been given to writing one given BLAS 1 kernel that is suitable(in terms of acceptable speed) for both GPU and CPU. This was not a priority due to the High Performance Computing requirement — as speed on the GPU would certainly be sacrificed with such a focus. Further experiments with a kernel more suiting of both architecture types had been of interest. Here, adoptable memory access must be investigated further (un-cluttered and maintainable code that has acceptable memory access pattern on both architecture types).
- Auto-tuning for hardware.
- The running on multiple compute devices(1), and running on entire cluster with multiple compute devices in each node with help of MPI(2), as CUKr can.
- Implementing BCSR and BELL SpMV formats; new performance testings.
- More experimentation with running on CPUs, and optimizing kernels for good utilization of the CPU cache-hierarchy. Especially on a

Non-Uniform Memory Access (NUMA) Nehalem cluster as the Okuda Laboratory has access to, where every node is connected with Infiniband network. In general, testing the combination of OpenCL with NUMA machines (CPU), build up experience and best practices in this area.

- Running on latest generation AMD/ATI hardware, such as the 58xx-series of cards. Running on Nvidia Fermi architecture based cards.
- Re-implement a CUKr-like library from scratch using only OpenCL, optionally taking other points mentioned here into account. The use of C++ to reduce complexity in the code to make it more readable. If done, the already existing (open-source) OpenCL C++ bindings will have to be used.

Bibliography

[1] The Cilk Project. *http://supertech.csail.mit.edu/cilk/*, *Accessed online March*, 2010.

[2] NESL - a nested data parallel language. *http://www.cs.cmu.edu/~scandal/nesl.html*, *Accessed online March*, 2010.

[3] OpenMP Application Program Interface Version 3.0. *http://www.openmp.org/mp-documents/spec30.pdf*, *Accessed online March*, 2010.

[4] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pages 115–126, New York, NY, USA, 2010. ACM.

[5] Olav Aanes Fagerlund. An investigation of the emerging Open Computing Language (OpenCL) – *and a comparison of OpenCL and CUDA implementations*. *NTNU, Computer Science Project Report*, December 2008.

[6] Serban Georgescu. CUKr User's Guide v. 1.0.0. 2009.

[7] Serban Georgescu. *Krylov Solvers Accelerated by Manycore Processors*. PhD thesis, The University of Tokyo, Japan, 2009.

[8] Magnus R. Hestenes and Eduard Steifel. Methods of conjugate gradients for solving linear systems. In *Journal of Research of the National Bureau of Standards*, pages 49: 409–436, Dec 1952.

[9] Apple Inc. WWDC'08 Open Compute Library Specification Draft Revision 1.0.06. *Obtained at WWDC'08*, 2008.

[10] B. Jacob. A case for studying dram issues at the system level. In *IEEE M MICRO*, page 23(4):44Ð56, 2003.

[11] Rune Erlend Jensen. Techniques and Tools for Optimizing Codes on Modern Architectures: A Low-Level Approach. *NTNU, Computer Science Master Thesis*, May 2009.

[12] The Khronos Group. Open Compute Library Specification Version 1.0, Document Revision 48. http://www.khronos.org/registry/cl/, 2009.

[13] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* http://llvm.cs.uiuc.edu.

[14] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[15] Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro, pages 39-55, Vol. 28, Issue 2*, 2008.

[16] Advanced Micro Devices. R700-Family Instruction Set Architecture. http://developer.amd.com/gpu_assets/R700-Family_Instruction_Set_Architecture.pdf, 2009.

[17] Advanced Micro Devices. Ati Stream Computing OpenCL Programming Guide. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf, 2010.

[18] Nvidia. OpenCL Programming Guide for the CUDA Architecture. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf, 2010.

[19] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. In *SIAM Journal on Scientific and Statistical Computing*, pages 7: 856–869, 1986.

[20] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[21] Richard Wilson Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, 2003.

Appendix A

Hardware specifications

In Table A.1 the bandwidth is computed by the product of memory channels, the memory clock speed towards the CPU, and the number of Bytes transferred at each clock. The i7 has *three* memory channels, and the Core 2 Quad has *two*. The peak performance is the product of amount of cores in the CPU, how many single precision floats fits in the SSE registers, and the clock speed of the CPU. For double precision the performance is half, as half as many values fit in the SSE registers (4 values, each 64 bit). Note the bandwidth advantage seen in the i7 based chips. The Core 2 Quad has a bandwidth of 1/3rd relative to the i7. All though they both utilize DDR3 memory, the Core 2 Quad is limited by the Front Side Bus (FSB), as all memory transfers goes through the North-bridge chip — severely limiting the memory bandwidth, by diminishing the effect of *two* memory channels (resulting in *one* transfer per clock instead of *two*, the latter being allowed by dual channel memory). The i7 has lower memory latency, due the advantage of directly communicating with the memory as the memory controller is part of the CPU chip itself. Table A.1 shows this difference in practice as a memory stream test was run on each of the architectures. A memory-bandwidth open-source benchmark program called *Stream* was used for this, and can be found at the *http://www.cs.virginia.edu/stream/* site.

	Nehalem i7 975	**Core 2 Quad Q9450**
Peak perf.	4 x 8 x 3.33GHz = 106.56GFlop/s	4 x 8 x 2.66GHz = 85.12GFlop/s
Bandwidth	3 x 1333MHz x 8 B = 32GB/s	1333MHz x 8 B = 10.66GB/s
Stream res.	28.5GB/s	9.4GB/s

Table A.1: Intel CPU characteristics

Property	**Value**
Fabrication process	55nm
Transistors	956 Million
Core Clock	750MHz
Stream Processors	800
Memory Clock	900MHz GDDR5 → 3600MHz data rate
Memory Bus Width	256 bit
Memory Bandwith	115.2 GB/s
Single precision peak performance	1.2 TeraFlop/s
Double precision peak performance	240 GigaFlop/s
Maximum board power	160 Watts

Table A.2: ATI Radeon HD 4870 characteristics

Property	**Value**
Fabrication process	40nm
Transistors	2.15 billion
Core Clock	850MHz
Stream Processors	1600
Memory Clock	1200MHz GDDR5 → 4800MHz data rate
Memory Bus Width	256 bit
Memory Bandwith	153.6 GB/s
Single precision peak performance	2.72 TeraFlop/s
Double precision peak performance	544 GigaFlop/s
Maximum board power	188 Watts

Table A.3: ATI Radeon HD 5870 characteristics

Property	Value
Fabrication process	65nm
Transistors	1.4 billion
Shader Clock	1296MHz
CUDA cores	240
Memory Clock	1107MHz GDDR3 $\longrightarrow$ 2214MHz data rate
Memory Bus Width	512 bit
Memory Bandwith	141.7 GB/s
Single precision peak performance	933 GigaFlop/s
Double precision peak performance	78 GigaFlop/s
Maximum board power	236 Watts

Table A.4: Nvidia GTX 280 characteristics

Property	Value
Fabrication process	40nm
Transistors	3 billion
Shader Clock	1401MHz
CUDA cores	480
Memory Clock	1848MHz GDDR5 $\longrightarrow$ 3696MHz data rate
Memory Bus Width	384 bit
Memory Bandwith	177.4 GB/s
Single precision peak performance	1344.96 GigaFlop/s
Double precision peak performance	168.12 GigaFlop/s
Maximum board power	250 Watts

Table A.5: Nvidia GTX 480 characteristics

Appendix B

OpenCL devices under different implementations

This Appendix part shows printouts of OpenCL characteristics, on different hardware and with different implementations. Notice especially how the ATI SDK v.2.1 shows 0 KB cache (global memory cache in OpenCL terms) of the Intel Nehalem based Xeon processor. The ATI SDK v.2.0.1 installed on a Intel Core 2 Quad system shows only 64KB of the cache(probably only the L1 cache). The reason for this is currently not known. Also notice that the Apple implementation only allows 1 work-item per work-group on the CPU. In contrast the ATI one allows 1024.

B.1 Apple Mac Pro, OS X 10.6.4

```
* * * 2 OpenCL devices found in the system * * *

Device number 0 :
-----------------------------------------
CL platform vendor: Apple
CL platform version: OpenCL 1.0 (Apr  7 2010 19:04:28)
CL device name: Radeon HD 4870
Max compute units: 10
Clock frequency: 750 MHz
Device global memory size: 512 MB
Device global memory cache size: 0 KB
Device global memory cache line size: 0 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_LOCAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 1024 threads
Device profiling timer resolution: 40 nanoseconds
Device preferred vector width int: 4
```

```
Device preferred vector width float: 4
Device preferred vector width double: 0
Device image support (1: true, 0 false): 0
Extensions supported :
  cl_APPLE_gl_sharing

Device number 1 :
-----------------------------------------
CL platform vendor: Apple
CL platform version: OpenCL 1.0 (Apr  7 2010 19:04:28)
CL device name: Intel(R) Xeon(R) CPU           E5462  @
   2.80GHz
Max compute units: 8
Clock frequency: 2800 MHz
Device global memory size: 3840 MB
Device global memory cache size: 6144 KB
Device global memory cache line size: 64 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_GLOBAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 1 threads
Device profiling timer resolution: 1 nanoseconds
Device preferred vector width int: 4
Device preferred vector width float: 4
Device preferred vector width double: 2
Device image support (1: true, 0 false): 1
Extensions supported :
  cl_khr_fp64
  cl_khr_global_int32_base_atomics
  cl_khr_global_int32_extended_atomics
  cl_khr_local_int32_base_atomics
  cl_khr_local_int32_extended_atomics
  cl_khr_byte_addressable_store
  cl_APPLE_gl_sharing
  cl_APPLE_SetMemObjectDestructor
  cl_APPLE_ContextLoggingFunctions
```

B.2 Apple Mac Pro, OS X 10.6.3

```
* * * 2 OpenCL devices found in the system * * *

Device number 0 :
-----------------------------------------
CL platform vendor: Apple
CL platform version: OpenCL 1.0 (Feb 10 2010 23:46:58)
CL device name: Radeon HD 4870
Max compute units: 10
Clock frequency: 750 MHz
```

```
Device global memory size: 512 MB
Device global memory cache size: 0 KB
Device global memory cache line size: 0 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_LOCAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 1024 threads
Device profiling timer resolution: 40 nanoseconds
Device preferred vector width int: 4
Device preferred vector width float: 4
Device preferred vector width double: 0
Device image support (1: true, 0 false): 0
Extensions supported :
  cl_APPLE_gl_sharing

Device number 1 :
----------------------------------------
CL platform vendor: Apple
CL platform version: OpenCL 1.0 (Feb 10 2010 23:46:58)
CL device name: Intel(R) Xeon(R) CPU           E5462  @
   2.80GHz
Max compute units: 8
Clock frequency: 2800 MHz
Device global memory size: 3840 MB
Device global memory cache size: 6144 KB
Device global memory cache line size: 64 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_GLOBAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 1 threads
Device profiling timer resolution: 1 nanoseconds
Device preferred vector width int: 4
Device preferred vector width float: 4
Device preferred vector width double: 2
Device image support (1: true, 0 false): 1
Extensions supported :
  cl_khr_fp64
  cl_khr_global_int32_base_atomics
  cl_khr_global_int32_extended_atomics
  cl_khr_local_int32_base_atomics
  cl_khr_local_int32_extended_atomics
  cl_khr_byte_addressable_store
  cl_APPLE_gl_sharing
  cl_APPLE_SetMemObjectDestructor
  cl_APPLE_ContextLoggingFunctions
```

B.3 Apple Macbook Pro, OS X 10.6.4

```
* * * 2 OpenCL devices found in the system * * *

Device number 0 :
-----------------------------------------
CL platform vendor: Apple
CL platform version: OpenCL 1.0 (Apr  7 2010 19:04:28)
CL device name: GeForce 8600M GT
Max compute units: 4
Clock frequency: 940 MHz
Device global memory size: 512 MB
Device global memory cache size: 0 KB
Device global memory cache line size: 0 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_LOCAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 512 threads
Device profiling timer resolution: 1000 nanoseconds
Device preferred vector width int: 1
Device preferred vector width float: 1
Device preferred vector width double: 0
Device image support (1: true, 0 false): 1
Extensions supported :
  cl_khr_byte_addressable_store
  cl_khr_global_int32_base_atomics
  cl_khr_global_int32_extended_atomics
  cl_APPLE_gl_sharing
  cl_APPLE_SetMemObjectDestructor
  cl_APPLE_ContextLoggingFunctions


Device number 1 :
-----------------------------------------
CL platform vendor: Apple
CL platform version: OpenCL 1.0 (Apr  7 2010 19:04:28)
CL device name: Intel(R) Core(TM)2 Duo CPU     T9300  @
   2.50GHz
Max compute units: 2
Clock frequency: 2500 MHz
Device global memory size: 3072 MB
Device global memory cache size: 6144 KB
Device global memory cache line size: 64 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_GLOBAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 1 threads
```

```
Device profiling timer resolution: 1 nanoseconds
Device preferred vector width int: 4
Device preferred vector width float: 4
Device preferred vector width double: 2
Device image support (1: true, 0 false): 1
Extensions supported :
  cl_khr_fp64
  cl_khr_global_int32_base_atomics
  cl_khr_global_int32_extended_atomics
  cl_khr_local_int32_base_atomics
  cl_khr_local_int32_extended_atomics
  cl_khr_byte_addressable_store
  cl_APPLE_gl_sharing
  cl_APPLE_SetMemObjectDestructor
  cl_APPLE_ContextLoggingFunctions
```

B.4 Apple Macbook Pro, OS X 10.6.3

```
* * * 2 OpenCL devices found in the system * * *

Device number 0 :
-----------------------------------------
CL platform vendor: Apple
CL platform version: OpenCL 1.0 (Feb 10 2010 23:46:58)
CL device name: GeForce 8600M GT
Max compute units: 4
Clock frequency: 940 MHz
Device global memory size: 512 MB
Device global memory cache size: 0 KB
Device global memory cache line size: 0 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_LOCAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 512 threads
Device profiling timer resolution: 1000 nanoseconds
Device preferred vector width int: 1
Device preferred vector width float: 1
Device preferred vector width double: 0
Device image support (1: true, 0 false): 1
Extensions supported :
  cl_khr_byte_addressable_store
  cl_khr_global_int32_base_atomics
  cl_khr_global_int32_extended_atomics
  cl_APPLE_gl_sharing
  cl_APPLE_SetMemObjectDestructor
  cl_APPLE_ContextLoggingFunctions
```

```
Device number 1 :
-----------------------------------------
CL platform vendor: Apple
CL platform version: OpenCL 1.0 (Feb 10 2010 23:46:58)
CL device name: Intel(R) Core(TM)2 Duo CPU     T9300  @
   2.50GHz
Max compute units: 2
Clock frequency: 2500 MHz
Device global memory size: 3072 MB
Device global memory cache size: 6144 KB
Device global memory cache line size: 64 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_GLOBAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 1 threads
Device profiling timer resolution: 1 nanoseconds
Device preferred vector width int: 4
Device preferred vector width float: 4
Device preferred vector width double: 2
Device image support (1: true, 0 false): 1
Extensions supported :
  cl_khr_fp64
  cl_khr_global_int32_base_atomics
  cl_khr_global_int32_extended_atomics
  cl_khr_local_int32_base_atomics
  cl_khr_local_int32_extended_atomics
  cl_khr_byte_addressable_store
  cl_APPLE_gl_sharing
  cl_APPLE_SetMemObjectDestructor
  cl_APPLE_ContextLoggingFunctions
```

B.5 Nvidia CUDA SDK 3.0 Linux

```
* * * 3 OpenCL devices found in the system * * *

Device number 0 :
-----------------------------------------
CL platform vendor:
CL platform version:
CL device name:
Max compute units: 30
Clock frequency: 1300 MHz
Device global memory size: 1023 MB
Device global memory cache size: 0 KB
Device global memory cache line size: 0 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_LOCAL
Device max constant buffer size: 64 KB
```

```
Device max work-item dimensions: 3
Device max work-group size: 512 threads
Device profiling timer resolution: 1000 nanoseconds
Device preferred vector width int: 1
Device preferred vector width float: 1
Device preferred vector width double: 1
Device image support (1: true, 0 false): 1
Extensions supported :


Device number 1 :
-----------------------------------------
CL platform vendor:
CL platform version:
CL device name:
Max compute units: 30
Clock frequency: 1300 MHz
Device global memory size: 1023 MB
Device global memory cache size: 0 KB
Device global memory cache line size: 0 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_LOCAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 512 threads
Device profiling timer resolution: 1000 nanoseconds
Device preferred vector width int: 1
Device preferred vector width float: 1
Device preferred vector width double: 1
Device image support (1: true, 0 false): 1
Extensions supported :


Device number 2 :
-----------------------------------------
CL platform vendor:
CL platform version:
CL device name:
Max compute units: 30
Clock frequency: 1300 MHz
Device global memory size: 1023 MB
Device global memory cache size: 0 KB
Device global memory cache line size: 0 Bytes
Device local memory size: 16 KB
Device local memory is physical memory type: CL_LOCAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 512 threads
Device profiling timer resolution: 1000 nanoseconds
```

```
Device preferred vector width int: 1
Device preferred vector width float: 1
Device preferred vector width double: 1
Device image support (1: true, 0 false): 1
Extensions supported :
```

B.6 ATI Stream SDK 2.1 Linux

```
* * * 1 OpenCL devices found in the system * * *

Device number 0 :
-----------------------------------------
CL platform vendor: Advanced Micro Devices, Inc.
CL platform version: OpenCL 1.0 ATI-Stream-v2.1 (145)
CL device name: Intel(R) Xeon(R) CPU           X5550  @
   2.67GHz
Max compute units: 8
Clock frequency: 2666 MHz
Device global memory size: 3072 MB
Device global memory cache size: 0 KB
Device global memory cache line size: 0 Bytes
Device local memory size: 32 KB
Device local memory is physical memory type: CL_GLOBAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 1024 threads
Device profiling timer resolution: 1 nanoseconds
Device preferred vector width int: 4
Device preferred vector width float: 4
Device preferred vector width double: 0
Device image support (1: true, 0 false): 0
Extensions supported :
  cl_khr_icd
  cl_amd_fp64
  cl_khr_global_int32_base_atomics
  cl_khr_global_int32_extended_atomics
  cl_khr_local_int32_base_atomics
  cl_khr_local_int32_extended_atomics
  cl_khr_int64_base_atomics
  cl_khr_int64_extended_atomics
  cl_khr_byte_addressable_store
  cl_khr_gl_sharing
  cl_ext_device_fission
  cl_amd_device_attribute_query
  cl_amd_printf
```

B.7 ATI Stream SDK 2.01 Linux

```
* * * 1 OpenCL devices found in the system * * *
```

```
Device number 0 :
-----------------------------------------
CL platform vendor: Advanced Micro Devices, Inc.
CL platform version: OpenCL 1.0 ATI-Stream-v2.0.1
CL device name: Intel(R) Core(TM)2 Quad  CPU   Q9450  @
   2.66GHz
Max compute units: 4
Clock frequency: 2667 MHz
Device global memory size: 3072 MB
Device global memory cache size: 64 KB
Device global memory cache line size: 64 Bytes
Device local memory size: 32 KB
Device local memory is physical memory type: CL_GLOBAL
Device max constant buffer size: 64 KB
Device max work-item dimensions: 3
Device max work-group size: 1024 threads
Device profiling timer resolution: 1 nanoseconds
Device preferred vector width int: 4
Device preferred vector width float: 4
Device preferred vector width double: 0
Device image support (1: true, 0 false): 0
Extensions supported :
  cl_khr_icd
  cl_khr_global_int32_base_atomics
  cl_khr_global_int32_extended_atomics
  cl_khr_local_int32_base_atomics
  cl_khr_local_int32_extended_atomics
  cl_khr_int64_base_atomics
  cl_khr_int64_extended_atomics
  cl_khr_byte_addressable_store
```

Appendix C

Matrix properties

Matrix	Rows	Cols	Nz	Rank	Full Rank	Structure	SPD	Type	Kind
apache1	80800	80800	542184	80800	yes	symmetric	yes	real	structural problem
cfd1	70656	70656	1825580	70656	yes	symmetric	yes	real	CFD problem
nasasrb	54870	54870	2677324	54870	yes	symmetric	yes	real	structural problem
cfd2	123440	123440	3085406	123440	yes	symmetric	yes	real	CFD problem
apache2	715176	715176	4817870	715176	yes	symmetric	yes	real	structural problem
crankseg_2	63838	63838	14148858	63838	yes	symmetric	yes	real	structural problem
af_0_k101	503625	503625	17550675	503625	yes	symmetric	yes	real	structural problem
af_1_k101	503625	503625	17550675	503625	yes	symmetric	yes	real	structural problem
af_2_k101	503625	503625	17550675	503625	yes	symmetric	yes	real	structural problem
af_3_k101	503625	503625	17550675	503625	yes	symmetric	yes	real	structural problem
af_4_k101	503625	503625	17550675	503625	yes	symmetric	yes	real	structural problem
af_5_k101	503625	503625	17550675	503625	yes	symmetric	yes	real	structural problem
af_shell3	504855	504855	17562051	504855	yes	symmetric	yes	real	structural problem
af_shell4	504855	504855	17562051	504855	yes	symmetric	yes	real	structural problem
af_shell7	504855	504855	17579155	504855	yes	symmetric	yes	real	structural problem
af_shell8	504855	504855	17579155	504855	yes	symmetric	yes	real	structural problem

Table C.1: Matrix properties table. The divisions shows the 3 groups used. From top to bottom; small – medium – large, respectively. The last four matrices are from subsequent structural problems. CFD is short for Computational Fluid Dynamics. All matrices are **2D/3D**.

Appendix D

Benchmark graphs

For the interested reader these graphs are supplied in addition to those commented on under the results chapter.

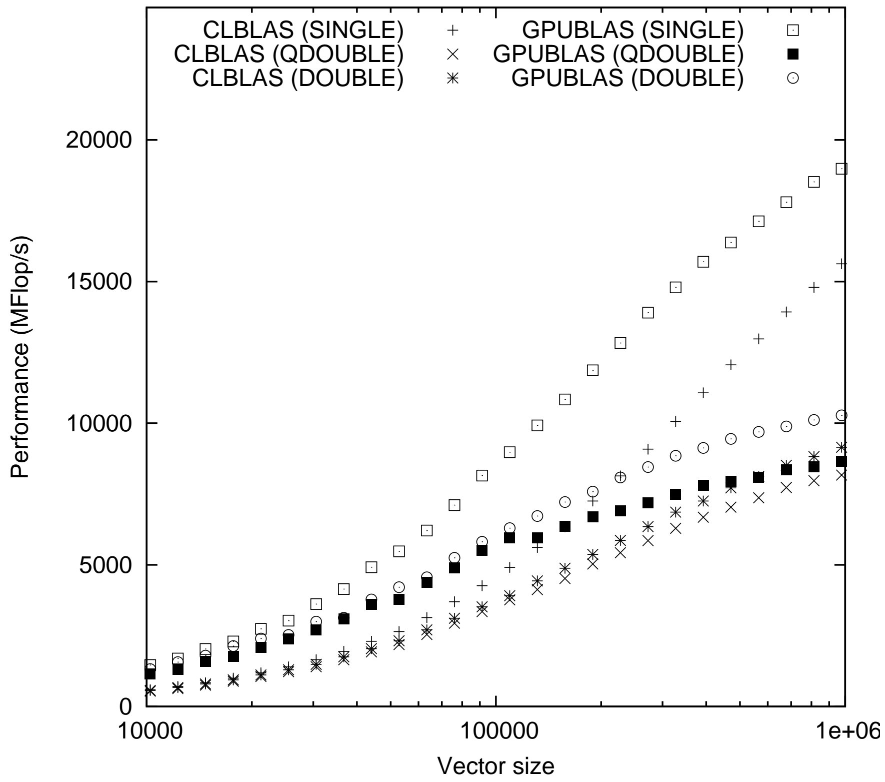

Figure D.1: AXPY, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does.

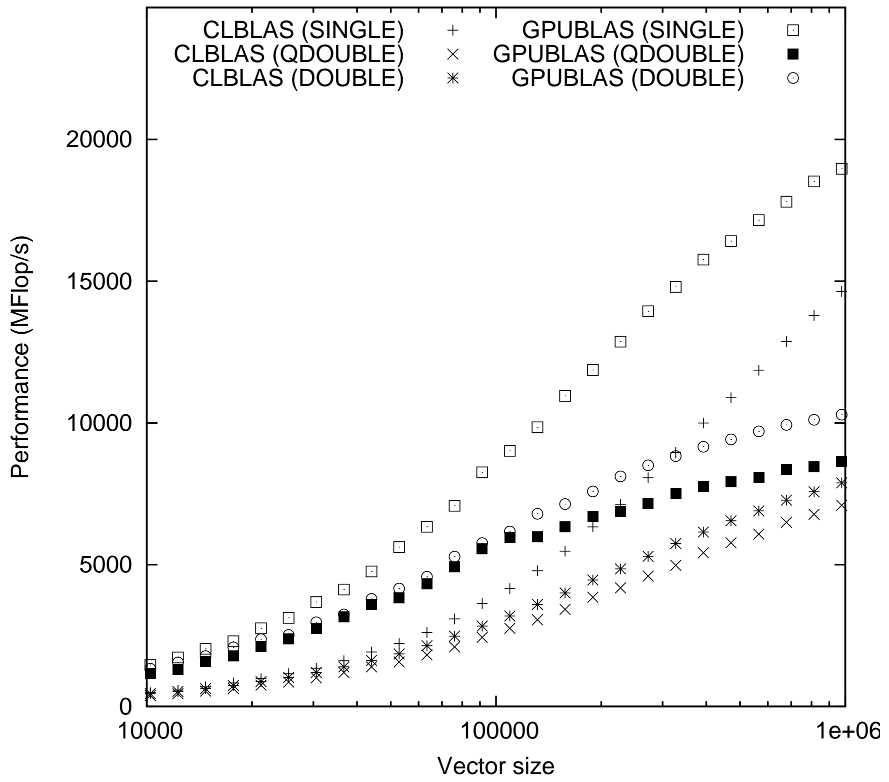

Figure D.2: AXPY, OpenCL kernels uses local memory, as the CUDA kernel also does.

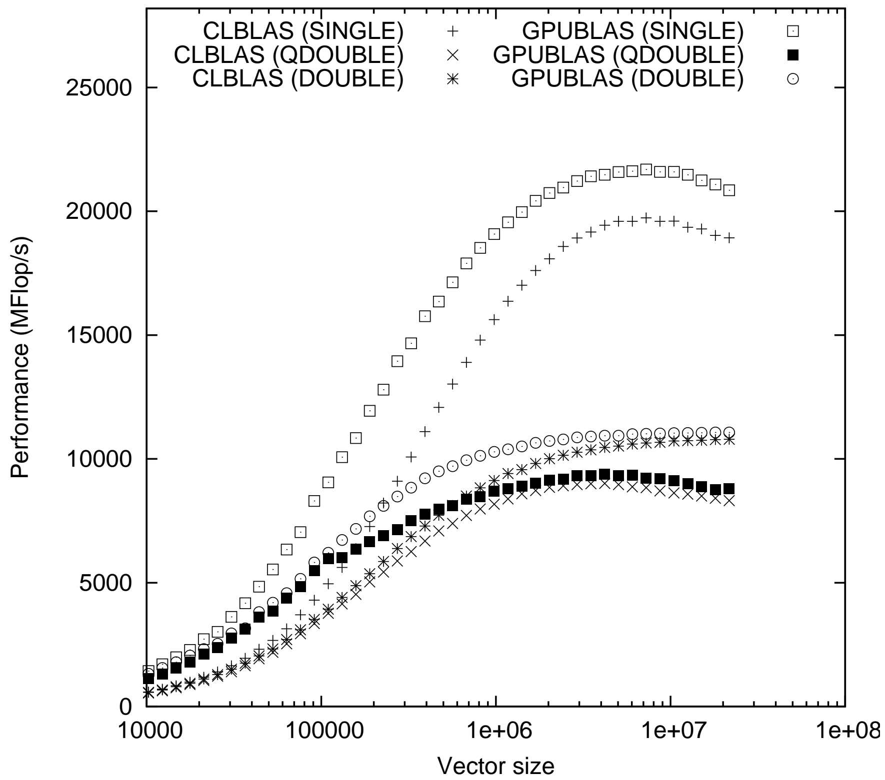

Figure D.3: AXPY with large vector sizes — up to 21 million elements, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does.

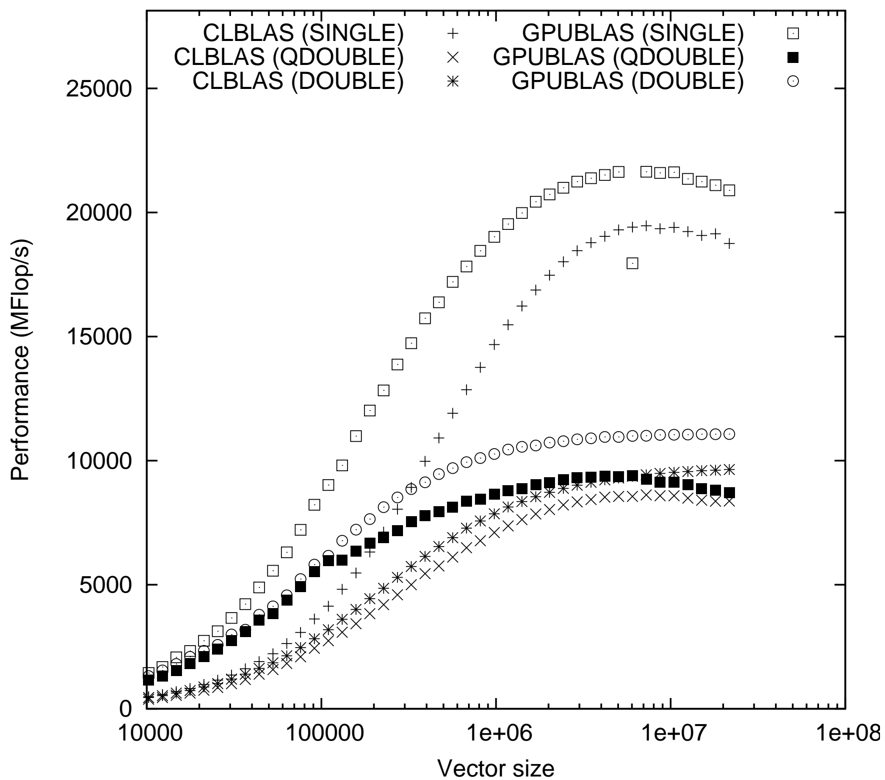

Figure D.4: AXPY with large vector sizes — up to 21 million elements, OpenCL kernels uses local memory, as the CUDA kernel also does.

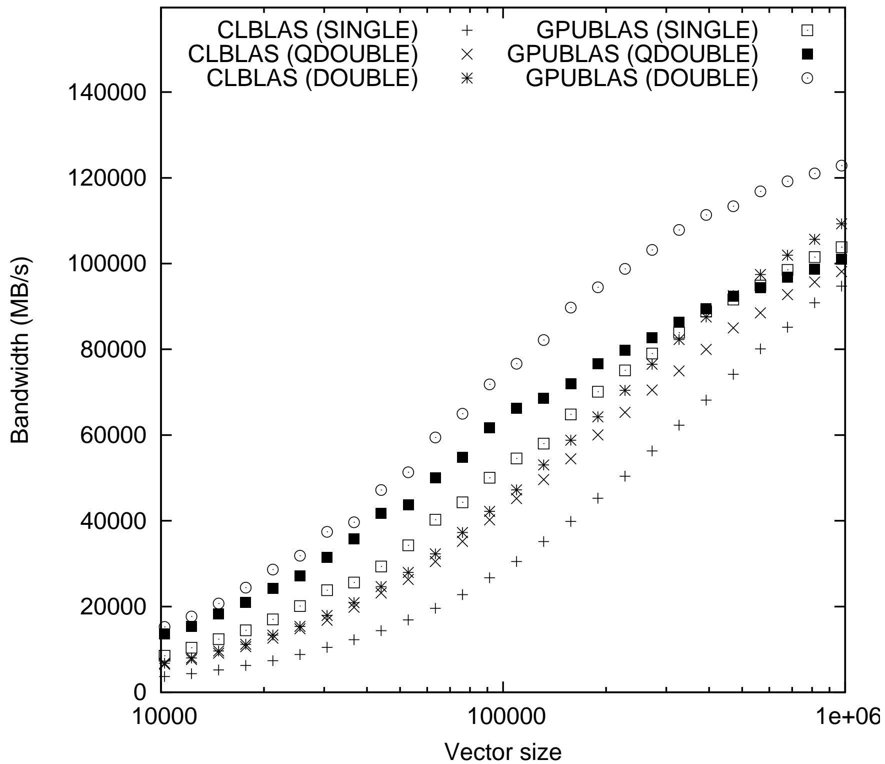

Figure D.5: AYPX, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. Partitioning sizes are also adjusted to suit. Bandwidth utilization is illustrated.

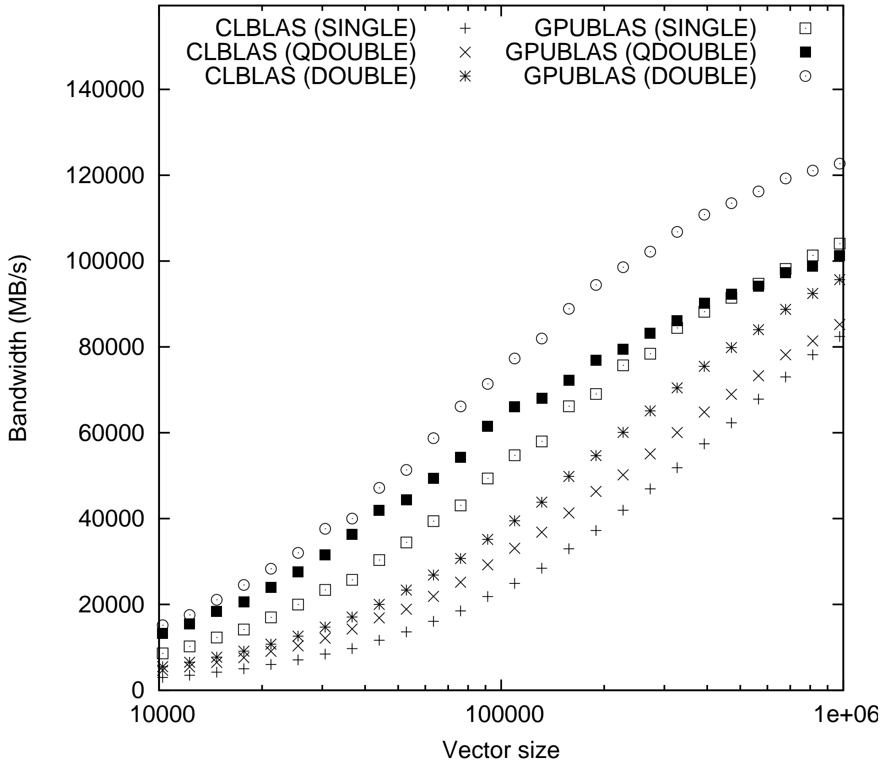

Figure D.6: AYPX, OpenCL kernels uses local memory, as the CUDA kernel also does. Similar partitioning sizes as to the CUDA kernels are used. Bandwidth utilization is illustrated.

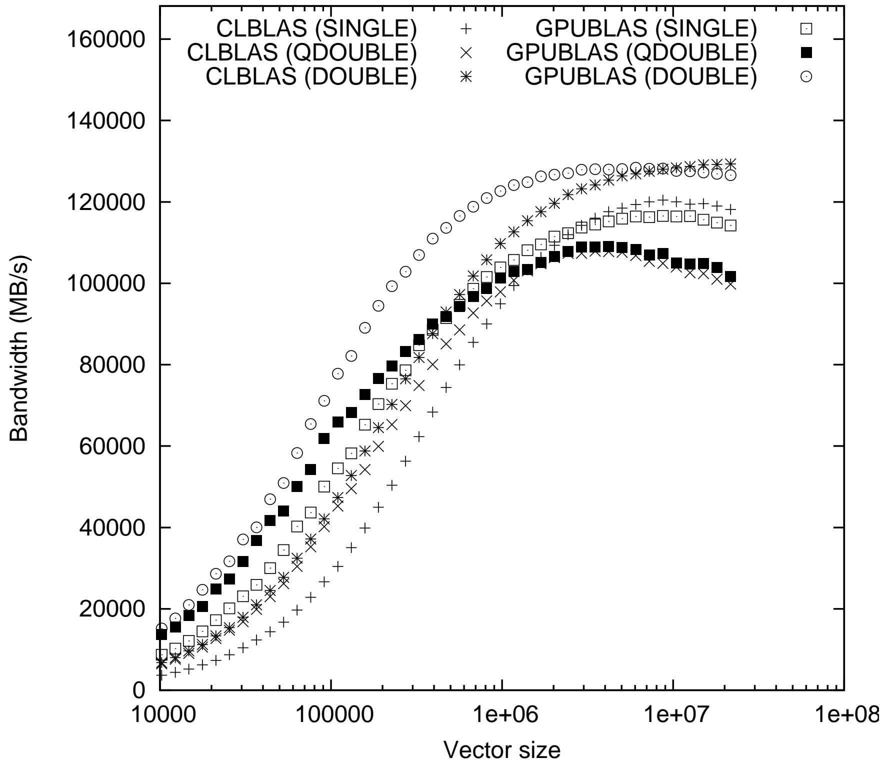

Figure D.7: AYPX with large vector sizes — up to 21 million elements, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. Partitioning sizes are also adjusted to suit. Bandwidth utilization is illustrated.

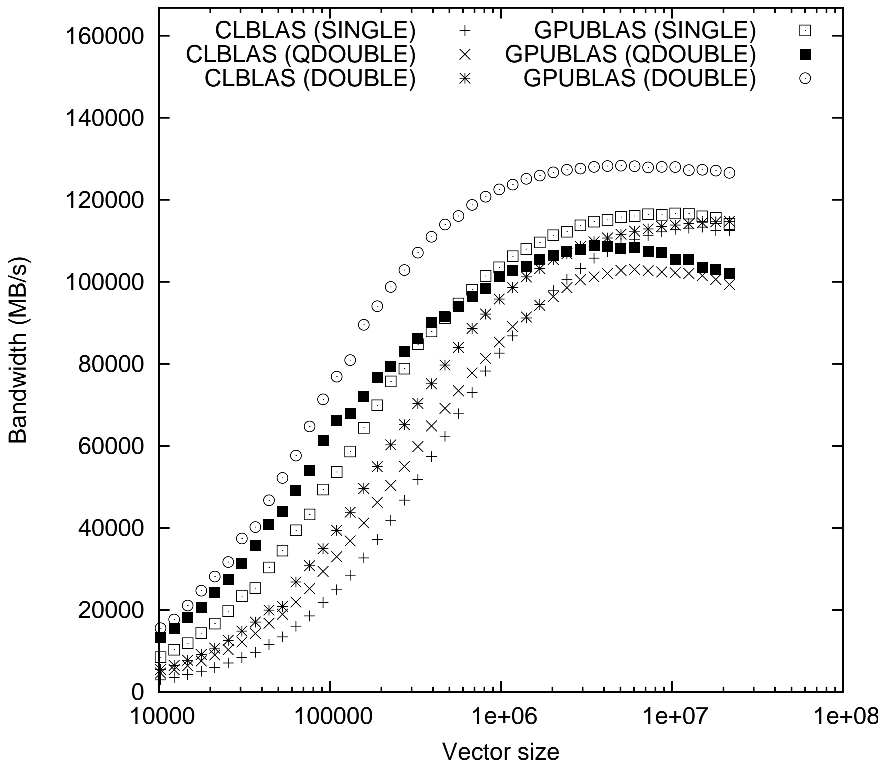

Figure D.8: AYPX with large vector sizes — up to 21 million elements, OpenCL kernels uses local memory, as the CUDA kernel also does. Similar partitioning sizes as to the CUDA kernels are used. Bandwidth utilization is illustrated.

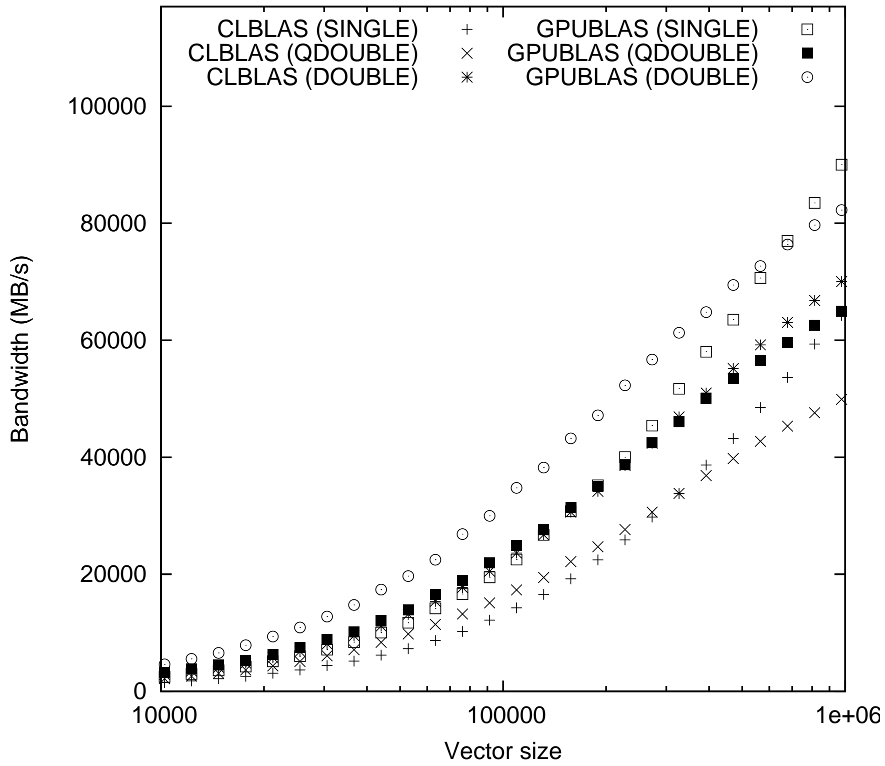

Figure D.9: DOT; OpenCL vs. CUDA implementation. Bandwidth utilization is illustrated.

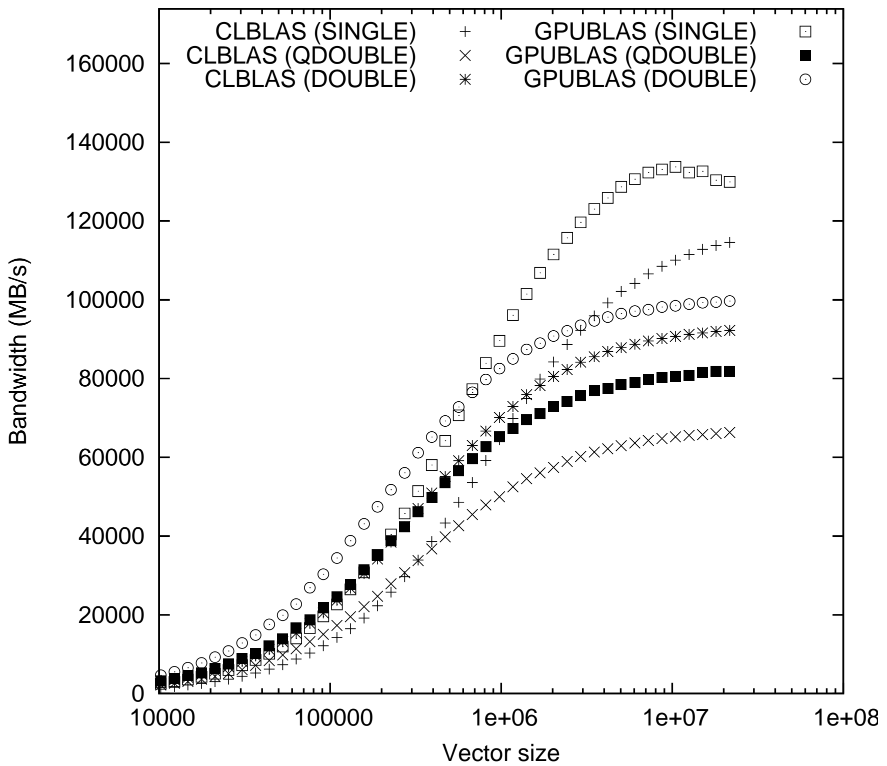

Figure D.10: DOT with large vector sizes — up to 21 million elements; OpenCL vs. CUDA implementation. Bandwidth utilization is illustrated.

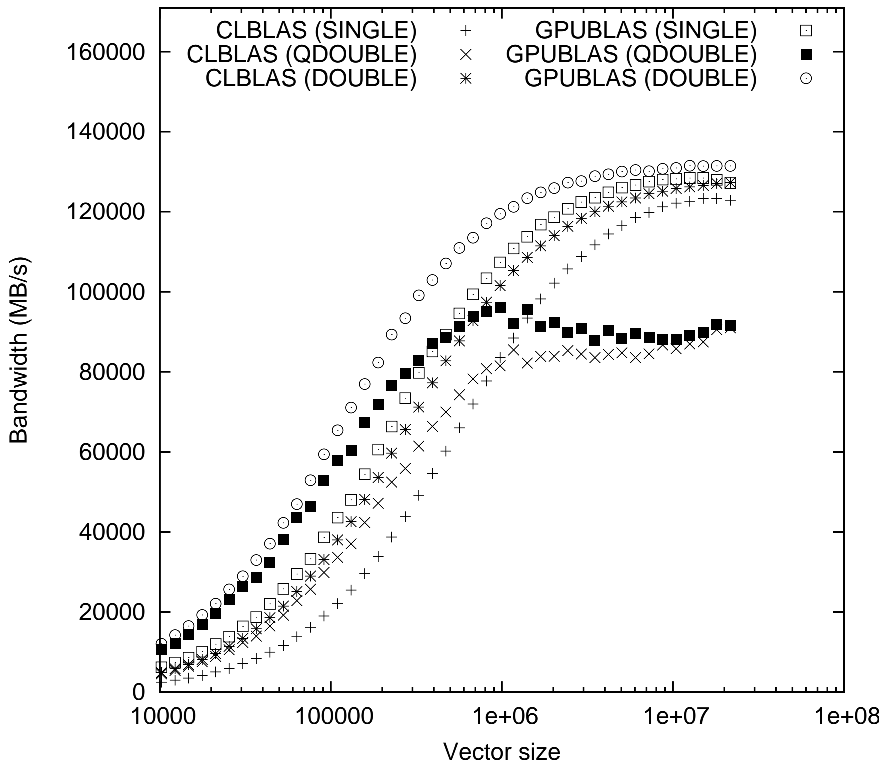

Figure D.11: SCAL with large vector sizes — up to 21 million elements, OpenCL kernels uses no local memory as opposed to the CUDA kernel which does. Bandwidth utilization is illustrated.

Appendix E

Code listings

Note that not all code written for this project is part of the code listings. Because of the large number of B5 format pages required, code-files are left out. The listings here contains the kernels, their set-up functions, the OpenCL CUKr initialization, and at the end the code for probing a system for OpenCL devices and printing out these characteristics. Other important additions such as the BLAS level infrastructure changes, and (especially important) the vector operations source-file is left out. These source-files both contain OpenCL and CUDA relevant code for compilation (where *one* of these technologies is chosen at cmake configuration time before compilation). This makes the files large; it was therefore decided to leave them out of the listings here. However, all the relevant source-code is found in the included attachment. Note that the BLAS 1 kernel code here uses local memory, as this was the last testing. The author expects this to be a benefit with future OpenCL implementations to be released.

- `src/init/`:

  Contains the OpenCL initialization code, executed at CUKr program start.

- `src/blas/blas.c`:

  Contains BLAS bindings for CUKr. Higher level abstract BLAS functions. Part of the BLAS level infrastructure.

- `src/blas/sp/blas1_sp.c` and `src/blas/sp/blas2_sp.c`:

  Interfaces for the BLAS routines, including the new OpenCL ones.

- `src/blas/impl/gpu/`:

  Contains a directory with all the OpenCL kernels and their set-up code. And `gpu_blas.c`; wrapper for the BLAS functions, wether OpenCL based or CUDA based.

- `src/mat_vec/sp/vector_sp.c`:

  Contains the necessary vector-operations for OpenCL, such as allocating the OpenCL vectors, freeing them, copying them between host and device and so on.

E.1 AXPY CPU Single

```
/*
 *  kernel_saxpy.cl
 *
 *
 *
 */

__kernel void kernel_saxpy(
    const int n,
    const float a,
    __global const float4* x,
    __global float4* y
)
{
  int work_item_domain;
  int totalWorkItems = get_global_size(0);

  // get amount of overshooting float values (0, 1, 2,
     or 3)
  int restSingle = n % 4;

  // amount of full float4 values
  int amountQuad = (n - restSingle) / 4;

  // how many full float4 values among the global work-
     items, assumes amountQuad >> totalWorkItems
  int restQuad = amountQuad % totalWorkItems;
  if (restQuad == 0)
    work_item_domain = amountQuad / totalWorkItems;
  else{
    work_item_domain = (amountQuad - restQuad) /
       totalWorkItems;
  }

  // find where to start at each work-item
  int ctaStart = get_global_id(0) * work_item_domain;
  // define the end
  int ctaEnd = ctaStart + work_item_domain;

  // Read the data, insert a #PRAGMA unroll if available
```

```
  for (int i = ctaStart; i < ctaEnd; i++)
  {
    y[i] = y[i] + a * x[i];
  }

  int lastQuad = ctaEnd + restQuad;

  // handle last overshooting float4 values (max
      totalWorkItems - 1)
  // this part should be improved to a more even
      distribution among all work-items(?)
  if (restQuad != 0 && (get_global_id(0) + 1 ==
      totalWorkItems)){
    for (int i = ctaEnd; i < lastQuad; i++)
    {
      y[i] = y[i] + a * x[i];
    }
  }

  // handle the last overshooting float values (max 3)
  if (restSingle != 0 && (get_global_id(0) + 1 ==
      totalWorkItems)){
    if (restSingle == 1)
      y[lastQuad].x = y[lastQuad].x + a * x[lastQuad].x;
    else if (restSingle == 2){
      y[lastQuad].x = y[lastQuad].x + a * x[lastQuad].x;
      y[lastQuad].y = y[lastQuad].y + a * x[lastQuad].y;
    }
    else if (restSingle == 3){
      y[lastQuad].x = y[lastQuad].x + a * x[lastQuad].x;
      y[lastQuad].y = y[lastQuad].y + a * x[lastQuad].y;
      y[lastQuad].z = y[lastQuad].z + a * x[lastQuad].z;
    }
  }

}
```

E.2 AXPY GPU Single

```
/*
 *  kernel_saxpy.cl
 *
 *
 *
 */

__kernel void kernel_saxpy(
    const int n,
    const float a,
```

```
    __global const float* x,
    __global float* y,
    __local float* buffer_x,
    __local float* buffer_y
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data (full lines) */
  for (int i = ctaStart + tx; i < n; i += totalThreads)
  {
    /* Prefetch x and y */
    buffer_x[tx] = x[i];
    buffer_y[tx] = y[i];

    /* Multiply and write */
    y[i] = buffer_y[tx] + a * buffer_x[tx];
    //y[i] = y[i] + a * x[i];
  }
}
```

E.3 AXPY GPU Double

```
/*
 *  kernel_saxpy.cl
 *
 *
 *
 */
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void kernel_daxpy(
    const int n,
    const double a,
    __global const double* x,
    __global double* y,
    __local double* buffer_x,
    __local double* buffer_y
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);
```

```
  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data (full lines) */
  for (int i = ctaStart + tx; i < n; i += totalThreads)
  {
    /* Prefetch x and y */
    buffer_x[tx] = x[i];
    buffer_y[tx] = y[i];

    /* Multiply and write */
    y[i] = buffer_y[tx] + a * buffer_x[tx];
    //y[i] = y[i] + a * x[i];
  }
}
```

E.4 AYPX GPU Single

```
/*
 *  kernel_saypx.cl
 *
 *
 *
 */

__kernel void kernel_saypx(
    const int n,
    const float a,
    __global const float* x,
    __global float* y,
    __local float* buffer_x,
    __local float* buffer_y
)
{
  // Starting point for this block
  int ctaStart = get_group_id(0) * get_local_size(0);

  // Total no. of threads in the kernel
  int totalThreads = get_global_size(0);

  // Get current thread
  int tx = get_local_id(0);

  // Read the data (full lines)
  for (int i = ctaStart + tx; i < n; i += totalThreads)
```

```
  {
    // Prefetch x and y
    buffer_x[tx] = x[i];
    buffer_y[tx] = y[i];


    // Multiply and write
    y[i] = buffer_x[tx] + a * buffer_y[tx];
    //y[i] = x[i] + a * y[i];
  }
}
```

E.5 AYPX GPU Double

```
/*
 *  kernel_saypx.cl
 *
 *
 *
 */

#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void kernel_daypx(
    const int n,
    const double a,
    __global const double* x,
    __global double* y,
    __local double* buffer_x,
    __local double* buffer_y
)
{
  // Starting point for this block
  int ctaStart = get_group_id(0) * get_local_size(0);

  // Total no. of threads in the kernel
  int totalThreads = get_global_size(0);

  // Get current thread
  int tx = get_local_id(0);

  // Read the data (full lines)
  for (int i = ctaStart + tx; i < n; i += totalThreads)
  {
    // Prefetch x and y
    buffer_x[tx] = x[i];
    buffer_y[tx] = y[i];
```

```
    // Multiply and write
    y[i] = buffer_x[tx] + a * buffer_y[tx];
    //y[i] = x[i] + a * y[i];
  }
}
```

E.6 DOT GPU Single

```
/*
 *  kernel_sdot.cl
 *
 *
 *
 */

__kernel void kernel_sdot(
    const int n,
    __global const float* x,
    __global const float* y,
    __global float* res_m,
    __local float* partial_sum
)
{
  float sum = 0;

  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);
  int bx = get_group_id(0);

  /* Read the sum data */
  for (int i = ctaStart + tx; i < n; i += totalThreads)
     {
    sum += x[i] * y[i];
  }

  partial_sum[tx] = sum;

  /* Reduce data for the work-group */
  for (int i = get_local_size(0) >> 1; i > 0; i >>= 1)
  {
    barrier(CLK_LOCAL_MEM_FENCE);
    if (tx < i) {
      partial_sum[tx] += partial_sum[tx + i];
```

```
    }
  }

  if (tx == 0) {
        res_m[bx] = partial_sum[tx];
    }
}
```

E.7 DOT GPU Double

```
/*
 *  kernel_sdot.cl
 *
 *
 *
 */

 #pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void kernel_ddot(
    const int n,
    __global const double* x,
    __global const double* y,
    __global double* res_m,
    __local double* partial_sum
)
{
  double sum = 0.0;

  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);
  int bx = get_group_id(0);

  /* Read the sum data */
  for (int i = ctaStart + tx; i < n; i += totalThreads)
     {
    sum += x[i]*y[i];
  }
  partial_sum[tx] = sum;

  /* Reduce data for the block */
  for (int i = get_local_size(0) >> 1; i > 0; i >>= 1)
  {
```

```
    barrier(CLK_LOCAL_MEM_FENCE);
    if (tx < i) {
      partial_sum[tx] += partial_sum[tx + i];
    }
  }

  if (tx == 0) {
        res_m[bx] = partial_sum[tx];
    }
}
```

E.8 SCAL GPU Single

```
/*
 *  kernel_sscal.cl
 *
 *
 *
 */

__kernel void kernel_sscal(
    const int n,
    const float a,
    __global   float* x,
    __local float* buffer_x
)
{
    // Starting point for this block
  int ctaStart = get_group_id(0) * get_local_size(0);

  // Total no. of threads in the kernel
  int totalThreads = get_global_size(0);

  // Get current thread
  int tx = get_local_id(0);

  // Read the data (full lines)
  for (int i = ctaStart + tx; i < n; i += totalThreads)
  {
    // Prefetch x
    buffer_x[tx] = x[i];

    // Scale and write
    x[i] = buffer_x[tx] * a;
    //x[i] = x[i] * a;
  }
}
```

E.9 SCAL GPU Double

```
/*
 *  kernel_sscal.cl
 *
 *
 *
 */

#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void kernel_dscal(
    const int n,
    const double a,
    __global double* x,
    __local double* buffer_x
)
{
    /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data (full lines) */
  for (int i = ctaStart + tx; i < n; i += totalThreads)
  {
    /* Prefetch x */
    buffer_x[tx] = x[i];

    /* Scale and write */
    x[i] = buffer_x[tx] * a;
    //x[i] = x[i] * a;
  }
}
```

E.10 SPMV CSR GPU Single

```
/*
 *  kernel_sspmv_csr.cl
 *
 *
 * csr port
 *
 *  \brief  Sparse matrix vector multiply in
 *        CSR format, in single precision
```

```
 *   \param  rows Matrix size
 *   \param  d_ptr Row index vector (first non-zero
     element in row)
 *   \param  d_idx Integer index for non-zero element
 *   \param  d_val Value vector. All non-zero values of
     the matrix
 *   \param  d_x Vector being multiplied
 *   \param  d_y Result vector
 */

#define USE_KAHAN_IN_SPMV 0

__kernel void kernel_sspmv_csr(
    const int rows,
    __global const int* d_ptr,
    __global const int* d_idx,
    __global const float* d_val,
    __global const float* d_x,
    __global float* d_y,
    const float alpha
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  //printf("Inside csr\n");
  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
      totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    #if USE_KAHAN_IN_SPMV
      float sum = d_val[iRowBeg] * d_x[(d_idx[iRowBeg])
          - 1];
      float c = 0.0;
      for (int j = iRowBeg + 1; j < iRowEnd; j++) {
        float y = d_val[j] * d_x[(d_idx[j]) - 1] - c;
        float t = sum + y;
```

```
            c = (t - sum) - y;
            sum = t;
        }
    #else
        float sum = 0;
        for (int j = iRowBeg; j < iRowEnd; j++) {
            sum += d_val[j] * d_x[(d_idx[j]) - 1];
        }
    #endif

    /* Write the result to global memory */
    d_y[i] += alpha * sum;
  }
}
```

E.11 SPMV CSR_B0 GPU Single

```
/*
 *  kernel_sspmv_csr.cl
 *
 *
 * csr port
 *
 */

#define USE_KAHAN_IN_SPMV 0

__kernel void kernel_sspmv_csr_b0(
    const int rows,
    __global const int* d_ptr,
    __global const int* d_idx,
    __global const float* d_val,
    __global const float* d_x,
    __global   float* d_y,
    const float alpha
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  //printf("inside CSR4 b0\n");
  /* Read the data*/
```

```
  for (int i = ctaStart + tx; i < rows; i +=
      totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    #if USE_KAHAN_IN_SPMV
      float sum = d_val[iRowBeg] * d_x[(d_idx[iRowBeg])
          - 1];
      float c = 0.0;
      for (int j = iRowBeg + 1; j < iRowEnd; j++) {
        float y = d_val[j] * d_x[(d_idx[j]) - 1] - c;
        float t = sum + y;
            c = (t - sum) - y;
            sum = t;
      }
    #else
      float sum = 0;
      for (int j = iRowBeg; j < iRowEnd; j++) {
        sum += d_val[j] * d_x[(d_idx[j]) - 1];
      }
    #endif

    /* Write the result to global memory */
    d_y[i] = alpha * sum;
  }
}
```

E.12 SPMV CSR_A1 GPU Single

```
/*
 *  kernel_sspmv_csr.cl
 *
 *
 * csr port
 *
 */

#define USE_KAHAN_IN_SPMV 0

__kernel void kernel_sspmv_csr_a1(
    const int rows,
    __global const int* d_ptr,
    __global const int* d_idx,
    __global const float* d_val,
    __global const float* d_x,
```

```
    __global   float* d_y
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
     totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    #if USE_KAHAN_IN_SPMV
      float sum = d_val[iRowBeg] * d_x[(d_idx[iRowBeg])
          - 1];
      float c = 0.0;
      for (int j = iRowBeg + 1; j < iRowEnd; j++) {
        float y = d_val[j] * d_x[(d_idx[j]) - 1] - c;
        float t = sum + y;
            c = (t - sum) - y;
            sum = t;
      }
    #else
      float sum = 0;
      for (int j = iRowBeg; j < iRowEnd; j++) {
        sum += d_val[j] * d_x[(d_idx[j]) - 1];
      }
    #endif

    /* Write the result to global memory */
    d_y[i] += sum;
  }
}
```

E.13 SPMV CSR_A1_B0 GPU Single

```
/*
 *  kernel_sspmv_csr.cl
 *
```

```
 * csr port
 *
 *
 */

#define USE_KAHAN_IN_SPMV 0

__kernel void kernel_sspmv_csr_a1_b0(
    const int rows,
    __global const int* d_ptr,
    __global const int* d_idx,
    __global const float* d_val,
    __global const float* d_x,
    __global float* d_y
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  //printf("inside CSR a1 b0\n");
  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
     totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this work-item */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    #if USE_KAHAN_IN_SPMV
      float sum = d_val[iRowBeg] * d_x[(d_idx[iRowBeg])
         - 1];
      float c = 0.0;
      for (int j = iRowBeg + 1; j < iRowEnd; j++) {
        float y = d_val[j] * d_x[(d_idx[j]) - 1] - c;
        float t = sum + y;
            c = (t - sum) - y;
            sum = t;
      }
    #else
      float sum = 0;
      for (int j = iRowBeg; j < iRowEnd; j++) {
```

```
        sum += d_val[j] * d_x[(d_idx[j]) - 1];
      }
    #endif

    /* Write the result to global memory */
    d_y[i] = sum;
  }
}
```

E.14 SPMV CSR GPU Double

```
/*
 *  kernel_sspmv_csr.cl
 *
 *
 * csr port
 *
 *  \brief  Sparse matrix vector multiply in
 *      CSR format, in single precision
 *  \param  rows Matrix size
 *  \param  d_ptr Row index vector (first non-zero
    element in row)
 *  \param  d_idx Integer index for non-zero element
 *  \param  d_val Value vector. All non-zero values of
    the matrix
 *  \param  d_x Vector being multiplied
 *  \param  d_y Result vector
 */

#pragma OPENCL EXTENSION cl_khr_fp64 : enable

#define USE_KAHAN_IN_SPMV 0

__kernel void kernel_dspmv_csr(
    const int rows,
    __global const int* d_ptr,
    __global const int* d_idx,
    __global const double* d_val,
    __global const double* d_x,
    __global double* d_y,
    const double alpha
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);
```

```
  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
      totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    #if USE_KAHAN_IN_SPMV
      double sum = d_val[iRowBeg] * d_x[(d_idx[iRowBeg])
          - 1];
      double c = 0.0;
      for (int j = iRowBeg + 1; j < iRowEnd; j++) {
        double y = d_val[j] * d_x[(d_idx[j]) - 1] - c;
        double t = sum + y;
            c = (t - sum) - y;
            sum = t;
      }
    #else
      double sum = 0.0;
      for (int j = iRowBeg; j < iRowEnd; j++) {
        sum += d_val[j] * d_x[(d_idx[j]) - 1];
      }
    #endif

    /* Write the result to global memory */
    d_y[i] += alpha * sum;
  }
}
```

E.15 SPMV CSR_B0 GPU Double

```
/*
 *  kernel_sspmv_csr.cl
 *
 *
 * csr port
 *
 */

#pragma OPENCL EXTENSION cl_khr_fp64 : enable

#define USE_KAHAN_IN_SPMV 0
```

```
__kernel void kernel_dspmv_csr_b0(
    const int rows,
    __global const int* d_ptr,
    __global const int* d_idx,
    __global const double* d_val,
    __global const double* d_x,
    __global double* d_y,
    const double alpha
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
      totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    #if USE_KAHAN_IN_SPMV
      double sum = d_val[iRowBeg] * d_x[(d_idx[iRowBeg])
          - 1];
      double c = 0.0;
      for (int j = iRowBeg + 1; j < iRowEnd; j++) {
        double y = d_val[j] * d_x[(d_idx[j]) - 1] - c;
        double t = sum + y;
            c = (t - sum) - y;
            sum = t;
      }
    #else
      double sum = 0.0;
      for (int j = iRowBeg; j < iRowEnd; j++) {
        sum += d_val[j] * d_x[(d_idx[j]) - 1];
      }
    #endif

    /* Write the result to global memory */
    d_y[i] = alpha * sum;
  }
```

```
}
```

E.16 SPMV CSR4 GPU Single

```
/*
 *   kernel_sspmv_csr4.cl
 *
 *
 * csr port
 *
 */

__kernel void kernel_sspmv_csr4(
    const int rows,
    __global const int* d_ptr,
    __global const int4* d_idx,
    __global const float4* d_val,
    __global const float* d_x,
    __global float* d_y,
    const float alpha
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
      totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    float sum = 0;
    for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {
      float4 val, x;
      int4 idx;

      idx = d_idx[j];
      val = d_val[j];
```

```
      idx -= 1;

      x.x = d_x[idx.x];
      x.y = d_x[idx.y];
      x.z = d_x[idx.z];
      x.w = d_x[idx.w];

      sum += dot(x, val);
    }

    /* Write the result to global memory */
    d_y[i] += alpha * sum;
  }
}
```

E.17 SPMV CSR4_B0 GPU Single

```
/*
 *  kernel_sspmv_csr4_b0.cl
 *
 *
 * csr port
 *
 */

__kernel void kernel_sspmv_csr4_b0(
    const int rows,
    __global const int* d_ptr,
    __global const int4* d_idx,
    __global const float4* d_val,
    __global const float* d_x,
    __global float* d_y,
    const float alpha
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
      totalThreads)
  {
    /* Read the beginning and end of the row
```

```
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    float sum = 0;
    for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {
      float4 val, x;
      int4 idx;

      idx = d_idx[j];
      val = d_val[j];

      idx -= 1;

      x.x = d_x[idx.x];
      x.y = d_x[idx.y];
      x.z = d_x[idx.z];
      x.w = d_x[idx.w];

      sum += dot(x, val);
    }

    /* Write the result to global memory */
    d_y[i] = alpha * sum;
  }
}
```

E.18 SPMV CSR4_A1 GPU Single

```
/*
 *   kernel_sspmv_csr4_a1.cl
 *
 *
 * csr port
 *
 */

__kernel void kernel_sspmv_csr4_a1(
    const int rows,
    __global const int* d_ptr,
    __global const int4* d_idx,
    __global const float4* d_val,
    __global const float* d_x,
    __global float* d_y
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);
```

```
  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  //printf("inside CSR4 a1\n");
  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
     totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    float sum = 0;
    for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {
      float4 val, x;
      int4 idx;

      idx = d_idx[j];
      val = d_val[j];

      idx -= 1;

      x.x = d_x[idx.x];
      x.y = d_x[idx.y];
      x.z = d_x[idx.z];
      x.w = d_x[idx.w];

      sum += dot(x, val);
    }

    /* Write the result to global memory */
    d_y[i] += sum;
  }
}
```

E.19 SPMV CSR4_A1_B0 GPU Single

```
/*
 *   kernel_sspmv_csr4_a1_b0.cl
 *
 *
 * csr port
 *
```

```
 */

#pragma OPENCL EXTENSION all : enable

__kernel void kernel_sspmv_csr4_a1_b0(
    const int rows,
    __global const int* d_ptr,
    __global const int4* d_idx,
    __global const float4* d_val,
    __global const float* d_x,
    __global float* d_y
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data*/
  float sum;
  float4 val, x;
  int4 idx;

  int iRowBeg, iRowEnd;

  for (int i = ctaStart + tx; i < rows; i +=
      totalThreads)
  {
    // Read the beginning and end of the row
    // which will be processed by this thread
    iRowBeg = d_ptr[i] - 1;
    iRowEnd = d_ptr[i+1] - 1;

    /// Read and sum for the column vectors
    sum = 0;
    for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {

      idx = d_idx[j];
      idx -= 1;

      x = (float4)(d_x[idx.s0], d_x[idx.s1], d_x[idx.s2
         ], d_x[idx.s3]);
      val = d_val[j];

      sum += dot(x, val);
```

```
    }

    // Write the result to global memory
    d_y[i] = sum;
  }
}
```

E.20 SPMV CSR4 GPU Double

```
/*
 *  kernel_sspmv_csr4.cl
 *
 *
 * csr port
 *
 */
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void kernel_dspmv_csr4(
    const int rows,
    __global const int* d_ptr,
    __global const int4* d_idx,
    __global const double4* d_val,
    __global const double* d_x,
    __global double* d_y,
    const double alpha
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
     totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    double sum = 0.0;
    for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {
```

```
      double4 val, x;
      int4 idx;

      idx = d_idx[j];
      val = d_val[j];

      idx -= 1;

      x.x = d_x[idx.x];
      x.y = d_x[idx.y];
      x.z = d_x[idx.z];
      x.w = d_x[idx.w];

      sum += dot(x, val);
    }

    /* Write the result to global memory */
    d_y[i] += alpha * sum;
  }
}
```

E.21 SPMV CSR4_B0 GPU Double

```
/*
 *  kernel_sspmv_csr4_b0.cl
 *
 *
 * csr port
 *
 */
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void kernel_dspmv_csr4_b0(
    const int rows,
    __global const int* d_ptr,
    __global const int4* d_idx,
    __global const double4* d_val,
    __global const double* d_x,
    __global double* d_y,
    const double alpha
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
```

```
  int tx = get_local_id(0);

  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
     totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    /* Read and sum for the column vectors */
    double sum = 0.0;
    for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {
      double4 val, x;
      int4 idx;

      idx = d_idx[j];
      val = d_val[j];

      idx -= 1;

      x.x = d_x[idx.x];
      x.y = d_x[idx.y];
      x.z = d_x[idx.z];
      x.w = d_x[idx.w];

      sum += dot(x, val);
    }

    /* Write the result to global memory */
    d_y[i] = alpha * sum;
  }
}
```

E.22 SPMV ELL GPU Single

```
/*
 *  kernel_sspmv_ell.cl
 *
 *
 * ell port
 *
 */

#define large_grid_thread_id(void) (((uint)mul24((uint)
   get_local_size(0),(uint)get_group_id(0) + (uint)mul24
   ((uint)get_group_id(1),(uint)get_num_groups(0))) + (
   uint)get_local_id(0)))
```

```
__kernel void kernel_sspmv_ell(
    const int rows,
    const float alpha,
    const int ell_nz_row,
    const int ell_stride,
    __global const int *ell_idx,
    __global const float *ell_val,
    const float beta,
    __global float *d_y,
    __global float *d_x
)
{
  const int row = large_grid_thread_id();

  if(row >= rows){
    return;
  }

  float sum = 0;
  if (beta)
    sum = beta * d_y[row];
  ell_idx += row;
  ell_val += row;
  for(int n = 0; n < ell_nz_row; n++){
    const float A_ij = *ell_val;

    if(A_ij != 0){
      const int col = *ell_idx - 1;
      sum += A_ij * d_x[col]; // this last d_x.. -> can
          be replaced by image access..
    }

    ell_idx += ell_stride;
    ell_val += ell_stride;
  }
  d_y[row] = sum;
}
```

E.23 SPMV ELL GPU Double

```
/*
 *  kernel_sspmv_ell.cl
 *
 *
 * ell port
 *
 */
```

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

#define large_grid_thread_id(void) (((uint)mul24((uint)
    get_local_size(0),(uint)get_group_id(0) + (uint)mul24
    ((uint)get_group_id(1),(uint)get_num_groups(0))) + (
    uint)get_local_id(0)))

__kernel void kernel_dspmv_ell(
     const int rows,
     const double alpha,
     const int ell_nz_row,
     const int ell_stride,
     __global const int *ell_idx,
     __global const double *ell_val,
     const double beta,
     __global double *d_y,
     __global double *d_x
)
{
  const int row = large_grid_thread_id();

  if(row >= rows){
    return;
  }

  double sum = 0;
  if (beta)
    sum = beta * d_y[row];
  ell_idx += row;
  ell_val += row;
  for(int n = 0; n < ell_nz_row; n++){
    const double A_ij = *ell_val;

    if(A_ij != 0){
      const int col = *ell_idx - 1;
      sum += A_ij * d_x[col]; // this last d_x -> can be
          replaced by image access..
    }

    ell_idx += ell_stride;
    ell_val += ell_stride;
  }
  d_y[row] = sum;
}
```

E.24 Kernels GPU single-double (quasi-double)

```
#define USE_KAHAN_IN_SPMV 0
```

```
/**
 * @ds_ops
 * @author NVIDIA
 * @since 2008
 *
 * Defines a double-single (qdouble)
 * operations:
 *   - double-single add (addition)
 *   - double-single sub (subtraction)
 *   - double-single mul (multiplication)
 *
 *   OpenCL port Olav Aanes Fagerlund 2010
 */

/**
 * \brief Double-single (qdouble) addition
 * \param c0,c1 Head and tail for the result
 * \param a0,a1 Head and tail for the first
 *              operand
 * \param b0,b1 Head and tail for the second
 *              operand
 */
float2 dsadd(const float a0, const float a1, const float
    b0, const float b1)
{
  //printf("gets here\n");
  float c0, c1;
  float t1, t2, e;

    // Compute dsa + dsb using Knuth's trick.
    t1 = a0 + b0;
    e = t1 - a0;
    t2 = ((b0 - e) + (a0 - (t1 - e))) + a1 + b1;

    // The result is t1 + t2, after normalization.
    c0 = e = t1 + t2;
    c1 = t2 - (e - t1);

  return (float2)(c0, c1);
}

/**
 * \brief Double-single (qdouble) subtraction
 * \param c0,c1 Head and tail for the result
 * \param a0,a1 Head and tail for the first
 *              operand
 * \param b0,b1 Head and tail for the second
 *              operand
 */
```

```
float2 dssub(const float a0, const float  a1, const
    float b0, const float b1){

  float t1, t2, e, c0, c1;;

  // Compute dsa - dsb using Knuth's trick.
    t1 = a0 - b0;
    e = t1 - a0;
    t2 = ((-b0 - e) + (a0 - (t1 - e))) + a1 - b1;

    // The result is t1 + t2, after normalization.
    c0 = e = t1 + t2;
    c1 = t2 - (e - t1);

  return (float2)(c0, c1);
}


/**
 * \brief Double-single (qdouble) multiplication
 * \param c0,c1 Head and tail for the result
 * \param a0,a1 Head and tail for the first
 *              operand
 * \param b0,b1 Head and tail for the second
 *              operand
 */

float2 dsmul(const float a0, const float a1, const float
     b0, const float b1)
{
  float c0, c1;
  float cona, conb, sa1, sa2, sb1, sb2, c11, c21, c2, t1
      , e, t2;

  cona = a0 * 8193.0f;
  conb = b0 * 8193.0f;
  sa1 = cona - (cona - a0);
  sb1 = conb - (conb - b0);
  sa2 = a0 - sa1;
  sb2 = b0 - sb1;

  // Multilply a0 * b0 using Dekker's method.
  c11 = a0 * b0;
  c21 = (((sa1 * sb1 - c11) + sa1 * sb2) + sa2 * sb1) +
      sa2 * sb2;

  // Compute a0 * b1 + a1 * b0 (only high-order word is
      needed).
  c2 = a0 * b1 + a1 * b0;
```

```
  // Compute (c11, c21) + c2 using Knuth's trick, also
      adding low-order product.
  t1 = c11 + c2;
    e = t1 - c11;
  t2 = ((c2 - e) + (c11 - (t1 - e))) + c21 + a1 * b1;

  // The result is t1 + t2, after normalization.
  c0 = e = t1 + t2;
  c1 = t2 - (e - t1);

  return (float2)(c0, c1);
}

__kernel void kernel_qaxpy(
    const int n,
    const float a0,
    const float a1,
    __global const float* xh,
    __global const float* xt,
    __global float* yh,
    __global float* yt,
    __local float* buffer_xh,
    __local float* buffer_xt,
    __local float* buffer_yh,
    __local float* buffer_yt
)
{
  float2 ret, ret2;

  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  /* Read the data (full lines) */
  for (int i = ctaStart + tx; i < n; i += totalThreads)
  {
    /* Prefetch x and y */
    buffer_xh[tx] = xh[i];
    buffer_xt[tx] = xt[i];
    buffer_yh[tx] = yh[i];
    buffer_yt[tx] = yt[i];

    /* Multiply and write */
```

```
    ret = dsmul(a0, a1, buffer_xh[tx], buffer_xt[tx]);
    ret2 = dsadd(buffer_yh[tx], buffer_yt[tx], ret.x,
       ret.y);
    yh[i] = ret2.x;
    yt[i] = ret2.y;
  }
}

__kernel void kernel_qaypx(
    const int n,
    const float a0,
    const float a1,
    __global const float* xh,
    __global const float* xt,
    __global float* yh,
    __global float* yt,
    __local float* buffer_xh,
    __local float* buffer_xt,
    __local float* buffer_yh,
    __local float* buffer_yt
)
{
  float2 ret, ret2;

  // Starting point for this block
  int ctaStart = get_group_id(0) * get_local_size(0);

  // Total no. of threads in the kernel
  int totalThreads = get_global_size(0);

  // Get current thread
  int tx = get_local_id(0);

  // Read the data (full lines)
  for (int i = ctaStart + tx; i < n; i += totalThreads)
  {
    // Prefetch x and y

    buffer_xh[tx] = xh[i];
    buffer_xt[tx] = xt[i];
    buffer_yh[tx] = yh[i];
    buffer_yt[tx] = yt[i];

    // Multiply and write
    ret = dsmul(a0, a1, buffer_yh[tx], buffer_yt[tx]);
    ret2 = dsadd(buffer_xh[tx], buffer_xt[tx], ret.x,
       ret.y);
    yh[i] = ret2.x;
    yt[i] = ret2.y;
```

```
  }

}

__kernel void kernel_qdot(
    const int n,
    __global const float* xh,
    __global const float* xt,
    __global const float* yh,
    __global const float* yt,
    __global float* d_sh,
    __global float* d_st,
    __local float* partial_sum_h,
    __local float* partial_sum_t
)
{
  float2 c, sum = (float2)(0, 0), p_sum;

  // Starting point for this block
  int ctaStart = get_group_id(0) * get_local_size(0);
  // Total no. of threads in the kernel
  int totalThreads = get_global_size(0);

  // Get current thread
  int tx = get_local_id(0);
  int bx = get_group_id(0);

  // Read the sum data */
  for (int i = ctaStart + tx; i < n; i += totalThreads)
  {
    // Multiply
    c = dsmul(xh[i], xt[i], yh[i], yt[i]);

    // Accumulate
    sum = dsadd(c.x, c.y, sum.x, sum.y);
  }
  partial_sum_h[tx] = sum.x;
  partial_sum_t[tx] = sum.y;

  // Reduce data for the block
  for (int i = get_local_size(0) >> 1; i > 0; i >>= 1)
  {
    barrier(CLK_LOCAL_MEM_FENCE);
    if (tx < i) {
      p_sum = dsadd(partial_sum_h[tx], partial_sum_t[tx
         ], partial_sum_h[tx + i], partial_sum_t[tx + i
         ]);
      partial_sum_h[tx] = p_sum.x;
      partial_sum_t[tx] = p_sum.y;
```

```
      }
    }
    if (tx == 0) {
      d_sh[bx] = partial_sum_h[tx];
      d_st[bx] = partial_sum_t[tx];
      }
}

__kernel void kernel_qscal(
      const int n,
      const float a0,
      const float a1,
      __global float* xh,
      __global float* xt,
      __local float* buffer_xh,
      __local float* buffer_xt
)
{
    float2 res;

      // Starting point for this block
    int ctaStart = get_group_id(0) * get_local_size(0);

    // Total no. of threads in the kernel
    int totalThreads = get_global_size(0);

    // Get current thread
    int tx = get_local_id(0);

    // Read the data (full lines)
    for (int i = ctaStart + tx; i < n; i += totalThreads)
    {
      // Prefetch x
      buffer_xh[tx] = xh[i];
      buffer_xt[tx] = xt[i];

      // Scale and write
      res = dsmul (buffer_xh[tx], buffer_xt[tx], a0, a1);
      xh[i] = res.x; //xh[i] * a0;
      xt[i] = res.y; //xt[i] * a1;
    }
}

__kernel void kernel_qspmv_csr_a1_b0(
      const int rows,
      __global const int* d_ptr,
      __global const int* d_idx,
      __global const float* d_valh,
      __global const float* d_valt,
```

```
    __global const float* d_xh,
    __global const float* d_xt,
    __global float* d_yh,
    __global float* d_yt
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  int i;
  float2 c;

  /* Read the data*/
  for (i = ctaStart + tx; i < rows; i += totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    float2 sum = (float2)(0, 0);

    /* Read and sum for the column vectors */
    #if USE_KAHAN_IN_SPMV
      int col = d_idx[iRowBeg] - 1;
      sum = dsmul(d_valh[iRowBeg], d_valt[iRowBeg], d_xh
          [col], d_xt[col]);
      float2 cc = (float2)(0.0f, 0.0f);
      for (int j= iRowBeg + 1; j < iRowEnd; j++) {
        float2 y, t;
        col = d_idx[j] - 1;
        c = dsmul(d_valh[j], d_valt[j], d_xh[col], d_xt[
            col]);
        y = dssub(c.x, c.y, cc.x, cc.y);
        t = dsadd(sum.x, sum.y, y.x, y.y);
        c = dssub(t.x, t.y, sum.x, sum.y);
        cc = dssub(c.x, c.y, y.x, y.y);
        sum.x = t.x;
        sum.y = t.y;
      }
    #else
      for ( int j = iRowBeg; j < iRowEnd; j++) {
        int col = d_idx[j] - 1;
```

```
        c = dsmul(d_valh[j], d_valt[j], d_xh[col], d_xt[
            col]);
        sum = dsadd(c.x, c.y, sum.x, sum.y);
      }
    #endif

    /* Write the result to global memory */
    d_yh[i] = sum.x;
    d_yt[i] = sum.y;
  }
}

__kernel void kernel_qspmv_csr_a1(
    const int rows,
    __global const int* d_ptr,
    __global const int* d_idx,
    __global const float* d_valh,
    __global const float* d_valt,
    __global const float* d_xh,
    __global const float* d_xt,
    __global float* d_yh,
    __global float* d_yt
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  int i;
  float2 c;

  /* Read the data*/
  for (i = ctaStart + tx; i < rows; i += totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    float2 sum = (float2)(0, 0);

    /* Read and sum for the column vectors */
    #if USE_KAHAN_IN_SPMV
      int col = d_idx[iRowBeg] - 1;
```

```
      sum = dsmul(d_valh[iRowBeg], d_valt[iRowBeg], d_xh
          [col], d_xt[col]);
      float2 cc = (float2)(0.0f, 0.0f);
      for (int j= iRowBeg + 1; j < iRowEnd; j++) {
        float2 y, t;
        col = d_idx[j] - 1;
        c = dsmul(d_valh[j], d_valt[j], d_xh[col], d_xt[
            col]);
        y = dssub(c.x, c.y, cc.x, cc.y);
        t = dsadd(sum.x, sum.y, y.x, y.y);
        c = dssub(t.x, t.y, sum.x, sum.y);
        cc = dssub(c.x, c.y, y.x, y.y);
        sum.x = t.x;
        sum.y = t.y;
      }
    #else
      for ( int j = iRowBeg; j < iRowEnd; j++) {
        int col = d_idx[j] - 1;
        c = dsmul(d_valh[j], d_valt[j], d_xh[col], d_xt[
            col]);
        sum = dsadd(c.x, c.y, sum.x, sum.y);
      }
    #endif

    /* Write the result to global memory */
    d_yh[i] += sum.x;
    d_yt[i] += sum.y;
  }
}

__kernel void kernel_qspmv_csr_b0(
    const int rows,
    __global const int* d_ptr,
    __global const int* d_idx,
    __global const float* d_valh,
    __global const float* d_valt,
    __global const float* d_xh,
    __global const float* d_xt,
    __global float* d_yh,
    __global float* d_yt,
    const float alpha0,
    const float alpha1
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);
```

```
/* Get current thread */
int tx = get_local_id(0);

int i;
float2 c, res;

/* Read the data*/
for (i = ctaStart + tx; i < rows; i += totalThreads)
{
  /* Read the beginning and end of the row
   * which will be processed by this thread */
  int iRowBeg = d_ptr[i] - 1;
  int iRowEnd = d_ptr[i+1] - 1;

  float2 sum = (float2)(0, 0);

  /* Read and sum for the column vectors */
  #if USE_KAHAN_IN_SPMV
    int col = d_idx[iRowBeg] - 1;
    sum = dsmul(d_valh[iRowBeg], d_valt[iRowBeg], d_xh
        [col], d_xt[col]);
    float2 cc = (float2)(0.0f, 0.0f);
    for (int j= iRowBeg + 1; j < iRowEnd; j++) {
      float2 y, t;
      col = d_idx[j] - 1;
      c = dsmul(d_valh[j], d_valt[j], d_xh[col], d_xt[
          col]);
      y = dssub(c.x, c.y, cc.x, cc.y);
      t = dsadd(sum.x, sum.y, y.x, y.y);
      c = dssub(t.x, t.y, sum.x, sum.y);
      cc = dssub(c.x, c.y, y.x, y.y);
      sum.x = t.x;
      sum.y = t.y;
    }
  #else
    for ( int j = iRowBeg; j < iRowEnd; j++) {
      int col = d_idx[j] - 1;
      c = dsmul(d_valh[j], d_valt[j], d_xh[col], d_xt[
          col]);
      sum = dsadd(c.x, c.y, sum.x, sum.y);
    }
  #endif

  /* Write the result to global memory */
  res = dsmul(alpha0, alpha1, sum.x, sum.y);
  d_yh[i] = res.x;
  d_yt[i] = res.y;
}
```

```
}

__kernel void kernel_qspmv_csr(
    const int rows,
    __global const int* d_ptr,
    __global const int* d_idx,
    __global const float* d_valh,
    __global const float* d_valt,
    __global const float* d_xh,
    __global const float* d_xt,
    __global float* d_yh,
    __global float* d_yt,
    const float alpha0,
    const float alpha1
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  int i;
  float2 c;

  /* Read the data*/
  for (i = ctaStart + tx; i < rows; i += totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    float2 sum = (float2)(0, 0);

    /* Read and sum for the column vectors */
    #if USE_KAHAN_IN_SPMV
      int col = d_idx[iRowBeg] - 1;
      sum = dsmul(d_valh[iRowBeg], d_valt[iRowBeg], d_xh
          [col], d_xt[col]);
      float2 cc = (float2)(0.0f, 0.0f);
      for (int j= iRowBeg + 1; j < iRowEnd; j++) {
        float2 y, t;
        col = d_idx[j] - 1;
        c = dsmul(d_valh[j], d_valt[j], d_xh[col], d_xt[
            col]);
```

```
        y = dssub(c.x, c.y, cc.x, cc.y);
        t = dsadd(sum.x, sum.y, y.x, y.y);
        c = dssub(t.x, t.y, sum.x, sum.y);
        cc = dssub(c.x, c.y, y.x, y.y);
        sum.x = t.x;
        sum.y = t.y;
      }
    #else
      for ( int j = iRowBeg; j < iRowEnd; j++) {
        int col = d_idx[j] - 1;
        c = dsmul(d_valh[j], d_valt[j], d_xh[col], d_xt[
            col]);
        sum = dsadd(c.x, c.y, sum.x, sum.y);
      }
    #endif

    // Multiply with alpha
    sum = dsmul(alpha0, alpha1, sum.x, sum.y);

    /* Write the result to global memory */
    d_yh[i] += sum.x;
    d_yt[i] += sum.y;
  }
}

__kernel void kernel_qspmv_csr4_a1_b0(
    const int rows,
    __global const int* d_ptr,
    __global const int4* d_idx,
    __global const float4* d_valh,
    __global const float4* d_valt,
    __global const float* d_xh,
    __global const float* d_xt,
    __global float* d_yh,
    __global float* d_yt
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  float2 c;

  /* Read the data*/
```

```
for (int i = ctaStart + tx; i < rows; i +=
    totalThreads)
{
  /* Read the beginning and end of the row
   * which will be processed by this thread */
  int iRowBeg = d_ptr[i] - 1;
  int iRowEnd = d_ptr[i+1] - 1;

  float2 sum = (float2)(0, 0);

  //#pragma unroll
  for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {
    float4 valh, valt, xh, xt;
    int4 idx;

    // Read idx and val
    idx = d_idx[j];
    valh = d_valh[j];
    valt = d_valt[j];

    // Idx is base 1, change to 0
    idx.x -= 1;
    idx.y -= 1;
    idx.z -= 1;
    idx.w -= 1;

    // Read head part of x
    xh.x = d_xh[idx.x];
    xh.y = d_xh[idx.y];
    xh.z = d_xh[idx.z];
    xh.w = d_xh[idx.w];

    // Read tail part of x
    xt.x = d_xt[idx.x];
    xt.y = d_xt[idx.y];
    xt.z = d_xt[idx.z];
    xt.w = d_xt[idx.w];

    // Multiply and add
    c = dsmul(xh.x, xt.x, valh.x, valt.x);
    sum = dsadd(c.x, c.y, sum.x, sum.y);

    c = dsmul(xh.y, xt.y, valh.y, valt.y);
    sum = dsadd(c.x, c.y, sum.x, sum.y);

    c = dsmul(xh.z, xt.z, valh.z, valt.z);
    sum =  dsadd(c.x, c.y, sum.x, sum.y);

    c = dsmul(xh.w, xt.w, valh.w, valt.w);
```

```
      sum = dsadd(c.x, c.y, sum.x, sum.y);
    }

    // Write the result to global memory
    d_yh[i] = sum.x;
    d_yt[i] = sum.y;
  }
}

__kernel void kernel_qspmv_csr4_a1(
    const int rows,
    __global const int* d_ptr,
    __global const int4* d_idx,
    __global const float4* d_valh,
    __global const float4* d_valt,
    __global const float* d_xh,
    __global const float* d_xt,
    __global float* d_yh,
    __global float* d_yt
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  float2 c, res;

  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
     totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    float2 sum = (float2)(0, 0);

    for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {
      float4 valh, valt, xh, xt;
      int4 idx;

      // Read idx and val
      idx = d_idx[j];
```

```
      valh = d_valh[j];
      valt = d_valt[j];

      // Idx is base 1, change to 0
      idx.x -= 1;
      idx.y -= 1;
      idx.z -= 1;
      idx.w -= 1;

      // Read head part of x
      xh.x = d_xh[idx.x];
      xh.y = d_xh[idx.y];
      xh.z = d_xh[idx.z];
      xh.w = d_xh[idx.w];

      // Read tail part of x
      xt.x = d_xt[idx.x];
      xt.y = d_xt[idx.y];
      xt.z = d_xt[idx.z];
      xt.w = d_xt[idx.w];

      // Multiply and add
      c = dsmul(xh.x, xt.x, valh.x, valt.x);
      sum = dsadd(c.x, c.y, sum.x, sum.y);

      c = dsmul(xh.y, xt.y, valh.y, valt.y);
      sum = dsadd(c.x, c.y, sum.x, sum.y);

      c = dsmul(xh.z, xt.z, valh.z, valt.z);
      sum =  dsadd(c.x, c.y, sum.x, sum.y);

      c = dsmul(xh.w, xt.w, valh.w, valt.w);
      sum = dsadd(c.x, c.y, sum.x, sum.y);
    }

    // Add/write the result to global memory
    res = dsadd(d_yh[i], d_yt[i], sum.x, sum.y);
    d_yh[i] = res.x;
    d_yt[i] = res.y;
  }
}

__kernel void kernel_qspmv_csr4_b0(
    const int rows,
    __global const int* d_ptr,
    __global const int4* d_idx,
    __global const float4* d_valh,
    __global const float4* d_valt,
    __global const float* d_xh,
```

```
    __global const float* d_xt,
    __global float* d_yh,
    __global float* d_yt,
    const float alpha0,
    const float alpha1
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);

  /* Get current thread */
  int tx = get_local_id(0);

  float2 c, res;

  /* Read the data*/
  for (int i = ctaStart + tx; i < rows; i +=
     totalThreads)
  {
    /* Read the beginning and end of the row
     * which will be processed by this thread */
    int iRowBeg = d_ptr[i] - 1;
    int iRowEnd = d_ptr[i+1] - 1;

    float2 sum = (float2)(0, 0);

    for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {
      float4 valh, valt, xh, xt;
      int4 idx;

      // Read idx and val
      idx = d_idx[j];
      valh = d_valh[j];
      valt = d_valt[j];

      // Idx is base 1, change to 0
      idx.x -= 1;
      idx.y -= 1;
      idx.z -= 1;
      idx.w -= 1;

      // Read head part of x
      xh.x = d_xh[idx.x];
      xh.y = d_xh[idx.y];
      xh.z = d_xh[idx.z];
      xh.w = d_xh[idx.w];
```

```
      // Read tail part of x
      xt.x = d_xt[idx.x];
      xt.y = d_xt[idx.y];
      xt.z = d_xt[idx.z];
      xt.w = d_xt[idx.w];

      // Multiply and add
      c = dsmul(xh.x, xt.x, valh.x, valt.x);
      sum = dsadd(c.x, c.y, sum.x, sum.y);

      c = dsmul(xh.y, xt.y, valh.y, valt.y);
      sum = dsadd(c.x, c.y, sum.x, sum.y);

      c = dsmul(xh.z, xt.z, valh.z, valt.z);
      sum =  dsadd(c.x, c.y, sum.x, sum.y);

      c = dsmul(xh.w, xt.w, valh.w, valt.w);
      sum = dsadd(c.x, c.y, sum.x, sum.y);
    }

    // Multiply with alpha and write to global memory
    res = dsmul(alpha0, alpha1, sum.x, sum.y);

    d_yh[i] = res.x;
    d_yt[i] = res.y;
  }
}

__kernel void kernel_qspmv_csr4(
    const int rows,
    __global const int* d_ptr,
    __global const int4* d_idx,
    __global const float4* d_valh,
    __global const float4* d_valt,
    __global const float* d_xh,
    __global const float* d_xt,
    __global float* d_yh,
    __global float* d_yt,
    const float alpha0,
    const float alpha1
)
{
  /* Starting point for this block */
  int ctaStart = get_group_id(0) * get_local_size(0);

  /* Total no. of threads in the kernel */
  int totalThreads = get_global_size(0);
```

```
/* Get current thread */
int tx = get_local_id(0);

float2 c, res;

/* Read the data*/
for (int i = ctaStart + tx; i < rows; i +=
    totalThreads)
{
  /* Read the beginning and end of the row
   * which will be processed by this thread */
  int iRowBeg = d_ptr[i] - 1;
  int iRowEnd = d_ptr[i+1] - 1;

  float2 sum = (float2)(0, 0);

  //#pragma unroll
  for (int j = iRowBeg / 4; j < iRowEnd / 4; j++) {
    float4 valh, valt, xh, xt;
    int4 idx;

    // Read idx and val
    idx = d_idx[j];
    valh = d_valh[j];
    valt = d_valt[j];

    // Idx is base 1, change to 0
    idx.x -= 1;
    idx.y -= 1;
    idx.z -= 1;
    idx.w -= 1;

    // Read head part of x
    xh.x = d_xh[idx.x];
    xh.y = d_xh[idx.y];
    xh.z = d_xh[idx.z];
    xh.w = d_xh[idx.w];

    // Read tail part of x
    xt.x = d_xt[idx.x];
    xt.y = d_xt[idx.y];
    xt.z = d_xt[idx.z];
    xt.w = d_xt[idx.w];

    // Multiply and add
    c = dsmul(xh.x, xt.x, valh.x, valt.x);
    sum = dsadd(c.x, c.y, sum.x, sum.y);

    c = dsmul(xh.y, xt.y, valh.y, valt.y);
```

```
      sum = dsadd(c.x, c.y, sum.x, sum.y);

      c = dsmul(xh.z, xt.z, valh.z, valt.z);
      sum =  dsadd(c.x, c.y, sum.x, sum.y);

      c = dsmul(xh.w, xt.w, valh.w, valt.w);
      sum = dsadd(c.x, c.y, sum.x, sum.y);
    }

    // Multiply with alpha
    sum = dsmul(alpha0, alpha1, sum.x, sum.y);

    // Write the result to global memory
    res = dsadd(d_yh[i], d_yt[i], sum.x, sum.y);

    d_yh[i] = res.x;
    d_yt[i] = res.y;
  }
}

#define large_grid_thread_id(void) (((uint)mul24((uint)
   get_local_size(0),(uint)get_group_id(0) + (uint)mul24
   ((uint)get_group_id(1),(uint)get_num_groups(0))) + (
   uint)get_local_id(0)))

__kernel void kernel_qspmv_ell(
    const int   rows,
    const float alpha0,
    const float alpha1,
    const int    ell_nz_row,
    const int    ell_stride,
    __global const int   *ell_idx,
    __global const float *ell_valh,
    __global const float *ell_valt,
    const float  beta0,
    const float  beta1,
    __global  float *d_yh,
    __global  float *d_yt,
    __global  float *d_xh,
    __global  float *d_xt
)
{
  const int row = large_grid_thread_id();

  if(row >= rows){
    return;
  }

  float2 sum = (float2)(0, 0);
```

```
  if (beta0 || beta1)
    sum = dsmul(beta0, beta1, d_yh[row], d_yt[row]);

  ell_idx += row;
  ell_valh += row;
  ell_valt += row;

  for(int n = 0; n < ell_nz_row; n++){
    const float A_ij_h = *ell_valh;
    const float A_ij_t = *ell_valt;

    if(A_ij_h != 0){
      float2 c;
      const int col = *ell_idx - 1;
      c = dsmul(A_ij_h, A_ij_t, d_xh[col], d_xt[col]);
          // this last d_x.. -> can be replaced by image
          access..
      sum = dsadd(sum.x, sum.y, c.x, c.y);
    }
    ell_idx += ell_stride;
    ell_valh += ell_stride;
    ell_valt += ell_stride;
  }

  d_yh[row] = sum.x;
  d_yt[row] = sum.y;
}
```

E.25 Kernels GPU single set-up

```
/*
 *  kernels_single.c
 *
 *
 *  Created by Olav Aanes Fagerlund.
 *
 */

#define LOGGER_NAME "CUKr.blas.impl.gpu.opencl.
   kernels_single"
#include "kernels_single.h"

#define AUTO_LOCAL 0
#define ORIGINAL_VALUES 0

#if ORIGINAL_VALUES
// Work-group sizes and the number of those equal to the
     CUDA version
#include "kernels_config_orig.h"
```

```
#else
// Work-group sizes and the number of those giving
    better OpenCL performance
#include "kernels_config.h"
#endif

float opencl_kernel_sdot(int n, cl_mem* x, cl_mem* y)
{

  int LOCAL_SIZE = CUKR_SDOT_THREADS;

  #if AUTO_LOCAL
  LOCAL_SIZE = AUTO_LOCAL_SIZE_SDOT;
  #endif

  size_t sizes[5];
  void *values[5];

  float res = 0;
  float* res_arr = malloc(CUKR_SDOT_CTAS * sizeof(float)
      );
  cl_mem res_buf = clCreateBuffer(ComputeContext,
      CL_MEM_WRITE_ONLY, CUKR_SDOT_CTAS * sizeof(cl_float
      ), /*(void*)res*/ NULL, &err);
  if (err != CL_SUCCESS) {
    printf("Setting␣up␣res_buf␣for␣sdot␣failed!\n");
  }

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(cl_mem);
  values[1] = (void *)x;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)y;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)&res_buf;
  sizes[4] =  LOCAL_SIZE * sizeof(cl_float);
  values[4] = NULL;

  int x_inc;
  for (x_inc = 0; x_inc<5; x_inc++) {
    err = clSetKernelArg(ComputeKernel_sdot, x_inc,
        sizes[x_inc], values[x_inc]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed␣%d␣arg␣sdot\n", x_inc);
      //return -1;
    }
    if (err != CL_SUCCESS)
```

```
    {
      printf("clSetKernelArg␣%d␣failed\n", x_inc);
      //return -1;
    }
  }

  size_t global[1];
  size_t local[1];

  local[0]  = LOCAL_SIZE;
  global[0] = CUKR_SDOT_CTAS * local[0];

  /* Enqueue kernel for execution */
  err = clEnqueueNDRangeKernel(ComputeCommands,
      ComputeKernel_sdot, 1, NULL, global, local, 0, NULL
      , &sdot_event);

  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣failed␣%d\n", n);
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &sdot_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }

  // Get result
  err = clEnqueueReadBuffer(ComputeCommands, res_buf,
      CL_TRUE, 0, CUKR_SDOT_CTAS * sizeof(cl_float), (
      void*)res_arr, 0, NULL, NULL);
  if (err == CL_INVALID_COMMAND_QUEUE) {
    printf("YES!!!\n");
  }
  else if (err != CL_SUCCESS) {
    printf("readback␣of␣res_buf␣in␣sdot␣failed!␣err-code
        :␣%d\n", err);
  }

  err = clReleaseMemObject(res_buf);
  if (err != CL_SUCCESS) {
    printf("release␣of␣cl␣memobject␣res_buf␣in␣sdot␣
        failed!\n");
  }

  int inc;
  for (inc = 0; inc < CUKR_SDOT_CTAS; inc++) {
    res += res_arr[inc];
```

```
  }

  // free the array
  free(res_arr);
  res_arr = NULL;

  return res;
}

void opencl_kernel_saxpy(int n, float a, cl_mem* cl_d_x,
    cl_mem* cl_d_y)
{
  // Set workgroup sizes
  /*
  err = clGetKernelWorkGroupInfo(ComputeKernel_saxpy,
      ComputeDeviceId, CL_KERNEL_WORK_GROUP_SIZE, sizeof(
      size_t), &auto_local, NULL);
  if (err != CL_SUCCESS) {
    printf("clGetKernelWorkGroupInfo failed! : %d\n",
        err);
  }
  */

  size_t sizes[6];
  void *values[6];

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(float);
  values[1] = (void *)&a;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)cl_d_x;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)cl_d_y;

  sizes[4] = CUKR_SAXPY_THREADS * sizeof(cl_float);
  values[4] = NULL;
  sizes[5] = CUKR_SAXPY_THREADS * sizeof(cl_float);
  values[5] = NULL;

  /* Kernel invocation */
  int x;
  for (x = 0; x<6; x++) {
    err = clSetKernelArg(ComputeKernel_saxpy, x, sizes[x
        ], values[x]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed %d failed\n", x);
    }
```

```
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", x);
    }
  }

  size_t global[1];
  size_t local[1];

  local[0] = CUKR_SAXPY_THREADS;
  global[0] = CUKR_SAXPY_CTAS * local[0];

  /* Enqueue kernel for execution */
  err = clEnqueueNDRangeKernel(ComputeCommands,
      ComputeKernel_saxpy, 1, NULL, global, local, 0,
      NULL, &saxpy_event);
  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣failed␣%d\n",n);
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &saxpy_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }
}

void opencl_kernel_saypx(int n, float a, cl_mem* cl_d_x,
     cl_mem* cl_d_y)
{
  // Set workgroup sizes
  /*
  err = clGetKernelWorkGroupInfo(ComputeKernel_saypx,
      ComputeDeviceId, CL_KERNEL_WORK_GROUP_SIZE, sizeof(
      size_t), &auto_local, NULL);
  if (err != CL_SUCCESS) {
    printf("clGetKernelWorkGroupInfo failed! : %d\n",
        err);
  }
  */

  size_t sizes[6];
  void *values[6];

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(float);
  values[1] = (void *)&a;
```

```
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)cl_d_x;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)cl_d_y;

  sizes[4] = CUKR_SAYPX_THREADS * sizeof(cl_float);
  values[4] = NULL;
  sizes[5] = CUKR_SAYPX_THREADS * sizeof(cl_float);
  values[5] = NULL;

  // Set kernel args
  int x;
  for (x = 0; x<6; x++) {
    err = clSetKernelArg(ComputeKernel_saypx, x, sizes[x
       ], values[x]);
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", x);
      //return -1;
    }
  }
  //}

  size_t global[1];
  size_t local[1];

  local[0] = CUKR_SAYPX_THREADS;
  global[0] = CUKR_SAYPX_CTAS * local[0];

  // Enqueue kernel for execution
  err = clEnqueueNDRangeKernel(ComputeCommands,
     ComputeKernel_saypx, 1, NULL, global, local, 0,
     NULL, &saypx_event);
  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣failed␣%d\n",n);
  }

  err = clWaitForEvents(1, &saypx_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }

}

void opencl_kernel_sscal(int n, float a, cl_mem* x)
{
  size_t sizes[4];
  void *values[4];
```

```
  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(float);
  values[1] = (void *)&a;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)x;
  sizes[3] = CUKR_SSCAL_THREADS * sizeof(cl_float);
  values[3] = NULL;

  /* Kernel invocation */
  int i;
  for (i = 0; i<4; i++) {
    err = clSetKernelArg(ComputeKernel_sscal, i, sizes[i
       ], values[i]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed␣%d␣failed\n", i);
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", i);
      //return -1;
    }
  }

  size_t global[1];
  size_t local[1];

  local[0] = CUKR_SSCAL_THREADS;
  global[0] = CUKR_SSCAL_CTAS * local[0];

  /* Enqueue kernel for execution */
  err = clEnqueueNDRangeKernel(ComputeCommands,
      ComputeKernel_sscal, 1, NULL, global, local, 0,
      NULL, &sscal_event);
  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣sscal␣failed␣%d\n",n);
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &sscal_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }

}
```

```
void opencl_scopy(int n, cl_mem* cl_d_x, int incx,
    cl_mem* cl_d_y, int incy)
{
  err = clEnqueueCopyBuffer(ComputeCommands, *cl_d_x, *
      cl_d_y, 0, 0, n * sizeof(cl_float), 0, NULL, &
      scopy_event);
  if (err != CL_SUCCESS)
  {
    printf("clEnqueueCopyBuffer failed %d\n", n);
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &scopy_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents failed!\n");
  }
}

void opencl_sspmv_csr(int rows, int cols, int nz, float
    alpha, cl_mem* d_ptr,
             cl_mem* d_idx, cl_mem* d_val, cl_mem* d_x,
                float beta, cl_mem* d_y)
{
  size_t sizes[7];
  void *values[7];

  sizes[0] = sizeof(int);
  values[0] = (void *)&rows;
  sizes[1] = sizeof(cl_mem);
  values[1] = (void *)d_ptr;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)d_idx;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)d_val;
  sizes[4] = sizeof(cl_mem);
  values[4] = (void *)d_x;
  sizes[5] = sizeof(cl_mem);
  values[5] = (void *)d_y;
  sizes[6] = sizeof(float);
  values[6] = (void *)&alpha;

  size_t global[1];
  size_t local[1];

  int inc;

  /* If beta != 0 */
  if (beta != 0)
```

```
{
  /* If beta != 1, do a scaling first */
  if (beta != 1)
    opencl_kernel_sscal(rows, beta, d_y);

  /* If alpha = 1, no need to consider */
    if (alpha == 1) {
    // Set workgroup sizes
    local[0] = CUKR_SSPMV_CSR_THREADS; //512;
    global[0] = CUKR_SSPMV_CSR_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 6; inc++) {
      err = clSetKernelArg(ComputeKernel_sspmv_csr_a1,
          inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
       ComputeKernel_sspmv_csr_a1, 1, NULL, global,
       local, 0, NULL, &sspmv_csr_a1_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣sspmv_csr_a1␣failed␣%d\n
         ", nz);
    }

    // Synchronize for timing
    err = clWaitForEvents(1, &sspmv_csr_a1_event);
    if (err != CL_SUCCESS) {
      printf("clWaitForEvents␣failed!\n");
    }
    }
  /* Else, take the most general case */
  else {
    // Set workgroup sizes
    local[0] = CUKR_SSPMV_CSR_THREADS; //512;
    global[0] = CUKR_SSPMV_CSR_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 7; inc++) {
```

```
      err = clSetKernelArg(ComputeKernel_sspmv_csr,
          inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_sspmv_csr, 1, NULL, global, local
        , 0, NULL, &sspmv_csr_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣sspmv_csr␣failed␣%d\n",
          nz);
    }
    // Synchronize for timing
    err = clWaitForEvents(1, &sspmv_csr_event);
    if (err != CL_SUCCESS) {
      printf("clWaitForEvents␣failed!\n");
    }
  }

}
/* If beta = 0 */
else {
  /* If alpha = 1 as well, no need to
   * consider them both */
  if (alpha == 1) {
    // Set workgroup sizes
    local[0] = CUKR_SSPMV_CSR_THREADS; //512;
    global[0] = CUKR_SSPMV_CSR_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 6; inc++) {
      err = clSetKernelArg(
          ComputeKernel_sspmv_csr_a1_b0, inc, sizes[inc
          ], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
```

```
    {
      printf("clSetKernelArg␣%d␣failed\n", inc);
      //return -1;
    }
  }
  err = clEnqueueNDRangeKernel(ComputeCommands,
      ComputeKernel_sspmv_csr_a1_b0, 1, NULL, global,
       local, 0, NULL, &sspmv_csr_a1_b0_event);
  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣sspmv_csr_a1_b0␣failed␣%
        d\n", nz);
  }
  // Synchronize for timing
  err = clWaitForEvents(1, &sspmv_csr_a1_b0_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }
}
    /* If alpha != 1, have to consider it */
    else {
      // Set workgroup sizes
  local[0] = CUKR_SSPMV_CSR_THREADS; //512;
  global[0] = CUKR_SSPMV_CSR_CTAS * local[0];

  // Set kernel args
  for (inc = 0; inc < 7; inc++) {
    err = clSetKernelArg(ComputeKernel_sspmv_csr_b0,
         inc, sizes[inc], values[inc]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed␣%d␣failed\n", inc);
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", inc);
      //return -1;
    }
  }
  err = clEnqueueNDRangeKernel(ComputeCommands,
      ComputeKernel_sspmv_csr_b0, 1, NULL, global,
      local, 0, NULL, &sspmv_csr_b0_event);
  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣sspmv_csr_b0␣failed␣%d\n
        ", nz);
  }
  // Synchronize for timing
```

```
      err = clWaitForEvents(1, &sspmv_csr_b0_event);
      if (err != CL_SUCCESS) {
        printf("clWaitForEvents␣failed!\n");
      }
        }

  }
}

void opencl_sspmv_csr4(int rows, int cols, int nz, float
    alpha, cl_mem* d_ptr,
            cl_mem* d_idx, cl_mem* d_val, cl_mem* d_x,
               float beta, cl_mem* d_y)
{
  size_t sizes[7];
  void *values[7];

  sizes[0] = sizeof(int);
  values[0] = (void *)&rows;
  sizes[1] = sizeof(cl_mem);
  values[1] = (void *)d_ptr;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)d_idx;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)d_val;
  sizes[4] = sizeof(cl_mem);
  values[4] = (void *)d_x;
  sizes[5] = sizeof(cl_mem);
  values[5] = (void *)d_y;
  sizes[6] = sizeof(float);
  values[6] = (void *)&alpha;

  size_t global[1];
  size_t local[1];

  int inc;

  /* If beta != 0 */
  if (beta != 0)
  {
    /* If beta != 1, do a scaling first */
    if (beta != 1)
      opencl_kernel_sscal(rows, beta, d_y);

    /* If alpha = 1, no need to consider */
      if (alpha == 1) {
          // Set workgroup sizes
      local[0] = CUKR_SSPMV_CSR4_THREADS; //512;
      global[0] = CUKR_SSPMV_CSR4_CTAS * local[0];
```

```
    // Set kernel args
    for (inc = 0; inc < 6; inc++) {
      err = clSetKernelArg(ComputeKernel_sspmv_csr4_a1
          , inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_sspmv_csr4_a1, 1, NULL, global,
        local, 0, NULL, &sspmv_csr4_a1_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣sspmv_csr4_a1␣failed␣%d\
          n", nz);
    }
    // Synchronize for timing
    err = clWaitForEvents(1, &sspmv_csr4_a1_event);
    if (err != CL_SUCCESS) {
      printf("clWaitForEvents␣failed!\n");
    }
    }
  /* Else, take the most general case */
  else {
    // Set workgroup sizes
    size_t auto_local;
    local[0] = CUKR_SSPMV_CSR4_THREADS; //512;
    global[0] = CUKR_SSPMV_CSR4_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 7; inc++) {
      err = clSetKernelArg(ComputeKernel_sspmv_csr4,
          inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
```

```
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_sspmv_csr4, 1, NULL, global,
        local, 0, NULL, &sspmv_csr4_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣sspmv_csr4␣failed␣%d\n",
          nz);
    }
    // Synchronize for timing
    err = clWaitForEvents(1, &sspmv_csr4_event);
    if (err != CL_SUCCESS) {
      printf("clWaitForEvents␣failed!\n");
    }
  }

}
/* If beta = 0 */
else {
  /* If alpha = 1 as well, no need to
   * consider them both */
  if (alpha == 1) {
    // Set workgroup sizes
    local[0] = CUKR_SSPMV_CSR4_THREADS; //512;
    global[0] = CUKR_SSPMV_CSR4_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 6; inc++) {
      err = clSetKernelArg(
          ComputeKernel_sspmv_csr4_a1_b0, inc, sizes[
          inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_sspmv_csr4_a1_b0, 1, NULL, global
        , local, 0, NULL, &sspmv_csr4_a1_b0_event);
    if (err != CL_SUCCESS)
    {
```

```
        printf("clExecuteKernel␣sspmv_csr4_a1_b0␣failed␣
            %d\n", nz);
      }

      // Synchronize for timing
      err = clWaitForEvents(1, &sspmv_csr4_a1_b0_event);
      if (err != CL_SUCCESS) {
        printf("clWaitForEvents␣failed!\n");
      }
    }
        /* If alpha != 1, have to consider it */
        else {
          // Set workgroup sizes
      local[0] = CUKR_SSPMV_CSR4_THREADS; //512;
      global[0] = CUKR_SSPMV_CSR4_CTAS * local[0];

      // Set kernel args
      for (inc = 0; inc < 7; inc++) {
        err = clSetKernelArg(ComputeKernel_sspmv_csr4_b0
            , inc, sizes[inc], values[inc]);
        if (err == CL_INVALID_KERNEL)
        {
          printf("Failed␣%d␣failed\n", inc);
          //return -1;
        }
        if (err != CL_SUCCESS)
        {
          printf("clSetKernelArg␣%d␣failed\n", inc);
          //return -1;
        }
      }
      err = clEnqueueNDRangeKernel(ComputeCommands,
          ComputeKernel_sspmv_csr4_b0, 1, NULL, global,
          local, 0, NULL, &sspmv_csr4_b0_event);
      if (err != CL_SUCCESS)
      {
        printf("clExecuteKernel␣sspmv_csr4_b0␣failed␣%d\
            n", nz);
      }
      // Synchronize for timing
      err = clWaitForEvents(1, &sspmv_csr4_b0_event);
      if (err != CL_SUCCESS) {
        printf("clWaitForEvents␣failed!\n");
      }
        }

  }
}
```

```
/**
 * \brief Wrapper for SSPMV_HYB
 * \param rows,cols Matrix size
 * \param nz Number of nonzeros
 * \param alpha Scale factor for Ax
 * \param ell_nz_row,ell_stride ELL dimensions
 * \param csr_nz CSR dimension
 * \param ell_idx ELL column index vector
 * \param ell_val ELL value vector
 * \param csr_ptr CSR ptr vector
 * \param csr_idx CSR idx vector
 * \param csr_val CSR value vector
 * \param x Vector being multiplied
 * \param beta Scale factor for y
 * \param y Result vector
 */
void opencl_sspmv_hyb(int rows, int cols, int nz, float
   alpha, int ell_nz_row, int ell_stride, int csr_nz,
                    cl_mem* d_ell_idx, cl_mem* d_ell_val
                       , cl_mem* d_csr_ptr, cl_mem*
                       d_csr_idx, cl_mem* d_csr_val,
                    cl_mem* d_x, float beta, cl_mem* d_y
                       )
{
  #define DIVIDE_INTO(x, y) (((x) + (y) - 1)/(y))

  const unsigned int BLOCK_SIZE_ELL = 256;
  unsigned int num_blocks = DIVIDE_INTO(rows,
     BLOCK_SIZE_ELL);

  // Prepare data for the ell kernel
  size_t sizes[9];
  void *values[9];

  sizes[0] = sizeof(int);
  values[0] = (void *)&rows;
  sizes[1] = sizeof(float);
  values[1] = (void *)&alpha;
  sizes[2] = sizeof(int);
  values[2] = (void *)&ell_nz_row;
  sizes[3] = sizeof(int);
  values[3] = (void *)&ell_stride;
  sizes[4] = sizeof(cl_mem);
  values[4] = (void *)d_ell_idx;
  sizes[5] = sizeof(cl_mem);
  values[5] = (void *)d_ell_val;
  sizes[6] = sizeof(float);
  values[6] = (void *)&beta;
  sizes[7] = sizeof(cl_mem);
```

```
values[7] = (void *)d_y;
sizes[8] = sizeof(cl_mem);
values[8] = (void *)d_x;

size_t global[1];
size_t local[1];

local[0] = BLOCK_SIZE_ELL;
global[0] = num_blocks * local[0];

int inc;
// Set kernel args
for (inc = 0; inc < 9; inc++) {
  err = clSetKernelArg(ComputeKernel_sspmv_ell, inc,
      sizes[inc], values[inc]);
  if (err == CL_INVALID_KERNEL)
  {
    printf("Failed␣%d␣failed\n", inc);
  }
  if (err != CL_SUCCESS)
  {
    printf("clSetKernelArg␣%d␣failed\n", inc);
  }
}

// Launch the kernel
err = clEnqueueNDRangeKernel(ComputeCommands,
    ComputeKernel_sspmv_ell, 1, NULL, global, local, 0,
     NULL, &sspmv_hyb_event);
if (err != CL_SUCCESS)
{
  printf("clExecuteKernel␣sspmv_ell␣failed␣%d\n", nz);
}

local[0] = CUKR_SSPMV_CSR4_THREADS;
global[0] = CUKR_SSPMV_CSR4_CTAS * local[0];

if(csr_nz){
  // The rest in CSR4
  if (alpha == 1){
    sizes[0] = sizeof(int);
    values[0] = (void *)&rows;
    sizes[1] = sizeof(cl_mem);
    values[1] = (void *)d_csr_ptr;
    sizes[2] = sizeof(cl_mem);
    values[2] = (void *)d_csr_idx;
    sizes[3] = sizeof(cl_mem);
    values[3] = (void *)d_csr_val;
    sizes[4] = sizeof(cl_mem);
```

```
    values[4] = (void *)d_x;
    sizes[5] = sizeof(cl_mem);
    values[5] = (void *)d_y;

    // Set kernel args
    for (inc = 0; inc < 6; inc++) {
      err = clSetKernelArg(ComputeKernel_sspmv_csr4_a1
          , inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
       ComputeKernel_sspmv_csr4_a1, 1, NULL, global,
       local, 0, NULL, &sspmv_hyb_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣sspmv_csr4_a1␣failed␣%d\
         n", nz);
    }
  }
  else {
    sizes[0] = sizeof(int);
    values[0] = (void *)&rows;
    sizes[1] = sizeof(cl_mem);
    values[1] = (void *)d_csr_ptr;
    sizes[2] = sizeof(cl_mem);
    values[2] = (void *)d_csr_idx;
    sizes[3] = sizeof(cl_mem);
    values[3] = (void *)d_csr_val;
    sizes[4] = sizeof(cl_mem);
    values[4] = (void *)d_x;
    sizes[5] = sizeof(cl_mem);
    values[5] = (void *)d_y;
    sizes[6] = sizeof(float);
    values[6] = (void *)&alpha;

    // Set kernel args
    for (inc = 0; inc < 7; inc++) {
      err = clSetKernelArg(ComputeKernel_sspmv_csr4,
          inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
```

```
        }
        if (err != CL_SUCCESS)
        {
          printf("clSetKernelArg␣%d␣failed\n", inc);
        }
      }
      err = clEnqueueNDRangeKernel(ComputeCommands,
          ComputeKernel_sspmv_csr4, 1, NULL, global,
          local, 0, NULL, &sspmv_hyb_event);
      if (err != CL_SUCCESS)
      {
        printf("clExecuteKernel␣sspmv_csr4␣failed␣%d\n",
            nz);
      }
    }
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &sspmv_hyb_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }
}

void opencl_sspmv_bcsr(int rows, int cols, int nz, int r
    , int c, float alpha, cl_mem* ptr,
             cl_mem* idx, cl_mem* val, cl_mem* x, float
                beta,
             cl_mem* y)
{
  printf("test␣opencl␣sspmv␣bcsr\n");
}

void opencl_sspmv_bcsr4(int rows, int cols, int nz, int
    r, int c, float alpha, cl_mem* ptr,
           cl_mem* idx, cl_mem* val, cl_mem* x, float
              beta,
           cl_mem* y)
{
  printf("test␣opencl␣sspmv␣bcsr4\n");
}
```

E.26 Kernels GPU single set-up, header

```
/*
 *  kernels_single.h
 *
 *
 *  Created by Olav Aanes Fagerlund.
```

```
 *
 */

#include "../../../../init/init_opencl.h"

float opencl_kernel_sdot(int n, cl_mem* x, cl_mem* y);

void opencl_kernel_saxpy(int n, float a, cl_mem *cl_d_x,
     cl_mem *cl_d_y);

void opencl_kernel_saypx(int n, float a, cl_mem *cl_d_x,
     cl_mem *cl_d_y);

void opencl_scopy(int n, cl_mem* x, int incx, cl_mem* y,
     int incy);

void opencl_kernel_sscal(int n, float a, cl_mem* x);

void opencl_sspmv_bcsr(int rows, int cols, int nz, int r
    , int c, float alpha, cl_mem* ptr,
             cl_mem* idx, cl_mem* val, cl_mem* x, float
                 beta, cl_mem* y);

void opencl_sspmv_bcsr4(int rows, int cols, int nz, int
    r, int c, float alpha, cl_mem* ptr,
             cl_mem* idx, cl_mem* val, cl_mem* x, float
                 beta, cl_mem* y);

void opencl_sspmv_csr(int rows, int cols, int nz, float
    alpha, cl_mem* d_ptr,
             cl_mem* d_idx, cl_mem* d_val, cl_mem* d_x,
                 float beta, cl_mem* d_y);

void opencl_sspmv_csr4(int rows, int cols, int nz, float
     alpha, cl_mem* d_ptr,
             cl_mem* d_idx, cl_mem* d_val, cl_mem* d_x,
                 float beta, cl_mem* d_y);

void opencl_sspmv_hyb(int rows, int cols, int nz, float
    alpha, int ell_nz_row, int ell_stride, int csr_nz,
             cl_mem* d_ell_idx, cl_mem* d_ell_val, cl_mem
                 * d_csr_ptr, cl_mem* d_csr_idx, cl_mem*
                 d_csr_val,
             cl_mem* d_x, float beta, cl_mem* d_y);
```

E.27 Kernels GPU single-double (quasi-double) setup

```
/*
```

```
 *  kernels_single.c
 *
 *
 *  Created by Olav Aanes Fagerlund.
 *
 */

#define LOGGER_NAME "CUKr.blas.impl.gpu.opencl.
   kernels_qdouble"
#include "kernels_qdouble.h"

#define AUTO_LOCAL 0
#define ORIGINAL_VALUES 0

#if ORIGINAL_VALUES
// Work-group sizes and the number of those equal to the
     CUDA version
#include "kernels_config_orig.h"
#else
// Work-group sizes and the number of those giving
   better OpenCL performance
#include "kernels_config.h"
#endif

double opencl_kernel_qdot(int n, cl_mem* xh, cl_mem* xt,
    cl_mem* yh, cl_mem* yt)
{
  size_t global[1];
  size_t local[1];

  local[0] = CUKR_QDOT_THREADS;
  global[0] = CUKR_QDOT_CTAS * local[0];

  size_t sizes[9];
  void *values[9];

  double res = 0;

  // Setup work-group level result
  cl_mem res_bufh = clCreateBuffer(ComputeContext,
     CL_MEM_WRITE_ONLY, CUKR_QDOT_CTAS * sizeof(cl_float
     ), NULL, &err);
  if (err != CL_SUCCESS) {
    printf("Setting␣up␣res_bufh␣for␣qdot␣failed!\n");
  }
  cl_mem res_buft = clCreateBuffer(ComputeContext,
     CL_MEM_WRITE_ONLY, CUKR_QDOT_CTAS * sizeof(cl_float
     ), NULL, &err);
  if (err != CL_SUCCESS) {
```

```
    printf("Setting␣up␣res_buft␣for␣qdot␣failed!\n");
  }

  // Host memory to collect results from work-groups
  float *h_sh, *h_st;
  if ( (h_sh = (float*)malloc(CUKR_QDOT_CTAS*sizeof(h_sh
      [0]))) == NULL) {
    printf("FATAL:␣Error␣allocating␣memory␣for␣QDOT␣
        result␣vectors");
    exit(1);
  }
  if ( (h_st = (float*)malloc(CUKR_QDOT_CTAS*sizeof(h_st
      [0]))) == NULL) {
    printf("FATAL:␣Error␣allocating␣memory␣for␣QDOT␣
        result␣vectors");
    exit(1);
  }

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(cl_mem);
  values[1] = (void *)xh;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)xt;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)yh;
  sizes[4] = sizeof(cl_mem);
  values[4] = (void *)yt;
  sizes[5] = sizeof(cl_mem);
  values[5] = (void *)&res_bufh;
  sizes[6] = sizeof(cl_mem);
  values[6] = (void *)&res_buft;
  sizes[7] =  local[0] * sizeof(cl_float);
  values[7] = NULL;
  sizes[8] =  local[0] * sizeof(cl_float);
  values[8] = NULL;

  int x_inc;
  for (x_inc = 0; x_inc<9; x_inc++) {
    err = clSetKernelArg(ComputeKernel_qdot, x_inc,
        sizes[x_inc], values[x_inc]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed␣%d␣arg␣qdot\n", x_inc);
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", x_inc);
```

```
    //return -1;
  }
}

// Enqueue kernel for execution
err = clEnqueueNDRangeKernel(ComputeCommands,
    ComputeKernel_qdot, 1, NULL, global, local, 0, NULL
    , &qdot_event);
if (err != CL_SUCCESS)
{
  printf("clExecuteKernel failed %d\n",n);
}

// Synchronize for timing
err = clWaitForEvents(1, &qdot_event);
if (err != CL_SUCCESS) {
  printf("clWaitForEvents failed!\n");
}

// Get result from work-group level
err = clEnqueueReadBuffer(ComputeCommands, res_bufh,
    CL_TRUE, 0, CUKR_QDOT_CTAS * sizeof(cl_float), (
    void*)h_sh, 0, NULL, NULL);
if (err != CL_SUCCESS) {
  printf("readback of res_buf in qdot failed!\n");
}
err = clEnqueueReadBuffer(ComputeCommands, res_buft,
    CL_TRUE, 0, CUKR_QDOT_CTAS * sizeof(cl_float), (
    void*)h_st, 0, NULL, NULL);
if (err != CL_SUCCESS) {
  printf("readback of res_buf in qdot failed!\n");
}

// Releas memory objects
err = clReleaseMemObject(res_bufh);
if (err != CL_SUCCESS) {
  printf("release of cl memobject res_bufh in qdot 
      failed!\n");
}
err = clReleaseMemObject(res_buft);
if (err != CL_SUCCESS) {
  printf("release of cl memobject res_buft in qdot 
      failed!\n");
}

// Do the final sum
  double sum0 = 0, sum1 = 0;
int i;
  for (i = 0; i < CUKR_QDOT_CTAS; i++) {
```

```
    sum0 += h_sh[i];
      sum1 += h_st[i];
    }

  // Free allocated memory
  free(h_sh);
    free(h_st);

    return sum0 + sum1;
}

void opencl_kernel_qaxpy(int n, float a0, float a1,
    cl_mem* cl_xh, cl_mem* cl_xt, cl_mem* cl_yh, cl_mem*
    cl_yt)
{
  size_t sizes[11];
  void *values[11];

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(float);
  values[1] = (void *)&a0;
  sizes[2] = sizeof(float);
  values[2] = (void *)&a1;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)cl_xh;
  sizes[4] = sizeof(cl_mem);
  values[4] = (void *)cl_xt;
  sizes[5] = sizeof(cl_mem);
  values[5] = (void *)cl_yh;
  sizes[6] = sizeof(cl_mem);
  values[6] = (void *)cl_yt;

  sizes[7] = CUKR_QAXPY_THREADS * sizeof(cl_float);
  values[7] = NULL;
  sizes[8] = CUKR_QAXPY_THREADS * sizeof(cl_float);
  values[8] = NULL;
  sizes[9] = CUKR_QAXPY_THREADS * sizeof(cl_float);
  values[9] = NULL;
  sizes[10] = CUKR_QAXPY_THREADS * sizeof(cl_float);
  values[10] = NULL;

  /* Kernel invocation */
  int x;
  for (x = 0; x<11; x++) {
    err = clSetKernelArg(ComputeKernel_qaxpy, x, sizes[x
       ], values[x]);
    if (err == CL_INVALID_KERNEL)
    {
```

```
      printf("Failed␣%d␣failed\n", x);
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", x);
      //return -1;
    }
  }

  size_t global[1];
  size_t local[1];

  local[0] = CUKR_QAXPY_THREADS;
  global[0] = CUKR_QAXPY_CTAS * local[0];

  /* Enqueue kernel for execution */
  err = clEnqueueNDRangeKernel(ComputeCommands,
     ComputeKernel_qaxpy, 1, NULL, global, local, 0,
     NULL, &qaxpy_event);
  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣failed␣%d\n",n);
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &qaxpy_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }

}

void opencl_kernel_qaypx(int n, float a0, float a1,
    cl_mem* cl_xh, cl_mem* cl_xt, cl_mem* cl_yh, cl_mem*
    cl_yt)
{
  size_t sizes[11];
  void *values[11];

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(float);
  values[1] = (void *)&a0;
  sizes[2] = sizeof(float);
  values[2] = (void *)&a1;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)cl_xh;
  sizes[4] = sizeof(cl_mem);
```

```
values[4] = (void *)cl_xt;
sizes[5] = sizeof(cl_mem);
values[5] = (void *)cl_yh;
sizes[6] = sizeof(cl_mem);
values[6] = (void *)cl_yt;

sizes[7] = CUKR_QAYPX_THREADS * sizeof(cl_float);
values[7] = NULL;
sizes[8] = CUKR_QAYPX_THREADS * sizeof(cl_float);
values[8] = NULL;
sizes[9] = CUKR_QAYPX_THREADS * sizeof(cl_float);
values[9] = NULL;
sizes[10] = CUKR_QAYPX_THREADS * sizeof(cl_float);
values[10] = NULL;

/* Kernel invocation */
int x;
for (x = 0; x<11; x++) {
  err = clSetKernelArg(ComputeKernel_qaypx, x, sizes[x
     ], values[x]);
  if (err == CL_INVALID_KERNEL)
  {
    printf("Failed␣%d␣failed\n", x);
    //return -1;
  }
  if (err != CL_SUCCESS)
  {
    printf("clSetKernelArg␣%d␣failed\n", x);
    //return -1;
  }
}

size_t global[1];
size_t local[1];

local[0] = CUKR_QAYPX_THREADS;
global[0] = CUKR_QAYPX_CTAS * local[0];

/* Enqueue kernel for execution */
err = clEnqueueNDRangeKernel(ComputeCommands,
   ComputeKernel_qaypx, 1, NULL, global, local, 0,
   NULL, &qaypx_event);
if (err != CL_SUCCESS)
{
  printf("clExecuteKernel␣failed␣%d\n",n);
}

// Synchronize for timing
err = clWaitForEvents(1, &qaypx_event);
```

```
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }

}

void opencl_kernel_qscal(int n, float a0, float a1,
    cl_mem* xh, cl_mem* xt){

  size_t sizes[7];
  void *values[7];

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(float);
  values[1] = (void *)&a0;
  sizes[2] = sizeof(float);
  values[2] = (void *)&a1;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)xh;
  sizes[4] = sizeof(cl_mem);
  values[4] = (void *)xt;
  sizes[5] = CUKR_QSCAL_THREADS * sizeof(cl_float);
  values[5] = NULL;
  sizes[6] = CUKR_QSCAL_THREADS * sizeof(cl_float);
  values[6] = NULL;

  /* Kernel invocation */
  int i;
  for (i = 0; i<7; i++) {
    err = clSetKernelArg(ComputeKernel_qscal, i, sizes[i
        ], values[i]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed␣%d␣failed\n", i);
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", i);
      //return -1;
    }
  }

  size_t global[1];
  size_t local[1];

  local[0] = CUKR_QSCAL_THREADS;
  global[0] = CUKR_QSCAL_CTAS * local[0];
```

```
  /* Enqueue kernel for execution */
  err = clEnqueueNDRangeKernel(ComputeCommands,
     ComputeKernel_qscal, 1, NULL, global, local, 0,
     NULL, &qscal_event);
  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣qscal␣failed␣%d\n",n);
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &qscal_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }
}


void opencl_qspmv_csr(int rows, int cols, int nz, double
    a0, double a1, cl_mem* ptr,
          cl_mem* idx, cl_mem* valh, cl_mem* valt,
          cl_mem* xh, cl_mem* xt, double b0, double b1,
          cl_mem* yh, cl_mem* yt)
{

  size_t sizes[11];
  void *values[11];

  sizes[0] = sizeof(int);
  values[0] = (void *)&rows;

  sizes[1] = sizeof(cl_mem);
  values[1] = (void *)ptr;

  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)idx;

  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)valh;

  sizes[4] = sizeof(cl_mem);
  values[4] = (void *)valt;

  sizes[5] = sizeof(cl_mem);
  values[5] = (void *)xh;

  sizes[6] = sizeof(cl_mem);
  values[6] = (void *)xt;
```

```
sizes[7] = sizeof(cl_mem);
values[7] = (void *)yh;

sizes[8] = sizeof(cl_mem);
values[8] = (void *)yt;

sizes[9] = sizeof(float);
values[9] = (void *)&a0;

sizes[10] = sizeof(float);
values[10] = (void *)&a1;

size_t global[1];
size_t local[1];

int inc;

/*
 * Chose which kernel to run
 */

double a = a0 + a1;
double b = b0 + b1;

/* If beta != 0 */
if (b != 0)
{
  /* If beta != 1, do a scaling first */
  if (b != 1)
    opencl_kernel_qscal(rows, b0, b1, yh, yt);

  /* If alpha = 1, no need to consider */
    if (a == 1) {
    // Set workgroup sizes
    local[0] = CUKR_QSPMV_CSR_A1_THREADS; //512;
    global[0] = CUKR_QSPMV_CSR_A1_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 9; inc++) {
      err = clSetKernelArg(ComputeKernel_qspmv_csr_a1,
          inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
```

```
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_qspmv_csr_a1, 1, NULL, global,
        local, 0, NULL, &qspmv_csr_a1_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣qspmv_csr_a1␣failed␣%d\n
         ", nz);
    }
    // Synchronize for timing
    err = clWaitForEvents(1, &qspmv_csr_a1_event);
    if (err != CL_SUCCESS) {
      printf("clWaitForEvents␣failed!\n");
    }
    }
  /* Else, take the most general case */
  else {
    // Set workgroup sizes
    local[0] = CUKR_QSPMV_CSR_THREADS; //512;
    global[0] = CUKR_QSPMV_CSR_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 11; inc++) {
      err = clSetKernelArg(ComputeKernel_qspmv_csr,
         inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_qspmv_csr, 1, NULL, global, local
        , 0, NULL, &qspmv_csr_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣qspmv_csr␣failed␣%d\n",
         nz);
    }
    // Synchronize for timing
    err = clWaitForEvents(1, &qspmv_csr_event);
    if (err != CL_SUCCESS) {
```

```
      printf("clWaitForEvents␣failed!\n");
    }
  }

}
/* If beta = 0 */
else {
  /* If alpha = 1 as well, no need to
   * consider them both */
  if (a == 1) {
    // Set workgroup sizes
    local[0] = CUKR_QSPMV_CSR_THREADS; //512;
    global[0] = CUKR_QSPMV_CSR_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 9; inc++) {
      err = clSetKernelArg(
          ComputeKernel_qspmv_csr_a1_b0, inc, sizes[inc
          ], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_qspmv_csr_a1_b0, 1, NULL, global,
         local, 0, NULL, &qspmv_csr_a1_b0_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣qspmv_csr_a1_b0␣failed␣%
          d\n", nz);
    }
    // Synchronize for timing
    err = clWaitForEvents(1, &qspmv_csr_a1_b0_event);
    if (err != CL_SUCCESS) {
      printf("clWaitForEvents␣failed!\n");
    }
  }
      /* If alpha != 1, have to consider it */
      else {
        // Set workgroup sizes
    local[0] = CUKR_QSPMV_CSR_THREADS; //512;
    global[0] = CUKR_QSPMV_CSR_CTAS * local[0];
```

```
      // Set kernel args
      for (inc = 0; inc < 11; inc++) {
        err = clSetKernelArg(ComputeKernel_qspmv_csr_b0,
            inc, sizes[inc], values[inc]);
        if (err == CL_INVALID_KERNEL)
        {
          printf("Failed␣%d␣failed\n", inc);
          //return -1;
        }
        if (err != CL_SUCCESS)
        {
          printf("clSetKernelArg␣%d␣failed\n", inc);
          //return -1;
        }
      }
      err = clEnqueueNDRangeKernel(ComputeCommands,
          ComputeKernel_qspmv_csr_b0, 1, NULL, global,
          local, 0, NULL, &qspmv_csr_b0_event);
      if (err != CL_SUCCESS)
      {
        printf("clExecuteKernel␣qspmv_csr_b0␣failed␣%d\n
            ", nz);
      }
      // Synchronize for timing
      err = clWaitForEvents(1, &qspmv_csr_b0_event);
      if (err != CL_SUCCESS) {
        printf("clWaitForEvents␣failed!\n");
      }
        }

  }
}

void opencl_qspmv_csr4(int rows, int cols, int nz,
    double a0, double a1, cl_mem* ptr,
             cl_mem* idx, cl_mem* valh, cl_mem* valt,
             cl_mem* xh, cl_mem* xt, double b0, double
                 b1,
             cl_mem* yh, cl_mem* yt)
{
  size_t sizes[11];
  void *values[11];

  sizes[0] = sizeof(int);
  values[0] = (void *)&rows;

  sizes[1] = sizeof(cl_mem);
  values[1] = (void *)ptr;
```

```
sizes[2] = sizeof(cl_mem);
values[2] = (void *)idx;

sizes[3] = sizeof(cl_mem);
values[3] = (void *)valh;

sizes[4] = sizeof(cl_mem);
values[4] = (void *)valt;

sizes[5] = sizeof(cl_mem);
values[5] = (void *)xh;

sizes[6] = sizeof(cl_mem);
values[6] = (void *)xt;

sizes[7] = sizeof(cl_mem);
values[7] = (void *)yh;

sizes[8] = sizeof(cl_mem);
values[8] = (void *)yt;

sizes[9] = sizeof(float);
values[9] = (void *)&a0;

sizes[10] = sizeof(float);
values[10] = (void *)&a1;

size_t global[1];
size_t local[1];

int inc;

/*
 * Chose which kernel to run
 */

double a = a0 + a1;
double b = b0 + b1;

/* If beta != 0 */
if (b != 0)
{
  /* If beta != 1, do a scaling first */
  if (b != 1)
    opencl_kernel_qscal(rows, b0, b1, yh, yt);

  /* If alpha = 1, no need to consider */
    if (a == 1) {
```

```
    // Set workgroup sizes
    local[0] = CUKR_QSPMV_CSR4_THREADS; //512;
    global[0] = CUKR_QSPMV_CSR4_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 9; inc++) {
      err = clSetKernelArg(ComputeKernel_qspmv_csr4_a1
          , inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_qspmv_csr4_a1, 1, NULL, global,
        local, 0, NULL, &qspmv_csr4_a1_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣qspmv_csr_a1␣failed␣%d\n
          ", nz);
    }
    // Synchronize for timing
    err = clWaitForEvents(1, &qspmv_csr4_a1_event);
    if (err != CL_SUCCESS) {
      printf("clWaitForEvents␣failed!\n");
    }
    }
  /* Else, take the most general case */
  else {
    // Set workgroup sizes
    local[0] = CUKR_QSPMV_CSR4_THREADS; //512;
    global[0] = CUKR_QSPMV_CSR4_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 11; inc++) {
      err = clSetKernelArg(ComputeKernel_qspmv_csr4,
          inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
```

```
          {
            printf("clSetKernelArg␣%d␣failed\n", inc);
            //return -1;
          }
        }
        err = clEnqueueNDRangeKernel(ComputeCommands,
            ComputeKernel_qspmv_csr4, 1, NULL, global,
            local, 0, NULL, &qspmv_csr4_event);
        if (err != CL_SUCCESS)
        {
          printf("clExecuteKernel␣qspmv_csr␣failed␣%d\n",
              nz);
        }
        // Synchronize for timing
        err = clWaitForEvents(1, &qspmv_csr4_event);
        if (err != CL_SUCCESS) {
          printf("clWaitForEvents␣failed!\n");
        }
      }

    }
    /* If beta = 0 */
    else {
      /* If alpha = 1 as well, no need to
       * consider them both */
      if (a == 1) {
        // Set workgroup sizes
        local[0] = CUKR_QSPMV_CSR4_THREADS; //512;
        global[0] = CUKR_QSPMV_CSR4_CTAS * local[0];

        // Set kernel args
        for (inc = 0; inc < 9; inc++) {
          err = clSetKernelArg(
              ComputeKernel_qspmv_csr4_a1_b0, inc, sizes[
              inc], values[inc]);
          if (err == CL_INVALID_KERNEL)
          {
            printf("Failed␣%d␣failed\n", inc);
            //return -1;
          }
          if (err != CL_SUCCESS)
          {
            printf("clSetKernelArg␣%d␣failed\n", inc);
            //return -1;
          }
        }
        err = clEnqueueNDRangeKernel(ComputeCommands,
            ComputeKernel_qspmv_csr4_a1_b0, 1, NULL, global
            , local, 0, NULL, &qspmv_csr4_a1_b0_event);
```

```
      if (err != CL_SUCCESS)
      {
        printf("clExecuteKernel␣qspmv_csr_a1_b0␣failed␣%
            d\n", nz);
      }
      // Synchronize for timing
      err = clWaitForEvents(1, &qspmv_csr4_a1_b0_event);
      if (err != CL_SUCCESS) {
        printf("clWaitForEvents␣failed!\n");
      }
    }
        /* If alpha != 1, have to consider it */
        else {
          // Set workgroup sizes
      local[0] = CUKR_QSPMV_CSR4_THREADS; //512;
      global[0] = CUKR_QSPMV_CSR4_CTAS * local[0];

      // Set kernel args
      for (inc = 0; inc < 11; inc++) {
        err = clSetKernelArg(ComputeKernel_qspmv_csr4_b0
            , inc, sizes[inc], values[inc]);
        if (err == CL_INVALID_KERNEL)
        {
          printf("Failed␣%d␣failed\n", inc);
          //return -1;
        }
        if (err != CL_SUCCESS)
        {
          printf("clSetKernelArg␣%d␣failed\n", inc);
          //return -1;
        }
      }
      err = clEnqueueNDRangeKernel(ComputeCommands,
          ComputeKernel_qspmv_csr4_b0, 1, NULL, global,
          local, 0, NULL, &qspmv_csr4_b0_event);
      if (err != CL_SUCCESS)
      {
        printf("clExecuteKernel␣qspmv_csr_b0␣failed␣%d\n
            ", nz);
      }
      // Synchronize for timing
      err = clWaitForEvents(1, &qspmv_csr4_b0_event);
      if (err != CL_SUCCESS) {
        printf("clWaitForEvents␣failed!\n");
      }
        }

  }
}
```

```
void opencl_qspmv_hyb(int rows, int cols, int nz, float
    alpha0, float alpha1, int ell_nz_row, int ell_stride,
     int csr_nz,
                  cl_mem* d_ell_idx, cl_mem* d_ell_valh,
                      cl_mem* d_ell_valt, cl_mem*
                     d_csr_ptr, cl_mem* d_csr_idx,
                     cl_mem* d_csr_valh, cl_mem*
                     d_csr_valt,
                  cl_mem* d_xh, cl_mem* d_xt, float
                     beta0, float beta1, cl_mem* d_yh,
                     cl_mem* d_yt)
{
  #define DIVIDE_INTO(x, y) (((x) + (y) - 1)/(y))

  const unsigned int BLOCK_SIZE_ELL = 256;
  unsigned int num_blocks = DIVIDE_INTO(rows,
     BLOCK_SIZE_ELL);

  // Prepare data for the ell kernel
  size_t sizes[14];
  void *values[14];

  sizes[0] = sizeof(int);
  values[0] = (void *)&rows;
  sizes[1] = sizeof(float);
  values[1] = (void *)&alpha0;
  sizes[2] = sizeof(float);
  values[2] = (void *)&alpha1;
  sizes[3] = sizeof(int);
  values[3] = (void *)&ell_nz_row;
  sizes[4] = sizeof(int);
  values[4] = (void *)&ell_stride;
  sizes[5] = sizeof(cl_mem);
  values[5] = (void *)d_ell_idx;
  sizes[6] = sizeof(cl_mem);
  values[6] = (void *)d_ell_valh;
  sizes[7] = sizeof(cl_mem);
  values[7] = (void *)d_ell_valt;
  sizes[8] = sizeof(float);
  values[8] = (void *)&beta0;
  sizes[9] = sizeof(float);
  values[9] = (void *)&beta1;
  sizes[10] = sizeof(cl_mem);
  values[10] = (void *)d_yh;
  sizes[11] = sizeof(cl_mem);
  values[11] = (void *)d_yt;
  sizes[12] = sizeof(cl_mem);
  values[12] = (void *)d_xh;
```

```
sizes[13] = sizeof(cl_mem);
values[13] = (void *)d_xt;

size_t global[1];
size_t local[1];

local[0] = BLOCK_SIZE_ELL;
global[0] = num_blocks * local[0];

int inc;
// Set kernel args
for (inc = 0; inc < 14; inc++) {
  err = clSetKernelArg(ComputeKernel_qspmv_ell, inc,
      sizes[inc], values[inc]);
  if (err == CL_INVALID_KERNEL)
  {
    printf("Failed␣%d␣failed\n", inc);
  }
  if (err != CL_SUCCESS)
  {
    printf("clSetKernelArg␣%d␣failed\n", inc);
  }
}

// Launch the kernel
err = clEnqueueNDRangeKernel(ComputeCommands,
    ComputeKernel_qspmv_ell, 1, NULL, global, local, 0,
     NULL, &qspmv_hyb_event);
if (err != CL_SUCCESS)
{
  printf("clExecuteKernel␣qspmv_ell␣failed␣%d\n", nz);
}

local[0] = CUKR_SSPMV_CSR4_THREADS;
global[0] = CUKR_SSPMV_CSR4_CTAS * local[0];

if(csr_nz){
  // The rest in CSR4
  double alpha = alpha0 + alpha1;
  if (alpha == 1){
    sizes[0] = sizeof(int);
    values[0] = (void *)&rows;
    sizes[1] = sizeof(cl_mem);
    values[1] = (void *)d_csr_ptr;
    sizes[2] = sizeof(cl_mem);
    values[2] = (void *)d_csr_idx;
    sizes[3] = sizeof(cl_mem);
    values[3] = (void *)d_csr_valh;
    sizes[4] = sizeof(cl_mem);
```

```
    values[4] = (void *)d_csr_valt;
    sizes[5] = sizeof(cl_mem);
    values[5] = (void *)d_xh;
    sizes[6] = sizeof(cl_mem);
    values[6] = (void *)d_xt;
    sizes[7] = sizeof(cl_mem);
    values[7] = (void *)d_yh;
    sizes[8] = sizeof(cl_mem);
    values[8] = (void *)d_yt;

    // Set kernel args
    for (inc = 0; inc < 9; inc++) {
      err = clSetKernelArg(ComputeKernel_qspmv_csr4_a1
          , inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_qspmv_csr4_a1, 1, NULL, global,
        local, 0, NULL, &qspmv_hyb_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣qspmv_csr4_a1␣failed␣%d\
          n", nz);
    }
  }
  else {
    sizes[0] = sizeof(int);
    values[0] = (void *)&rows;
    sizes[1] = sizeof(cl_mem);
    values[1] = (void *)d_csr_ptr;
    sizes[2] = sizeof(cl_mem);
    values[2] = (void *)d_csr_idx;
    sizes[3] = sizeof(cl_mem);
    values[3] = (void *)d_csr_valh;
    sizes[4] = sizeof(cl_mem);
    values[4] = (void *)d_csr_valt;
    sizes[5] = sizeof(cl_mem);
    values[5] = (void *)d_xh;
    sizes[6] = sizeof(cl_mem);
    values[6] = (void *)d_xt;
    sizes[7] = sizeof(cl_mem);
    values[7] = (void *)d_yh;
```

```
      sizes[8] = sizeof(cl_mem);
      values[8] = (void *)d_yt;
      sizes[9] = sizeof(float);
      values[9] = (void *)&alpha0;
      sizes[10] = sizeof(float);
      values[10] = (void *)&alpha1;

      // Set kernel args
      for (inc = 0; inc < 11; inc++) {
        err = clSetKernelArg(ComputeKernel_qspmv_csr4,
            inc, sizes[inc], values[inc]);
        if (err == CL_INVALID_KERNEL)
        {
          printf("Failed␣%d␣failed\n", inc);
        }
        if (err != CL_SUCCESS)
        {
          printf("clSetKernelArg␣%d␣failed\n", inc);
        }
      }
      err = clEnqueueNDRangeKernel(ComputeCommands,
          ComputeKernel_qspmv_csr4, 1, NULL, global,
          local, 0, NULL, &qspmv_hyb_event);
      if (err != CL_SUCCESS)
      {
        printf("clExecuteKernel␣qspmv_csr4␣failed␣%d\n",
            nz);
      }
    }
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &qspmv_hyb_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }
}


void opencl_qspmv_bcsr(int rows, int cols, int nz, int r
    , int c, double a0, double a1, cl_mem* ptr,
             cl_mem* idx, cl_mem* valh, cl_mem* valt,
                 cl_mem* xh, cl_mem* xt, double b0,
                 double b1,
             cl_mem* yh, cl_mem* yt)
{
  printf("test␣opencl_qspmv_bcsr\n");
}
```

```
void opencl_qspmv_bcsr4(int rows, int cols, int nz, int
   r, int c, double a0, double a1, cl_mem* ptr,
            cl_mem* idx, cl_mem* valh, cl_mem* valt,
            cl_mem* xh, cl_mem* xt, double b0, double b1
               ,
            cl_mem* yh, cl_mem* yt)
{
  printf("test␣opencl_qspmv_bcsr4\n");
}
```

E.28 Kernels GPU single-double (quasi-double) setup, header

```
/*
 *  kernels_single.h
 *
 *
 *  Created by Olav Aanes Fagerlund.
 *
 */

#include "../../../../init/init_opencl.h"

double opencl_kernel_qdot(int n, cl_mem* xh, cl_mem* xt,
    cl_mem* yh, cl_mem* yt);

void opencl_kernel_qaxpy(int n, float a0, float a1,
   cl_mem* cl_xh, cl_mem* cl_xt, cl_mem* cl_yh, cl_mem*
   cl_yt);

void opencl_kernel_qaypx(int n, float a0, float a1,
   cl_mem* cl_xh, cl_mem* cl_xt, cl_mem* cl_yh, cl_mem*
   cl_yt);

void opencl_kernel_qscal(int n, float a0, float a1,
   cl_mem* xh, cl_mem* xt);

void opencl_qspmv_bcsr(int rows, int cols, int nz, int r
   , int c, double a0, double a1, cl_mem* ptr,
             cl_mem* idx, cl_mem* valh, cl_mem* valt,
                cl_mem* xh, cl_mem* xt, double b0,
                double b1,
             cl_mem* yh, cl_mem* yt);

void opencl_qspmv_bcsr4(int rows, int cols, int nz, int
   r, int c, double a0, double a1, cl_mem* ptr,
            cl_mem* idx, cl_mem* valh, cl_mem* valt,
            cl_mem* xh, cl_mem* xt, double b0, double b1
               ,
```

```
            cl_mem* yh, cl_mem* yt);

void opencl_qspmv_csr(int rows, int cols, int nz, double
    a0, double a1, cl_mem* ptr,
            cl_mem* idx, cl_mem* valh, cl_mem* valt,
            cl_mem* xh, cl_mem* xt, double b0, double b1
               ,
            cl_mem* yh, cl_mem* yt);

void opencl_qspmv_csr4(int rows, int cols, int nz,
   double a0, double a1, cl_mem* ptr,
             cl_mem* idx, cl_mem* valh, cl_mem* valt,
             cl_mem* xh, cl_mem* xt, double b0, double
                b1,
             cl_mem* yh, cl_mem* yt);

void opencl_qspmv_hyb(int rows, int cols, int nz, float
   alpha0, float alpha1, int ell_nz_row, int ell_stride,
    int csr_nz,
            cl_mem* d_ell_idx, cl_mem* d_ell_valh,
               cl_mem* d_ell_valt, cl_mem* d_csr_ptr,
               cl_mem* d_csr_idx, cl_mem* d_csr_valh,
               cl_mem* d_csr_valt,
            cl_mem* d_xh, cl_mem* d_xt, float beta0,
               float beta1, cl_mem* d_yh, cl_mem* d_yt);
```

E.29 Kernels GPU double set-up

```
/*
 *  kernels_single.c
 *
 *
 *  Created by Olav Aanes Fagerlund.
 *
 */

#define LOGGER_NAME "CUKr.blas.impl.gpu.opencl.
   kernels_double"
#include "kernels_double.h"

#define AUTO_LOCAL 0
#define ORIGINAL_VALUES 0

#if ORIGINAL_VALUES
// Work-group sizes and the number of those equal to the
    CUDA version
#include "kernels_config_orig.h"
#else
```

```
// Work-group sizes and the number of those giving
    better OpenCL performance
#include "kernels_config.h"
#endif

double opencl_kernel_ddot(int n, cl_mem* x, cl_mem* y)
{
  size_t sizes[5];
  void *values[5];

  double res = 0.0;
  double* res_arr = malloc(CUKR_DDOT_CTAS * sizeof(
      double));
  cl_mem res_buf = clCreateBuffer(ComputeContext,
      CL_MEM_WRITE_ONLY, CUKR_DDOT_CTAS * sizeof(
      cl_double), /*(void*)res*/ NULL, &err);
  if (err != CL_SUCCESS) {
    printf("Setting␣up␣res_buf␣for␣sdot␣failed!\n");
  }

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(cl_mem);
  values[1] = (void *)x;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)y;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)&res_buf;
  sizes[4] =  CUKR_DDOT_THREADS * sizeof(cl_double);
  values[4] = NULL;

  int x_inc;
  for (x_inc = 0; x_inc<5; x_inc++) {
    err = clSetKernelArg(ComputeKernel_ddot, x_inc,
        sizes[x_inc], values[x_inc]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed␣%d␣arg␣sdot\n", x_inc);
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", x_inc);
      //return -1;
    }
  }

  size_t global[1];
  size_t local[1];
```

```
  local[0]  = CUKR_DDOT_THREADS;
  global[0] = CUKR_DDOT_CTAS * local[0];

  /* Enqueue kernel for execution */
  err = clEnqueueNDRangeKernel(ComputeCommands,
     ComputeKernel_ddot, 1, NULL, global, local, 0, NULL
     , &ddot_event);
  if (err != CL_SUCCESS)
   {
     printf("clExecuteKernel␣failed␣%d\n", n);
   }

  // Synchronize for timing
  err = clWaitForEvents(1, &ddot_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }

  // Get result
  err = clEnqueueReadBuffer(ComputeCommands, res_buf,
     CL_TRUE, 0, CUKR_DDOT_CTAS * sizeof(cl_double), (
     void*)res_arr, 0, NULL, NULL);
  if (err == CL_INVALID_COMMAND_QUEUE) {
    printf("YES!!!\n");
  }
  else if (err != CL_SUCCESS) {
    printf("readback␣of␣res_buf␣in␣sdot␣failed!␣err-code
       :␣%d\n", err);
  }

  err = clReleaseMemObject(res_buf);
  if (err != CL_SUCCESS) {
    printf("release␣of␣cl␣memobject␣res_buf␣in␣sdot␣
       failed!\n");
  }

  int inc;
  for (inc = 0; inc < CUKR_DDOT_CTAS; inc++) {
    res += res_arr[inc];
  }

  // Free the array
  free(res_arr);
  res_arr = NULL;

  return res;
}
```

```
void opencl_kernel_daxpy(int n, double a, cl_mem* cl_d_x
    , cl_mem* cl_d_y)
{
  size_t sizes[6];
  void *values[6];

  size_t global[1];
  size_t local[1];

  local[0] = CUKR_DAXPY_THREADS;
  global[0] = local[0] * CUKR_DAXPY_CTAS;

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(double);
  values[1] = (void *)&a;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)cl_d_x;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)cl_d_y;

  sizes[4] = CUKR_DAXPY_THREADS * sizeof(cl_double);
  values[4] = NULL;
  sizes[5] = CUKR_DAXPY_THREADS * sizeof(cl_double);
  values[5] = NULL;

  /* Kernel invocation */
  int x;
  for (x = 0; x<6; x++) {
    err = clSetKernelArg(ComputeKernel_daxpy, x, sizes[x
       ], values[x]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed %d failed\n", x);
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg %d failed\n", x);
      //return -1;
    }
  }

  /* Enqueue kernel for execution */
  err = clEnqueueNDRangeKernel(ComputeCommands,
     ComputeKernel_daxpy, 1, NULL, global, local, 0,
     NULL, &daxpy_event);
  if (err != CL_SUCCESS)
  {
```

```
    printf("clExecuteKernel␣failed␣%d\n",n);
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &daxpy_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }

}

void opencl_kernel_daypx(int n, double a, cl_mem* cl_d_x
   , cl_mem* cl_d_y)
{
  size_t sizes[6];
  void *values[6];

  size_t global[1];
  size_t local[1];

  local[0] = CUKR_DAYPX_THREADS;
  global[0] = local[0] * CUKR_DAYPX_CTAS;

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(double);
  values[1] = (void *)&a;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)cl_d_x;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)cl_d_y;

  sizes[4] = CUKR_DAYPX_THREADS * sizeof(cl_double);
  values[4] = NULL;
  sizes[5] = CUKR_DAYPX_THREADS * sizeof(cl_double);
  values[5] = NULL;

  /* Kernel invocation */
  int x;
  for (x = 0; x<6; x++) {
    err = clSetKernelArg(ComputeKernel_daypx, x, sizes[x
       ], values[x]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed␣%d␣failed\n", x);
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
```

```
      printf("clSetKernelArg␣%d␣failed\n", x);
      //return -1;
    }
  }

  /* Enqueue kernel for execution */
  err = clEnqueueNDRangeKernel(ComputeCommands,
     ComputeKernel_daypx, 1, NULL, global, local, 0,
     NULL, &daypx_event);
  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣failed␣%d\n",n);
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &daypx_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }
}

void opencl_kernel_dscal(int n, double a, cl_mem* x)
{
  size_t sizes[4];
  void *values[4];

  size_t global[1];
  size_t local[1];

  local[0] = CUKR_DSCAL_THREADS;
  global[0] = local[0] * CUKR_DSCAL_CTAS;

  if (x == NULL) {
    printf("NULL␣POINTER␣x!\n");
  }

  sizes[0] = sizeof(int);
  values[0] = (void *)&n;
  sizes[1] = sizeof(double);
  values[1] = (void *)&a;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)x;
  sizes[3] = CUKR_DSCAL_THREADS * sizeof(cl_double);
  values[3] = NULL;

  /* Kernel invocation */
  int i;
  for (i = 0; i<4; i++) {
```

```
    err = clSetKernelArg(ComputeKernel_dscal, i, sizes[i
       ], values[i]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed␣%d␣failed\n", i);
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", i);
      //return -1;
    }
  }

  /* Enqueue kernel for execution */
  err = clEnqueueNDRangeKernel(ComputeCommands,
      ComputeKernel_dscal, 1, NULL, global, local, 0,
      NULL, &dscal_event);
  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣sscal␣failed␣%d\n",n);
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &dscal_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }
}


void opencl_dcopy(int n, cl_mem* cl_d_x, int incx,
    cl_mem* cl_d_y, int incy)
{
  err = clEnqueueCopyBuffer(ComputeCommands, *cl_d_x, *
      cl_d_y, 0, 0, n * sizeof(cl_double), 0, NULL, &
      dcopy_event);
  if (err != CL_SUCCESS)
  {
    printf("clEnqueueCopyBuffer␣failed␣%d\n",n);
  }

  // Synchronize for timing
  err = clWaitForEvents(1, &dcopy_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }
}
```

```
void opencl_dspmv_csr(int rows, int cols, int nz, double
    alpha, cl_mem* d_ptr,
            cl_mem* d_idx, cl_mem* d_val, cl_mem* d_x,
               double beta, cl_mem* d_y)
{
  size_t sizes[7];
  void *values[7];

  sizes[0] = sizeof(int);
  values[0] = (void *)&rows;
  sizes[1] = sizeof(cl_mem);
  values[1] = (void *)d_ptr;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)d_idx;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)d_val;
  sizes[4] = sizeof(cl_mem);
  values[4] = (void *)d_x;
  sizes[5] = sizeof(cl_mem);
  values[5] = (void *)d_y;
  sizes[6] = sizeof(double);
  values[6] = (void *)&alpha;

  size_t global[1];
  size_t local[1];

  int inc;

  /* If beta != 1, do a scaling first */
  if (beta != 1 && beta)
    opencl_kernel_dscal(rows, beta, d_y);

  if (beta) {

    local[0] = CUKR_DSPMV_CSR_THREADS; //512;
    global[0] = CUKR_DSPMV_CSR_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 7; inc++) {
      err = clSetKernelArg(ComputeKernel_dspmv_csr, inc,
          sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
```

```
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_dspmv_csr, 1, NULL, global, local,
        0, NULL, &dspmv_csr_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣sspmv_csr␣failed␣%d\n", nz
          );
    }
    // Synchronize for timing
    err = clWaitForEvents(1, &dspmv_csr_event);
    if (err != CL_SUCCESS) {
      printf("clWaitForEvents␣failed!\n");
    }
  }

  else {
    // Set workgroup sizes
    local[0] = CUKR_DSPMV_CSR_THREADS; //512;
    global[0] = CUKR_DSPMV_CSR_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 7; inc++) {
      err = clSetKernelArg(ComputeKernel_dspmv_csr_b0,
          inc, sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
        //return -1;
      }
      if (err != CL_SUCCESS)
      {
        printf("clSetKernelArg␣%d␣failed\n", inc);
        //return -1;
      }
    }
    err = clEnqueueNDRangeKernel(ComputeCommands,
        ComputeKernel_dspmv_csr_b0, 1, NULL, global,
        local, 0, NULL, &dspmv_csr_b0_event);
    if (err != CL_SUCCESS)
    {
      printf("clExecuteKernel␣sspmv_csr_b0␣failed␣%d\n",
          nz);
    }
    // Synchronize for timing
    err = clWaitForEvents(1, &dspmv_csr_b0_event);
    if (err != CL_SUCCESS) {
```

```
      printf("clWaitForEvents␣failed!\n");
    }
  }
}

void opencl_dspmv_csr4(int rows, int cols, int nz,
    double alpha, cl_mem* d_ptr,
              cl_mem* d_idx, cl_mem* d_val, cl_mem* d_x,
                 double beta, cl_mem* d_y)
{
  size_t sizes[7];
  void *values[7];

  sizes[0] = sizeof(int);
  values[0] = (void *)&rows;
  sizes[1] = sizeof(cl_mem);
  values[1] = (void *)d_ptr;
  sizes[2] = sizeof(cl_mem);
  values[2] = (void *)d_idx;
  sizes[3] = sizeof(cl_mem);
  values[3] = (void *)d_val;
  sizes[4] = sizeof(cl_mem);
  values[4] = (void *)d_x;
  sizes[5] = sizeof(cl_mem);
  values[5] = (void *)d_y;
  sizes[6] = sizeof(double);
  values[6] = (void *)&alpha;

  size_t global[1];
  size_t local[1];

  int inc;

  if (beta != 1 && beta)
    opencl_kernel_sscal(rows, beta, d_y);

    /* If alpha = 1, no need to consider */
  if (beta) {
    // Set workgroup sizes
    local[0] = CUKR_DSPMV_CSR4_THREADS; //512;
    global[0] = CUKR_DSPMV_CSR4_CTAS * local[0];

    // Set kernel args
    for (inc = 0; inc < 7; inc++) {
      err = clSetKernelArg(ComputeKernel_dspmv_csr4, inc
          , sizes[inc], values[inc]);
      if (err == CL_INVALID_KERNEL)
      {
        printf("Failed␣%d␣failed\n", inc);
```

```
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", inc);
      //return -1;
    }
  }
  err = clEnqueueNDRangeKernel(ComputeCommands,
      ComputeKernel_dspmv_csr4, 1, NULL, global, local,
       0, NULL, &dspmv_csr4_event);
  if (err != CL_SUCCESS)
  {
    printf("clExecuteKernel␣sspmv_csr4␣failed␣%d\n",
        nz);
  }
  // Synchronize for timing
  err = clWaitForEvents(1, &dspmv_csr4_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }
}

else {
  // Set workgroup sizes
  local[0] = CUKR_DSPMV_CSR4_THREADS; //512;
  global[0] = CUKR_DSPMV_CSR4_CTAS * local[0];

  // Set kernel args
  for (inc = 0; inc < 7; inc++) {
    err = clSetKernelArg(ComputeKernel_dspmv_csr4_b0,
        inc, sizes[inc], values[inc]);
    if (err == CL_INVALID_KERNEL)
    {
      printf("Failed␣%d␣failed\n", inc);
      //return -1;
    }
    if (err != CL_SUCCESS)
    {
      printf("clSetKernelArg␣%d␣failed\n", inc);
      //return -1;
    }
  }
  err = clEnqueueNDRangeKernel(ComputeCommands,
      ComputeKernel_dspmv_csr4_b0, 1, NULL, global,
      local, 0, NULL, &dspmv_csr4_b0_event);
  if (err != CL_SUCCESS)
  {
```

```
      printf("clExecuteKernel␣sspmv_csr4_b0␣failed␣%d\n"
          , nz);
    }
    // Synchronize for timing
    err = clWaitForEvents(1, &dspmv_csr4_b0_event);
    if (err != CL_SUCCESS) {
      printf("clWaitForEvents␣failed!\n");
    }
  }
}

void opencl_dspmv_hyb(int rows, int cols, int nz, double
    alpha, int ell_nz_row, int ell_stride, int csr_nz,
                    cl_mem* d_ell_idx, cl_mem* d_ell_val
                       , cl_mem* d_csr_ptr, cl_mem*
                       d_csr_idx, cl_mem* d_csr_val,
                    cl_mem* d_x, double beta, cl_mem*
                       d_y)
{
#define DIVIDE_INTO(x, y) (((x) + (y) - 1)/(y))

  const unsigned int BLOCK_SIZE_ELL = 256;
  unsigned int num_blocks = DIVIDE_INTO(rows,
     BLOCK_SIZE_ELL);

  // Prepare data for the ell kernel
  size_t sizes[9];
  void *values[9];

  sizes[0] = sizeof(int);
  values[0] = (void *)&rows;
  sizes[1] = sizeof(double);
  values[1] = (void *)&alpha;
  sizes[2] = sizeof(int);
  values[2] = (void *)&ell_nz_row;
  sizes[3] = sizeof(int);
  values[3] = (void *)&ell_stride;
  sizes[4] = sizeof(cl_mem);
  values[4] = (void *)d_ell_idx;
  sizes[5] = sizeof(cl_mem);
  values[5] = (void *)d_ell_val;
  sizes[6] = sizeof(double);
  values[6] = (void *)&beta;
  sizes[7] = sizeof(cl_mem);
  values[7] = (void *)d_y;
  sizes[8] = sizeof(cl_mem);
  values[8] = (void *)d_x;

  size_t global[1];
```

```
size_t local[1];

local[0] = BLOCK_SIZE_ELL;
global[0] = num_blocks * local[0];

int inc;
// Set kernel args
for (inc = 0; inc < 9; inc++) {
  err = clSetKernelArg(ComputeKernel_dspmv_ell, inc,
      sizes[inc], values[inc]);
  if (err == CL_INVALID_KERNEL)
  {
    printf("Failed %d failed\n", inc);
  }
  if (err != CL_SUCCESS)
  {
    printf("clSetKernelArg %d failed\n", inc);
  }
}

// Launch the kernel
err = clEnqueueNDRangeKernel(ComputeCommands,
    ComputeKernel_dspmv_ell, 1, NULL, global, local, 0,
     NULL, &dspmv_hyb_event);
if (err != CL_SUCCESS)
{
  printf("clExecuteKernel dspmv_ell failed %d\n", nz);
}

local[0] = CUKR_DSPMV_CSR4_THREADS;
global[0] = CUKR_DSPMV_CSR4_CTAS * local[0];

if(csr_nz){
  // The rest in CSR4
    sizes[0] = sizeof(int);
    values[0] = (void *)&rows;
    sizes[1] = sizeof(cl_mem);
    values[1] = (void *)d_csr_ptr;
    sizes[2] = sizeof(cl_mem);
    values[2] = (void *)d_csr_idx;
    sizes[3] = sizeof(cl_mem);
    values[3] = (void *)d_csr_val;
    sizes[4] = sizeof(cl_mem);
    values[4] = (void *)d_x;
    sizes[5] = sizeof(cl_mem);
    values[5] = (void *)d_y;
    sizes[6] = sizeof(double);
    values[6] = (void *)&alpha;
```

```
      // Set kernel args
      for (inc = 0; inc < 7; inc++) {
        err = clSetKernelArg(ComputeKernel_dspmv_csr4,
            inc, sizes[inc], values[inc]);
        if (err == CL_INVALID_KERNEL)
        {
          printf("Failed␣%d␣failed\n", inc);
        }
        if (err != CL_SUCCESS)
        {
          printf("clSetKernelArg␣%d␣failed\n", inc);
        }
      }
      err = clEnqueueNDRangeKernel(ComputeCommands,
          ComputeKernel_dspmv_csr4, 1, NULL, global,
          local, 0, NULL, &dspmv_hyb_event);
      if (err != CL_SUCCESS)
      {
        printf("clExecuteKernel␣dspmv_csr4␣failed␣%d\n",
            nz);
      }
  }
  // Synchronize for timing
  err = clWaitForEvents(1, &dspmv_hyb_event);
  if (err != CL_SUCCESS) {
    printf("clWaitForEvents␣failed!\n");
  }
}

void opencl_dspmv_bcsr(int rows, int cols, int nz, int r
   , int c, double alpha, cl_mem* ptr,
             cl_mem* idx, cl_mem* val, cl_mem* x, double
                 beta,
             cl_mem* y)
{
  printf("test␣opencl_dspmv_bcsr\n");
}

void opencl_dspmv_bcsr4(int rows, int cols, int nz, int
   r, int c, double alpha, cl_mem* ptr,
            cl_mem* idx, cl_mem* val, cl_mem* x, double
               beta,
            cl_mem* y)
{
  printf("test␣opencl_dspmv_bcsr4\n");
}
```

E.30 Kernels GPU double set-up, header

```
/*
 *   kernels_single.h
 *
 *
 *   Created by Olav Aanes Fagerlund.
 *
 */

#include "../../../../init/init_opencl.h"

double opencl_kernel_ddot(int n, cl_mem* x, cl_mem* y);

void opencl_kernel_daxpy(int n, double a, cl_mem *cl_d_x
   , cl_mem *cl_d_y);

void opencl_kernel_daypx(int n, double a, cl_mem *cl_d_x
   , cl_mem *cl_d_y);

void opencl_dcopy(int n, cl_mem* x, int incx, cl_mem* y,
    int incy);

void opencl_kernel_dscal(int n, double a, cl_mem* x);

void opencl_dspmv_bcsr(int rows, int cols, int nz, int r
   , int c, double alpha, cl_mem* ptr,
             cl_mem* idx, cl_mem* val, cl_mem* x, double
                 beta, cl_mem* y);

void opencl_dspmv_bcsr4(int rows, int cols, int nz, int
   r, int c, double alpha, cl_mem* ptr,
            cl_mem* idx, cl_mem* val, cl_mem* x, double
               beta,
            cl_mem* y);

void opencl_dspmv_csr(int rows, int cols, int nz, double
    alpha, cl_mem* ptr,
            cl_mem* idx, cl_mem* val, cl_mem* x, double
               beta,
            cl_mem* y);

void opencl_dspmv_csr4(int rows, int cols, int nz,
   double alpha, cl_mem* ptr,
             cl_mem* idx, cl_mem* val, cl_mem* x, double
                 beta,
             cl_mem* y);

void opencl_dspmv_hyb(int rows, int cols, int nz, double
    alpha, int ell_nz_row, int ell_stride, int csr_nz,
```

```
            cl_mem* d_ell_idx, cl_mem* d_ell_val, cl_mem
               * d_csr_ptr, cl_mem* d_csr_idx, cl_mem*
               d_csr_val,
            cl_mem* d_x, double beta, cl_mem* d_y);
```

E.31 OpenCL Initialize

```
/*
 *  init_opencl.c
 *
 *
 *  Created by Olav Aanes Fagerlund, winter 2010.
 *
 */

#include "init_opencl.h"

static char *
load_source(const char *filename)
{
    struct stat statbuf;
    FILE        *fh;
    char        *source;

    fh = fopen(filename, "r");
    if (fh == 0)
        return 0;

    stat(filename, &statbuf);
    source = (char *) malloc(statbuf.st_size + 1);
    fread(source, statbuf.st_size, 1, fh);
    source[statbuf.st_size] = '\0';

    return source;
}

int loadAndBuild(const char *file, char *kernel){
  /* Load kernel sources into memory */
  char *kernel_src = load_source(file);

  /* Create the compute program memory object */
  ComputeProgram = clCreateProgramWithSource(
     ComputeContext, 1, (const char **) &kernel_src,
     NULL, &err);
  if (!ComputeProgram || err != CL_SUCCESS)
    {
        printf("Error:␣Failed␣to␣create␣compute␣program␣
           for␣file␣%s!\n", file);
        return EXIT_FAILURE;
```

```
  }
else {
  printf("CL␣compute␣program␣memory␣object␣set␣up␣
     successfully!\n");
}

/* Build the executable and add to the compute program
    memory object */
err = clBuildProgram(ComputeProgram, 0, NULL, /*"-cl-
   finite-math-only"*/NULL, NULL, NULL);
if (err == CL_BUILD_PROGRAM_FAILURE) {
  printf("JA!\n");
}
else if (err != CL_SUCCESS)
{
  printf("Error:␣Failed␣build␣program␣for␣kernel␣%s!\n
     ␣%d\n", kernel, err);
}
else {
  printf("CL␣compute␣program␣source␣for␣%s␣built␣
     successfully!\n", kernel);
}

if (kernel == "kernels_qdouble"){

  ComputeKernel_qdot = clCreateKernel(ComputeProgram,
     "kernel_qdot", &err);
  if (!ComputeKernel_qdot || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
       !\n", "kernel_qdot");
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
       successfully!\n");
  }

  ComputeKernel_qscal = clCreateKernel(ComputeProgram,
      "kernel_qscal", &err);
  if (!ComputeKernel_qscal || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
       !\n", "kernel_qscal");
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
       successfully!\n");
  }
```

```
ComputeKernel_qaxpy = clCreateKernel(ComputeProgram,
     "kernel_qaxpy", &err);
if (!ComputeKernel_qaxpy || err != CL_SUCCESS)
{
  printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
      !\n", "kernel_qaxpy");
}
else {
  printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
      successfully!\n");
}

ComputeKernel_qaypx = clCreateKernel(ComputeProgram,
     "kernel_qaypx", &err);
if (!ComputeKernel_qaypx || err != CL_SUCCESS)
{
  printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
      !\n", "kernel_qaypx");
}
else {
  printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
      successfully!\n");
}

ComputeKernel_qspmv_csr = clCreateKernel(
    ComputeProgram, "kernel_qspmv_csr", &err);
if (!ComputeKernel_qspmv_csr || err != CL_SUCCESS)
{
  printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
      !\n", "kernel_qspmv_csr");
}
else {
  printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
      successfully!\n");
}

ComputeKernel_qspmv_csr_a1 = clCreateKernel(
    ComputeProgram, "kernel_qspmv_csr_a1", &err);
if (!ComputeKernel_qspmv_csr_a1 || err != CL_SUCCESS
    )
{
  printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
      !\n", "kernel_qspmv_csr_a1");
}
else {
  printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
      successfully!\n");
}
```

```
ComputeKernel_qspmv_csr_a1_b0 = clCreateKernel(
    ComputeProgram, "kernel_qspmv_csr_a1_b0", &err);
if (!ComputeKernel_qspmv_csr_a1_b0 || err !=
    CL_SUCCESS)
{
  printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
     !\n", "kernel_qspmv_csr_a1_b0");
}
else {
  printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
     successfully!\n");
}

ComputeKernel_qspmv_csr_b0 = clCreateKernel(
    ComputeProgram, "kernel_qspmv_csr_b0", &err);
if (!ComputeKernel_qspmv_csr_b0 || err != CL_SUCCESS
    )
{
  printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
     !\n", "kernel_qspmv_csr_b0");
}
else {
  printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
     successfully!\n");
}

ComputeKernel_qspmv_csr4 = clCreateKernel(
    ComputeProgram, "kernel_qspmv_csr4", &err);
if (!ComputeKernel_qspmv_csr4 || err != CL_SUCCESS)
{
  printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
     !\n", "kernel_qspmv_csr4");
}
else {
  printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
     successfully!\n");
}

ComputeKernel_qspmv_csr4_a1 = clCreateKernel(
    ComputeProgram, "kernel_qspmv_csr4_a1", &err);
if (!ComputeKernel_qspmv_csr4_a1 || err !=
    CL_SUCCESS)
{
  printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
     !\n", "kernel_qspmv_csr4_a1");
}
else {
  printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
     successfully!\n");
```

```
  }

  ComputeKernel_qspmv_csr4_a1_b0 = clCreateKernel(
      ComputeProgram, "kernel_qspmv_csr4_a1_b0", &err);
  if (!ComputeKernel_qspmv_csr4_a1_b0 || err !=
      CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", "kernel_qspmv_csr4_a1_b0");
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }

  ComputeKernel_qspmv_csr4_b0 = clCreateKernel(
      ComputeProgram, "kernel_qspmv_csr4_b0", &err);
  if (!ComputeKernel_qspmv_csr4_b0 || err !=
      CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", "kernel_qspmv_csr4_b0");
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }

  ComputeKernel_qspmv_ell = clCreateKernel(
      ComputeProgram, "kernel_qspmv_ell", &err);
  if (!ComputeKernel_qspmv_ell || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", "kernel_qspmv_ell");
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_sdot"){
  ComputeKernel_sdot = clCreateKernel(ComputeProgram,
      kernel, &err);
  if (!ComputeKernel_sdot || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
```

```
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }

  size_t auto_local;

  // Set workgroup sizes
  err = clGetKernelWorkGroupInfo(ComputeKernel_sdot,
      ComputeDeviceId, CL_KERNEL_WORK_GROUP_SIZE,
      sizeof(size_t), &auto_local, NULL);
  if (err != CL_SUCCESS) {
    printf("clGetKernelWorkGroupInfo␣failed!␣:␣%d\n",
        err);
  }
  AUTO_LOCAL_SIZE_SDOT = auto_local;
}

else if (kernel == "kernel_ddot"){
  ComputeKernel_ddot = clCreateKernel(ComputeProgram,
      kernel, &err);
  if (!ComputeKernel_ddot || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_sscal"){
  ComputeKernel_sscal = clCreateKernel(ComputeProgram,
      kernel, &err);
  if (!ComputeKernel_sscal || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_dscal"){
  ComputeKernel_dscal = clCreateKernel(ComputeProgram,
      kernel, &err);
```

```
  if (!ComputeKernel_dscal || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_saxpy"){
  ComputeKernel_saxpy = clCreateKernel(ComputeProgram,
       kernel, &err);
  if (!ComputeKernel_saxpy || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_daxpy"){
  ComputeKernel_daxpy = clCreateKernel(ComputeProgram,
       kernel, &err);
  if (!ComputeKernel_daxpy || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_saypx"){
  ComputeKernel_saypx = clCreateKernel(ComputeProgram,
       kernel, &err);
  if (!ComputeKernel_saypx || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
```

```
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_daypx"){
  ComputeKernel_daypx = clCreateKernel(ComputeProgram,
      kernel, &err);
  if (!ComputeKernel_daypx || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_sspmv_csr"){
  ComputeKernel_sspmv_csr = clCreateKernel(
      ComputeProgram, kernel, &err);
  if (!ComputeKernel_sspmv_csr || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_sspmv_csr_a1"){
  ComputeKernel_sspmv_csr_a1 = clCreateKernel(
      ComputeProgram, kernel, &err);
  if (!ComputeKernel_sspmv_csr_a1 || err != CL_SUCCESS
      )
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_sspmv_csr_a1_b0"){
```

```
    ComputeKernel_sspmv_csr_a1_b0 = clCreateKernel(
        ComputeProgram, kernel, &err);
    if (!ComputeKernel_sspmv_csr_a1_b0 || err !=
        CL_SUCCESS)
    {
      printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
          !\n", kernel);
    }
    else {
      printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
          successfully!\n");
    }
  }

  else if (kernel == "kernel_sspmv_csr_b0"){
    ComputeKernel_sspmv_csr_b0 = clCreateKernel(
        ComputeProgram, kernel, &err);
    if (!ComputeKernel_sspmv_csr_b0 || err != CL_SUCCESS
        )
    {
      printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
          !\n", kernel);
    }
    else {
      printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
          successfully!\n");
    }
  }
  else if (kernel == "kernel_sspmv_csr4"){
    ComputeKernel_sspmv_csr4 = clCreateKernel(
        ComputeProgram, kernel, &err);
    if (!ComputeKernel_sspmv_csr4 || err != CL_SUCCESS)
    {
      printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
          !\n", kernel);
    }
    else {
      printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
          successfully!\n");
    }
  }

  else if (kernel == "kernel_sspmv_csr4_a1"){
    ComputeKernel_sspmv_csr4_a1 = clCreateKernel(
        ComputeProgram, kernel, &err);
    if (!ComputeKernel_sspmv_csr4_a1 || err !=
        CL_SUCCESS)
    {
```

```
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_sspmv_csr4_a1_b0"){
  ComputeKernel_sspmv_csr4_a1_b0 = clCreateKernel(
      ComputeProgram, kernel, &err);
  if (!ComputeKernel_sspmv_csr4_a1_b0 || err !=
      CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_sspmv_csr4_b0"){
  ComputeKernel_sspmv_csr4_b0 = clCreateKernel(
      ComputeProgram, kernel, &err);
  if (!ComputeKernel_sspmv_csr4_b0 || err !=
      CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_sspmv_ell"){
  ComputeKernel_sspmv_ell = clCreateKernel(
      ComputeProgram, kernel, &err);
  if (!ComputeKernel_sspmv_ell || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
```

```
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

// SPMV double ones
else if (kernel == "kernel_dspmv_ell"){
  ComputeKernel_dspmv_ell = clCreateKernel(
      ComputeProgram, kernel, &err);
  if (!ComputeKernel_dspmv_ell || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_dspmv_csr"){
  ComputeKernel_dspmv_csr = clCreateKernel(
      ComputeProgram, kernel, &err);
  if (!ComputeKernel_dspmv_csr || err != CL_SUCCESS)
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}

else if (kernel == "kernel_dspmv_csr_b0"){
  ComputeKernel_dspmv_csr_b0 = clCreateKernel(
      ComputeProgram, kernel, &err);
  if (!ComputeKernel_dspmv_csr_b0 || err != CL_SUCCESS
      )
  {
    printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
        !\n", kernel);
  }
  else {
    printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
        successfully!\n");
  }
}
```

```
  else if (kernel == "kernel_dspmv_csr4"){
    ComputeKernel_dspmv_csr4 = clCreateKernel(
        ComputeProgram, kernel, &err);
    if (!ComputeKernel_dspmv_csr4 || err != CL_SUCCESS)
    {
      printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
          !\n", kernel);
    }
    else {
      printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
          successfully!\n");
    }
  }

  else if (kernel == "kernel_dspmv_csr4_b0"){
    ComputeKernel_dspmv_csr4_b0 = clCreateKernel(
        ComputeProgram, kernel, &err);
    if (!ComputeKernel_dspmv_csr4_b0 || err !=
        CL_SUCCESS)
    {
      printf("Error:␣Failed␣to␣create␣compute␣kernel␣%s
          !\n", kernel);
    }
    else {
      printf("CL␣compute␣kernel␣memory␣object␣set␣up␣
          successfully!\n");
    }
  }
}

int CukrInit_OpenCL_cl(){
  unsigned int num_devices_to_use = 1;
  unsigned int size_comp_dev_id = 1;

  unsigned int num_platforms = 0;
  unsigned int num_devices = 0;

  /* Find the OpenCL platform */
  err = clGetPlatformIDs(1, &ComputePlatformId, &
      num_platforms);
  if (err != CL_SUCCESS)
    {
        printf("Error:␣Failed␣to␣get␣platform␣IDs!␣%d\n"
            , err);
        return EXIT_FAILURE;
    }

  /* Find a GPU device */
```

```
err = clGetDeviceIDs(ComputePlatformId,
    CL_DEVICE_TYPE_GPU, size_comp_dev_id, &
    ComputeDeviceId, &num_devices);
if (err != CL_SUCCESS)
  {
      printf("Error:␣Failed␣to␣get␣device␣IDs!␣%d\n",
          err);
      return EXIT_FAILURE;
  }
printf("\n*␣*␣*␣%d␣OpenCL␣devices␣found␣in␣the␣system␣
    *␣*␣*\n", num_devices);

/* Create a compute context using the found OpenCL
    device */
ComputeContext = clCreateContext(NULL,
    num_devices_to_use, &ComputeDeviceId, NULL, NULL, &
    err);
if (err != CL_SUCCESS)
  {
      printf("Error:␣Failed␣to␣create␣compute␣context!
          ␣%d\n", err);
      return EXIT_FAILURE;
  }
else {
  printf("CL␣compute␣context␣set␣up␣successfully!\n");
}

/* Create the command queue */
ComputeCommands = clCreateCommandQueue(ComputeContext,
     ComputeDeviceId, /*CL_QUEUE_PROFILING_ENABLE*/0, &
    err);
if (err != CL_SUCCESS)
{
  printf("clCreateCommandQueue␣failed\n␣%d\n", err);
  return -1;
}

// Load and build all qdouble kernels (in same file as
     they depend on same ds_ops)
loadAndBuild("src/kernels_qdouble.cl", "
    kernels_qdouble");

loadAndBuild("src/kernel_saxpy.cl", "kernel_saxpy");
loadAndBuild("src/kernel_daxpy.cl", "kernel_daxpy");

loadAndBuild("src/kernel_saypx.cl", "kernel_saypx");
loadAndBuild("src/kernel_daypx.cl", "kernel_daypx");
```

```
  loadAndBuild("src/kernel_sdot.cl", "kernel_sdot");
  loadAndBuild("src/kernel_ddot.cl", "kernel_ddot");

  loadAndBuild("src/kernel_sscal.cl", "kernel_sscal");
  loadAndBuild("src/kernel_dscal.cl", "kernel_dscal");

  loadAndBuild("src/kernel_sspmv_csr.cl", "
     kernel_sspmv_csr");
  loadAndBuild("src/kernel_sspmv_csr_a1.cl", "
     kernel_sspmv_csr_a1");
  loadAndBuild("src/kernel_sspmv_csr_a1_b0.cl", "
     kernel_sspmv_csr_a1_b0");
  loadAndBuild("src/kernel_sspmv_csr_b0.cl", "
     kernel_sspmv_csr_b0");

  loadAndBuild("src/kernel_sspmv_csr4.cl", "
     kernel_sspmv_csr4");
  loadAndBuild("src/kernel_sspmv_csr4_a1.cl", "
     kernel_sspmv_csr4_a1");
  loadAndBuild("src/kernel_sspmv_csr4_a1_b0.cl", "
     kernel_sspmv_csr4_a1_b0");
  loadAndBuild("src/kernel_sspmv_csr4_b0.cl", "
     kernel_sspmv_csr4_b0");


  loadAndBuild("src/kernel_dspmv_csr.cl", "
     kernel_dspmv_csr");
  loadAndBuild("src/kernel_dspmv_csr_b0.cl", "
     kernel_dspmv_csr_b0");

  loadAndBuild("src/kernel_dspmv_csr4.cl", "
     kernel_dspmv_csr4");
  loadAndBuild("src/kernel_dspmv_csr4_b0.cl", "
     kernel_dspmv_csr4_b0");

  loadAndBuild("src/kernel_sspmv_ell.cl", "
     kernel_sspmv_ell");
  loadAndBuild("src/kernel_dspmv_ell.cl", "
     kernel_dspmv_ell");

  return num_devices;
}
```

E.32 OpenCL Initialize, header

```
/*
 *  init_opencl.h
 *
 *
```

```
 *  Created by Olav Aanes Fagerlund.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif


cl_context                         ComputeContext;
cl_command_queue                   ComputeCommands;

cl_kernel                          ComputeKernel_sdot;
cl_kernel                          ComputeKernel_qdot;
cl_kernel                          ComputeKernel_ddot;
cl_kernel                          ComputeKernel_saxpy;
cl_kernel                          ComputeKernel_qaxpy;
cl_kernel                          ComputeKernel_daxpy;
cl_kernel                          ComputeKernel_saypx;
cl_kernel                          ComputeKernel_qaypx;
cl_kernel                          ComputeKernel_daypx;
cl_kernel                          ComputeKernel_sscal;
cl_kernel                          ComputeKernel_qscal;
cl_kernel                          ComputeKernel_dscal;

cl_kernel            ComputeKernel_sspmv_csr;
cl_kernel            ComputeKernel_sspmv_csr_a1;
cl_kernel            ComputeKernel_sspmv_csr_a1_b0;
cl_kernel            ComputeKernel_sspmv_csr_b0;
cl_kernel            ComputeKernel_sspmv_csr4;
cl_kernel            ComputeKernel_sspmv_csr4_a1;
cl_kernel            ComputeKernel_sspmv_csr4_a1_b0;
cl_kernel            ComputeKernel_sspmv_csr4_b0;

cl_kernel            ComputeKernel_qspmv_csr;
cl_kernel            ComputeKernel_qspmv_csr_a1;
cl_kernel            ComputeKernel_qspmv_csr_a1_b0;
cl_kernel            ComputeKernel_qspmv_csr_b0;
cl_kernel            ComputeKernel_qspmv_csr4;
cl_kernel            ComputeKernel_qspmv_csr4_a1;
cl_kernel            ComputeKernel_qspmv_csr4_a1_b0;
cl_kernel            ComputeKernel_qspmv_csr4_b0;
```

```
cl_kernel             ComputeKernel_dspmv_csr;
cl_kernel             ComputeKernel_dspmv_csr_b0;
cl_kernel             ComputeKernel_dspmv_csr4;
cl_kernel             ComputeKernel_dspmv_csr4_b0;

cl_kernel             ComputeKernel_sspmv_ell;
cl_kernel             ComputeKernel_qspmv_ell;
cl_kernel             ComputeKernel_dspmv_ell;

static cl_program       ComputeProgram;
static cl_platform_id   ComputePlatformId;
cl_device_id           ComputeDeviceId;

int AUTO_LOCAL_SIZE_SDOT;

int err;
cl_mem test;
cl_event  qdot_event,  sdot_event,  ddot_event,
      qaxpy_event, saxpy_event, daxpy_event,
      qaypx_event, saypx_event, daypx_event,
      qscal_event, sscal_event, dscal_event,
      scopy_event, dcopy_event,

      sspmv_csr_event, sspmv_csr_a1_b0_event,
         sspmv_csr_a1_event, sspmv_csr_b0_event,
      sspmv_csr4_event, sspmv_csr4_a1_b0_event,
         sspmv_csr4_a1_event, sspmv_csr4_b0_event,

      qspmv_csr_event, qspmv_csr_a1_b0_event,
         qspmv_csr_a1_event, qspmv_csr_b0_event,
      qspmv_csr4_event, qspmv_csr4_a1_b0_event,
         qspmv_csr4_a1_event, qspmv_csr4_b0_event,

      dspmv_csr_event, dspmv_csr_b0_event,
      dspmv_csr4_event, dspmv_csr4_b0_event,

      sspmv_hyb_event, qspmv_hyb_event, dspmv_hyb_event;


static char * load_source(const char *filename);
int loadAndBuild(const char *filename, char *kernelname)
   ;
int CukrInit_OpenCL_cl();
```

E.33 OpenCL devices probing

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/stat.h>

#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif

int
main(int argc, char *argv[])
{
  cl_device_id compute_device_id[3];
  unsigned int num_devices = 0;
  int return_value = 0;

  cl_platform_id platform;
  cl_uint num_platforms;
  clGetPlatformIDs(1, &platform, &num_platforms);

  return_value = clGetDeviceIDs(platform,
     CL_DEVICE_TYPE_ALL, 3, compute_device_id, &
     num_devices);
  printf("\n*␣*␣*␣%d␣OpenCL␣devices␣found␣in␣the␣system␣
     *␣*␣*\n", num_devices);
  printf("\n");

  size_t ret_size;

  int x;
  for (x = 0; x < num_devices; x++) {

    printf("Device␣number␣%d␣:\n
       -----------------------------------------\n", x);

    clGetPlatformInfo(platform, CL_PLATFORM_VENDOR,
       sizeof(char), NULL, &ret_size);

    char platform_name[ret_size];
    clGetPlatformInfo(platform, CL_PLATFORM_VENDOR,
       sizeof(char[ret_size]), platform_name, NULL);
    printf("CL␣platform␣vendor:␣");
    int b;
    for (b = 0; b < ret_size; b++) {
      printf("%c", platform_name[b]);
    }
```

```
printf("\n");

clGetPlatformInfo(platform, CL_PLATFORM_VERSION,
   sizeof(char), NULL, &ret_size);
clGetPlatformInfo(platform, CL_PLATFORM_VERSION,
   sizeof(char[ret_size]), platform_name, NULL);
printf("CL␣platform␣version:␣");
for (b = 0; b < ret_size; b++) {
  printf("%c", platform_name[b]);
}
printf("\n");

clGetDeviceInfo(compute_device_id[x], CL_DEVICE_NAME
   , sizeof(char), NULL, &ret_size);
char devName[ret_size];

clGetDeviceInfo(compute_device_id[x], CL_DEVICE_NAME
   , sizeof(char[ret_size]), devName, NULL);
printf("CL␣device␣name:␣");
int i;
for (i = 0; i < ret_size; i++) {
  printf("%c", devName[i]);
}
printf("\n");

unsigned int maxComputeUnits = 0;
clGetDeviceInfo(compute_device_id[x],
   CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint), &
   maxComputeUnits, NULL);
printf("Max␣compute␣units:␣%d\n", maxComputeUnits);

unsigned int clockFreq = 0;
clGetDeviceInfo(compute_device_id[x],
   CL_DEVICE_MAX_CLOCK_FREQUENCY, sizeof(cl_uint), &
   clockFreq, NULL);
printf("Clock␣frequency:␣%d␣MHz\n", clockFreq);

unsigned long globalMemSize = 0;
clGetDeviceInfo(compute_device_id[x],
   CL_DEVICE_GLOBAL_MEM_SIZE, sizeof(cl_ulong), &
   globalMemSize, NULL);
printf("Device␣global␣memory␣size:␣%ld␣MB\n",
   globalMemSize/(1024*1024));

unsigned long long globalMemCacheSize = 0;
clGetDeviceInfo(compute_device_id[x],
   CL_DEVICE_GLOBAL_MEM_CACHE_SIZE, sizeof(cl_ulong)
   , &globalMemCacheSize, NULL);
```

```
printf("Device␣global␣memory␣cache␣size:␣%lld␣KB\n",
     globalMemCacheSize/1024);

unsigned int globalMemCacheLine = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE, sizeof(
    cl_uint), &globalMemCacheLine, NULL);
printf("Device␣global␣memory␣cache␣line␣size:␣%d␣
    Bytes\n", globalMemCacheLine);

unsigned long long localMemSize = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_LOCAL_MEM_SIZE, sizeof(cl_ulong), &
    localMemSize, NULL);
printf("Device␣local␣memory␣size:␣%lld␣KB\n",
    localMemSize/1024);

unsigned int localMemType = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_LOCAL_MEM_TYPE, sizeof(
    cl_device_local_mem_type), &localMemType, NULL);
if (localMemType == CL_LOCAL) {
  printf("Device␣local␣memory␣is␣physical␣memory␣
      type:␣CL_LOCAL␣\n");
}
else if (localMemType == CL_GLOBAL) {
  printf("Device␣local␣memory␣is␣physical␣memory␣
      type:␣CL_GLOBAL␣\n");
}

unsigned long long constBufferSize = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE, sizeof(
    cl_ulong), &constBufferSize, NULL);
printf("Device␣max␣constant␣buffer␣size:␣%lld␣KB\n",
     constBufferSize/1024);

unsigned int maxWorkItemDimensions = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof(
    cl_uint), &maxWorkItemDimensions, NULL);
printf("Device␣max␣work-item␣dimensions:␣%d\n",
    maxWorkItemDimensions);

unsigned int maxWorkGroupSize = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size_t), &
    maxWorkGroupSize, NULL);
```

```
printf("Device␣max␣work-group␣size:␣%d␣threads\n",
    maxWorkGroupSize);

unsigned int timerResolution = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_PROFILING_TIMER_RESOLUTION, sizeof(
    size_t), &timerResolution, NULL);
printf("Device␣profiling␣timer␣resolution:␣%d␣
    nanoseconds\n",timerResolution);

unsigned int vector_w_i = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT, sizeof(
    size_t), &vector_w_i, NULL);
printf("Device␣preferred␣vector␣width␣int:␣%d\n",
    vector_w_i);

unsigned int vector_w_f = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT, sizeof(
    size_t), &vector_w_f, NULL);
printf("Device␣preferred␣vector␣width␣float:␣%d\n",
    vector_w_f);

unsigned int vector_w_d = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE, sizeof(
    size_t), &vector_w_d, NULL);
printf("Device␣preferred␣vector␣width␣double:␣%d\n",
     vector_w_d);

unsigned int image_support = 0;
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_IMAGE_SUPPORT, sizeof(size_t), &
    image_support, NULL);
printf("Device␣image␣support␣(1:␣true,␣0␣false):␣%d\
    n", image_support);

clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_EXTENSIONS, sizeof(char), NULL, &
    ret_size);
char extensions[ret_size];
clGetDeviceInfo(compute_device_id[x],
    CL_DEVICE_EXTENSIONS, sizeof(char[ret_size]), &
    extensions, NULL);

printf("Extensions␣supported␣:\n␣␣");
int j;
for (j = 0; j < ret_size; j++) {
```

```
      if(extensions[j] == '␣')
        printf("\n␣␣");
      else
        printf("%c", extensions[j]);
    }
    printf("\n\n");
  }
}
```