NTNU

Norwegian University of
Science and Technology

Fast Implementation of Two Hash Algorithms on nVidia CUDA GPU

Gorka Lerchundi Osa

Master of Science in Communication Technology
Submission date: February 2009
Supervisor: Danilo Gligoroski, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

For the time being, SHA-2 a trustworthy hash algorithm is being used over the net. But who could assure that this is not going to do an unexpected turn with someone's release of a breaking algorithm which could put in risk the integrity of most of the trusted secure communications?

To avoid this, NIST (National Institute of Standards and Technology) has created "SHA-3 hash competition" in the same way as they done with AES encryption algorithm. When this competition finishes it's quite sure that the most important feature of the outgoing algorithm will be how much strong and hard-breaking it is. But how fast it is, is also important.

The task is about implementing a working version of Blue Midnight Wish algorithm (an algorithm designed by professor Danilo Gligoroski from NTNU and Vlastimil Klima an independent cryptographer from Czech Republic) presented at NIST competition into nVidia GPU to get as much speedup as possible. The work can be divided in a theoretical and a practical part. First, study the current and future possible parallel capable technologies. Secondly, implement a CUDA based version and get results which will help to assure what can and what can't be done with these technologies.

Assignment given: 23. September 2008
Supervisor: Danilo Gligoroski, ITEM

FAST IMPLEMENTATION OF TWO HASH ALGORITHMS ON NVIDIA CUDA GPU

Master Thesis

Student:	**Gorka Lertxundi Osa**
Supervisor:	**Danilo Gligoroski**
Institute:	**Institutt for telematikk**
Faculty:	**Fakultet for Informasjonsteknologi, matematikk og elektroteknikk**

1. Abstract

User needs increases as time passes. We started with computers like the size of a room where the perforated plaques did the same function as the current machine code object does and at present we are at a point where the number of processors within our graphic device unit it's not enough for our requirements.

A change in the evolution of computing is looming. We are in a transition where the sequential computation is losing ground on the benefit of the distributed. And not because of the birth of the new GPUs easily accessible this trend is novel but long before it was used for projects like SETI@Home, fightAIDS@Home, ClimatePrediction and there were shouting from the rooftops about what was to come. Grid computing was its formal name. Until now it was linked only to distributed systems over the network, but as this technology evolves it will take different meaning.

nVidia with CUDA has been one of the first companies to make this kind of software package noteworthy. Instead of being a proof of concept it's a real tool. Where the transition is expressed in greater magnitude in which the true artist is the programmer who uses it and achieves performance increases.

As with many innovations, a community distributed worldwide has grown behind this software package and each one doing its bit. It is noteworthy that after CUDA release a lot of software developments grown like the cracking of the hitherto insurmountable WPA.

With Sony-Toshiba-IBM (STI) alliance it could be said the same thing, it has a great community and great software (IBM is the company in charge of maintenance). Unlike nVidia is not as accessible as it is but IBM is powerful enough to enter home made supercomputing market. In this case, after IBM released the PS3 SDK, a notorious application was created using the benefits of parallel computing named Folding@Home. Its purpose is to, inter alia, find the cure for cancer.

To sum up, this is only the beginning, and in this thesis is sized up the possibility of using this technology for accelerating cryptographic hash algorithms. BLUE MIDNIGHT WISH (The hash algorithm that is applied to the surgery) is undergone to an environment change adapting it to a parallel capable code for creating empirical measures that compare to the current sequential implementations. It will answer questions that nowadays haven't been answered yet.

BLUE MIDNIGHT WISH is a candidate hash function for the next NIST standard SHA-3, designed by professor Danilo Gligoroski from NTNU and Vlastimil Klima – an independent cryptographer from Czech Republic.

So far, from speed point of view BLUE MIDNIGHT WISH is on the top of the charts (generally on the second place – right behind EDON-R - another hash function from professor Danilo Gligoroski).

One part of the work on this thesis was to investigate is it possible to achieve faster speeds in processing of Blue Midnight Wish when the computations are distributed among the cores in a CUDA device card. My numerous experiments give a clear answer: NO. Although the answer is negative, it still has a significant scientific value. The point is that my work acknowledges viewpoints and standings of a part of the cryptographic community that is doubtful that the cryptographic primitives will benefit when executed in parallel in many cores in one CPU. Indeed, my experiments show that the communication costs between cores in CUDA outweigh by big margin the computational costs done inside one core (processor) unit.

2. Index

3. Introduction

3.1. Goal

The goal of this thesis was, at the first time, the implementation of two hash algorithms developed in 'Institutt for telematikk' at NTNU into a specific GPGPU (General-Purpose Computing on Graphics Processing Units), concretely in a CUDA based graphic device card.

But after some tests in the thesis, it was slightly modified in able to get the needed results in this area. As it can be seen in the results, these are not as hopeful as expected. Consequently, the thesis was focused in the results and tries to answer why could or couldn't be these kinds of implementations applied to cryptographic area.

3.2. Methodology

Methodology followed is the next:

- **Diagnosis:** It is studied the project features and the goal it is needed.
- **Analysis:** It is analyzed the goal and proposed some solutions to this.
- **Proposal:** It is taken only one proposal and it is designed or thought how to integrate into the project.
- **Implementation:** It is implemented the proposal that was decided in the last step.
- **Results:** It is observed the final results of the proposal and if the result doesn't fit very well it is tried with another analyzed proposal.

3.3. Tools

Different operating systems require different tools to put the thesis working. There are different tools that together achieve the same result. In this thesis Windows and Mac OS X were used. The first one was used in the computer where the CUDA based device card was installed, named as "Thesis Computer". Mac OS X instead was the operating system used by the writer of this thesis named as "Personal Computer". It was necessary, because the big differences between these two O.S., to separate the tools description into two sections:

3.3.1. Thesis Computer

The *Thesis Computer* was a fresh Windows XP SP3 installation with nothing more than browsers, office automation tools and that kind of stuff. The goal of this computer was to run benchmarks with the selected and implemented proposal and create reports as output.

These were the *Thesis Computer's* features, only the CPU and GPU are shown because it is what really matters:

- **CPU:**

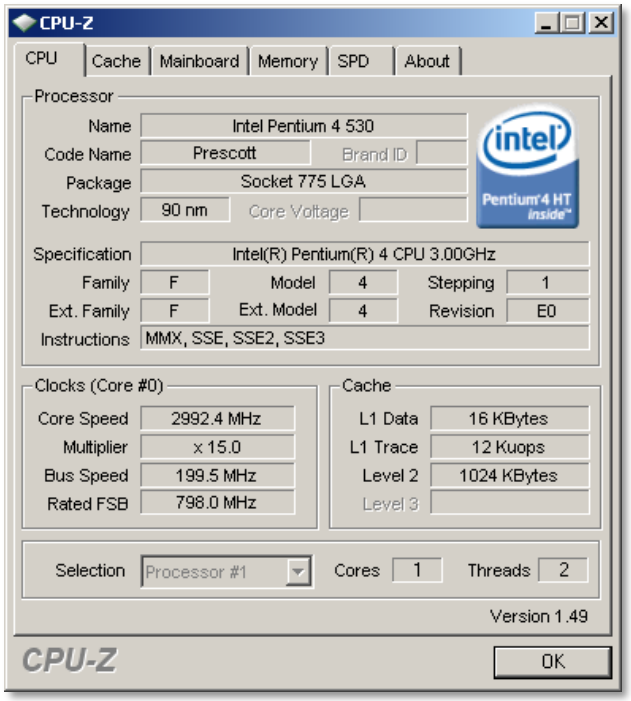


Figure 3.1: CPU-Z application showin CPU information

- **GPU:**

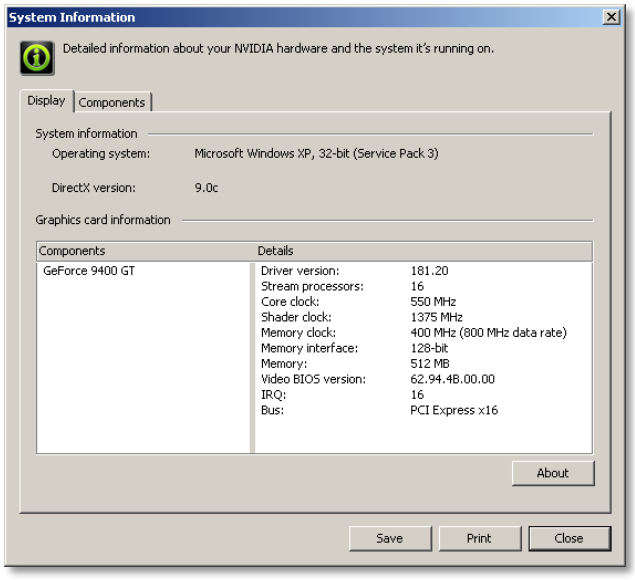


Figure 3.2: nVidia driver showing GPU information

```
There is 1 device supporting CUDA

Device 0: "GeForce 9400 GT"
  Major revision number:                         1
  Minor revision number:                         1
  Total amount of global memory:                 536543232 bytes
  Number of multiprocessors:                     2
  Number of cores:                               16
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.40 GHz
  Concurrent copy and execution:                 Yes
```

Figure 3.3: A homebrew application that retrieves GPU information

To get this computer working drivers for both GPU and CUDA were needed. In summary these are the used drivers:

- **GPU driver:** nVidia GeForce 182.20
- **CUDA driver:** 2.1
- **Software Development Kit:** nVidia CUDA SDK 2.10.1215.2015

After installing these packages and testing over the SDK samples that all worked well it was time to install the most known IDE (Integrated Development Environment), RAD (Rapid Application Development) or whatever it is, Microsoft Visual Studio (MVS). In this case it is necessary to install ***Microsoft Visual Studio 2005*** (8.0 internal version) to maintain the compatibility with the samples in the SDK. It also was helpful **CUDA VS Wizard**, a template to do the task of creating CUDA projects into Microsoft Visual Studio easier.

VNC (Virtual Network Computing) was installed using **TightVNC** package to be able to work without been physically where the "Thesis Computer" was.

To transfer some files and write the thesis SSH was installed too. **OpenSSH for Windows** was used.

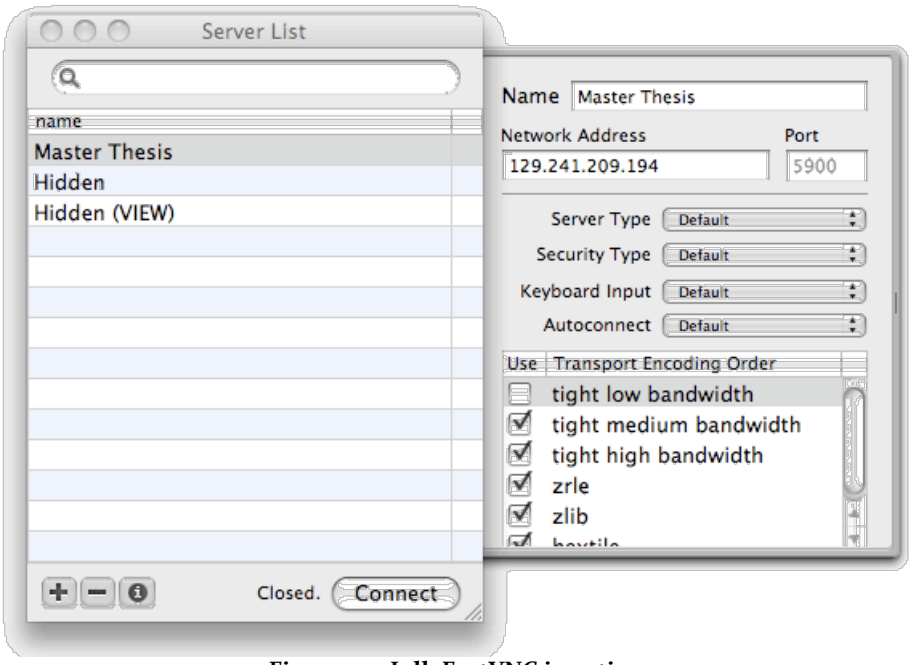
3.3.1. Personal Computer

The *Personal Computer* was used at the first stage of the process, coding the proposal and delivering to *Thesis Computer* a compiling implementation. This could be done because CUDA was able to compile and run projects without having any CUDA based device card. This was called *Emulation Mode* and as it says emulates an inexistent CUDA device. In this case, only nVidia SDK (the same version) was needed.

Because MVS cannot be used in Mac OS X alternatives were needed. In this case, a pack of tools was used to replace MVS:

- **GCC (GNU Compiler Collection):** The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages (libstdc++, libgcj, ...). Concretely, gcc/g++ was used to compile non-CUDA code.
- **SCons (A Software Construction Tool):** Is an improved, cross-platform substitute for the classic *make* utility with integrated functionality similar to *autoconf/automake* and compiler caches such as *ccache*. In short, SCons is an easier, more reliable and faster way to build software. It was prepared to work with CUDA (using CudaTool) on UNiX based systems.
- **VIM (Vi IMproved):** Vim is a highly configurable text editor built to enable efficient text editing. It is an improved version of the vi editor distributed with most UNIX systems. It was used to code CUDA programs as well as SConstruct files.
- **GNU Octave:** GNU Octave is a high-level language, primarily intended for numerical computations. It provides a convenient command line interface for solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with Matlab. It was used to create charts with generated benchmarking data.

Paralelly with TightVNC server, **JollyFastVNC** was used as client. Actually, at present it is possible to get inside *Thesis Computer*:



login: bruker
password: rekurb

Figure 3.4: JollyFastVNC in action

4. Background

The history before reaching the actual position in the graphic devices market has had a typical evolutionary way, the usual way to converge into the point that all the enterprises were looking for. Leaving the battles and saying "it's over" to the war of conquering the whole market and dividing this into pieces for controlling their own bubble. As it usually happens some battles will happen during the next years because some pieces of this market will bump into the others controlling bubble. Nowadays the war is to keep their bubbles intact and try to be innovative to control the upcoming sectors.

Concretely, we are talking about the war of the parallel computing. And how we get into this war is what the next lines will explain.

All the users and developers since the first mono task processor was created were finding parallel/multi task execution, at least the sensation of that without taking care it was real or not. From the mono task processor we evolved to virtual multi task over mono task processor, to made multi task transparent to the end-user. And now we have broken the hardware frontier where we were. At present, there are a lot of solutions with multi core ("processor") in the CPU or GPU market to get our real multi task system.

While one core processor manufacturing technology continues to improve, minimizing the transistor sizes, physical limits have become a major problem. As predicted by Moore's law, the main problem is, at this moment, the heat dissipation in that reduced transistor sizes and by consequence the problems that occur like incorrect data synchronization. This is why the most important manufacturers changed their priority. While getting core frequency increased is getting more and more complicated (It's expected to get atom sizes in 2 or 3 generations) they decided to change how to deal with the problem, using multi cores.

This new way to tackle the problem has advantages and disadvantages. If the operating system implements multi core processors, multi tasking is improved in the same quantity of cores has the processor, and the end user will notice immediately. But the tasks themselves, if are not coded thinking in parallel architecture, are not going to be improved or speeded up. Probably they will run slower than they did in mono processor architectures.

The reason of why the tasks will run slower is easy to understand. Usually adding more cores to the same processor wants to say that each core running independently will work slower. And probably the task that we are going to execute isn't prepared for threading or parallel computing and it will run only in one core that isn't as good as one mono processor. For example, after retrieving information from Intel for the same date and the same manufacturing technology (65nm) we could do this comparison (January, 2006):

Intel Pentium 4 HT:

Cores: 1
Frequency/core: 3-3,6 GHz
Total power consumption: 65 – 85W

Intel Core Duo:

Cores: 2
Frequency/core: 1,6 GHz
Total power consumption: 34W

This means that a task coded without thinking in parallel programming will run 50% slower in a Core Duo processor. Instead, if the task is thread-capable it will run as fast as in the first processor with at least a 50% less power consumption.

The same thing happens to the GPU's. When we are using only one core independently instead of in parallel mode, we are not getting benefit from the other cores (In nVidia Tesla for example, each one runs at 600 MHz).

Nowadays the reality is a little bit different. Although the hardware vendors have been moving to multi core architectures creating the situation of a big multi core dilemma, recently, more choices have become available for the different kind of applications. But there are a lot of different points of view to deal with the dilemma and a choice may be done.

4.1. Heterogeneous vs. Homogeneous Multi-Core

Some of the new multi core hardware/software shown below will look more homogenous, like a group of CPUs. Others, instead, will look more heterogeneous, like CPUs helped by different specialized cores like GPUs. Both options will improve our present situation and are going to benefit the industry, but there are pros and cons for each that should be considered before any software development planning.

Intel and *Sun* are planning to continue HOMOGENEOUS strategy:

- *Intel's Larrabee* project is a many-core CPU strategy that they argue will reduce (or eliminate) the need to use separate GPUs and other accelerators for general-purpose computing, although it seems likely that it could bear some similarities to GPUs in the areas of floating point and vector operations. One of the main advantages of this approach is that it leverages the existing x86 instruction set. Not everyone loves the x86 instruction set, but there are certainly huge benefits to keeping the existing tool-chains (compilers, debuggers, profilers) that are already in place. Some tests have been published with multi core Xeon systems and seen very good scalability with this approach for common applications.
- Sun released Niagara2, their follow up to the industry-leading Niagara multi core server.

IBM, *AMD* and *nVidia* are taking the HETEROGENEOUS approach:

- *AMD*, with their purchase of *ATI*, added GPU hardware to their offering, and has put *"stream computing"* and *"accelerated computing"* (formerly known as *"Fusion"*) in the middle of their strategy. The vision here is basically fusion of CPU and specialized *"accelerators"* so that the hardware is more tuned to different use cases. For example, CPUs are great at time slicing and scheduling, while GPUs are great at processing math in parallel. *AMD* recently made a public embrace of the nascent *OpenCL* standard as a programming model for GPU.
- *IBM*, in partnership with *Sony* and *Toshiba*, has brought the *CellBE* processor to market. The basic idea here is similar to *AMD*'s accelerated computing: specialized hardware tuned to different use cases all on a single chip.
- *nVidia* is focused on the GPU. Focus has a lot of advantages; *nVidia* is ahead of AMD in getting GPU into the mainstream market (although this is still very early stage) and has a more robust API and tool environment with *CUDA*. Many people in both industry and academic areas have reported significant throughput increases using *nVidia* GPUs for complex compute-intensive problems that are capable of running in a massively parallel environment.

4.2. The options

At present, we have these options. As explained before some are homogeneous and others heterogeneous with their pros and cons. We'll explain features of each one as brief and detailed as possible.

4.2.1. nVidia – CUDA

nVidia is the enterprise which has taken the first position on the easy-to-obtain market. In other words, the best option if we consider quality-price ratio. We need to put the blame on video game industry. How this market has grown in these years is incredible, we can corroborate if we track the news about gaming industry. News like this one are not surprising in the last years:

"2007 U.S. Video Game And PC Game Sales Exceed $18.8 Billion Marking Third Consecutive Year Of Record-Breaking Sales".

nVidia put all effort focused in this market to develop competitive GPU devices and expanded its market giving the chance to use this innovative products in research areas. CUDA was created with the intention to monopolize these areas.

As the reader probably knows, *nVidia* is only dedicated to the development of graphic device cards putting all its competence creating innovative devices. That's a great pro for the enterprise having at present always a step ahead others. *CUDA* is the reality of years of work.

But what's *CUDA*? *CUDA* (Compute Unified Device Architecture) is a SDK created by *nVidia* to use their graphic devices for non-graphical purposes. Its strength is based on the D&C (Divide & Conquer) strategy, creating a lot of threads for running in parallel over all the cores that a device has. After doing some calculations in each thread the result is unified in what we are looking for.

In CUDA: In-depth analysis a more detailed explanation about how it works is shown.

4.2.2. AMD ATi Fusion – ATi Stream Computing

ATi Stream Computing is the response of the AMD ATi Fusion to nVidia's CUDA SDK. Although, they created the SDK first (Close-To-Metal (CTM) low-level API), the well-known one is CUDA.

They have taken with dedication the development of a useful developer kit before showing it to the world and here is the result. A SDK that works with CPU and GPU together instead of doing separately as nVidia's CUDA does, because these ones have no control over the CPU.

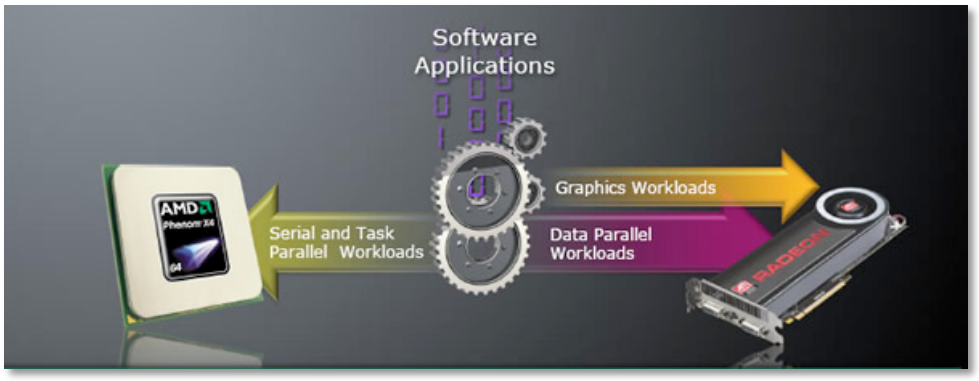


Figure 4.1: AMD/ATi Fusion environment

A presentation was done in the 2008 Q4 explaining how is going to work SDK and solved some doubts that were created during 2008.

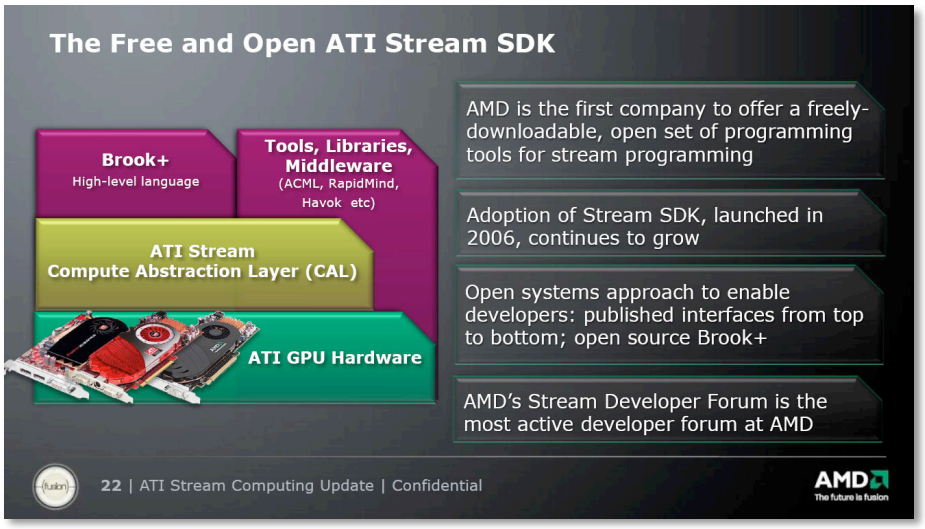


Figure 4.2: Features of Open ATi Stream SDK

First of all, at present the main language that is going to be used to code on the ATi graphic devices is a modified version of BrookGPU, supposed to be an optimized one, called Brook+ (We're going to explain it before).

This high-level language talks to the ATi Stream Layer, which is both a low-level language and an API. It's the CTM successor. Other applications and middleware also speak to this layer.

There was a bit commotion after this presentation because Brook+ was adopted as the primary tool to code on ATi cards. But in that moment, the world attended to the union between influencing enterprises shouting for an open standard adoption.

In some ways it reminds of the early days of the 3D-acceleration where we had competing 3D API's from 3dfx (Glide), Rendition (Redline) and PowerVR (VideoLogic now Imagination Technologies). The competition meant that developers had to program patches for each type of cards just to give users 3D-support. The solution at that time was a few broader standards, OpenGL and DirectX, and it looks like the solution will be similar this time. Now the standard is called OpenCL and is what the enterprises have been claiming for.

At the moment, ATi has sold more than 2 million ATi Radeon™ HD 4000 series graphic device cards. They announced that downloading a freely available package will unlock built-in ATi Stream capabilities giving the possibility, for example, to develop a ATi Radeon™ Folding@Home version that uses this tools.

4.2.3. Khronos Group – OpenCL

KHRONOS GROUP **OpenCL**

OpenCL (The Open Standard for Heterogeneous Parallel Programming) is a language created by Khronos Group during 2008. This is not true at all, because a draft of OpenCL proposal was given by Apple Inc. to Khronos Group. Here is the events chronology:

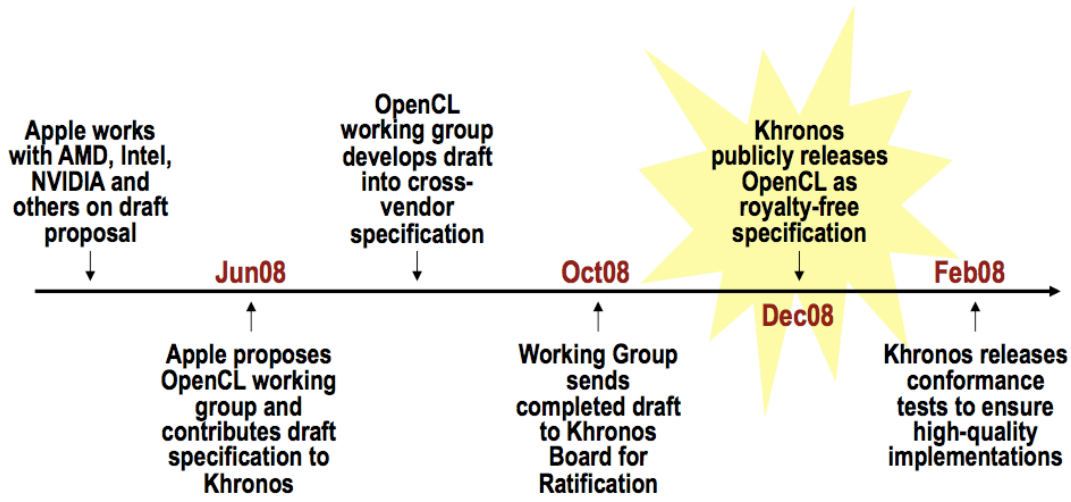


Figure 4.3: Chronology of OpenCL growing

Apple is going to include the first release of OpenCL in its upcoming Mac OS X Snow Leopard setting the trend and going a step ahead.

Its objective is to help growing the market for parallel computing for all kind of vendors of systems, silicon, middleware, tools and applications. It's open, royalty-free standard for heterogeneous parallel programming with a unified programming model for CPUs, GPUs, Cell, DSP and other processors. The next chart shows exactly how the *OpenCL* is going to fusion the worlds of CPU and GPU.

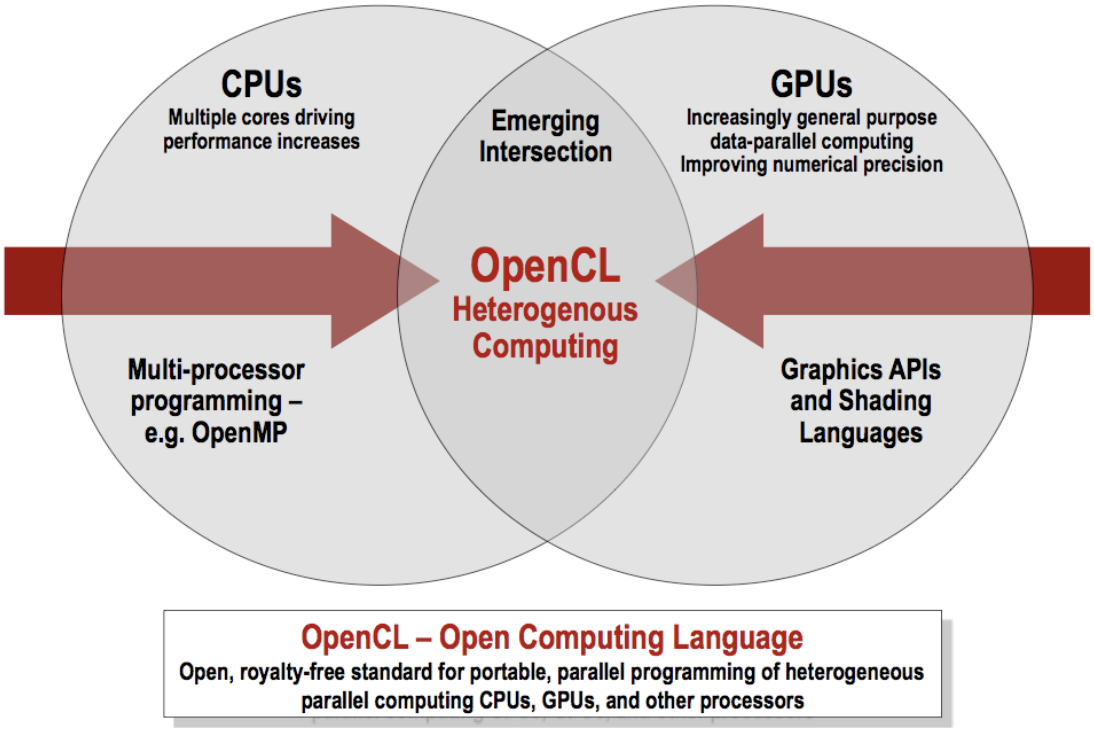


Figure 4.4: Fusion between CPU and GPU with OpenCL

Its main features are these ones:

- ***Foundation Layer:*** It's a layer that gives low-level access to create fast and efficient middleware and applications.
- ***Royalty free:*** It's open, with no cost for using the API.
- ***Cross-Vendor:*** One of the most important features, it doesn't matter if you are developing for a nVidia GeForce, ATi Radeon or IBM Cell processors, the code is going to be the same, transparent to the developer.
- ***Diverse Applications:*** It's going to be used in different kind of applications starting from embedded and mobile software through consumer applications to HPC solutions.
- ***Diverse Industry:*** Diverse applications entail to diverse industry participation like processor vendors, system OEMs, middleware vendors, application developers, etc.
- ***Rapid deployment in the market:*** Designed to run on current latest generations of GPU hardware.
- ***Focus of Graphics/Media:*** Khronos has an established focus on 2D/3D, video, imaging, audio APIs, etc.

If all these reasons are not enough, the support of many industry-leading experts and companies will confirm that this is what the sector was looking for. Below some of the companies in the *OpenCL* working group:

Figure 4.5: OpenCL sponsors

One missing company is Microsoft who plans to develop his own GPGPU managing tool inside DirectX API with a lot of new features, all included in the 11th version.

4.2.4. Stanford University – BrookGPU

BrookGPU is the *Stanford University* Graphics group's compiler and runtime implementation of the Brook stream programming language for using modern graphic device cards for non-graphical, or general-purpose computation.

It could be defined as the little unknown brother of *OpenCL* that created a programming language to unify the most important vendors at present. This is the scheme that indicates how it works:

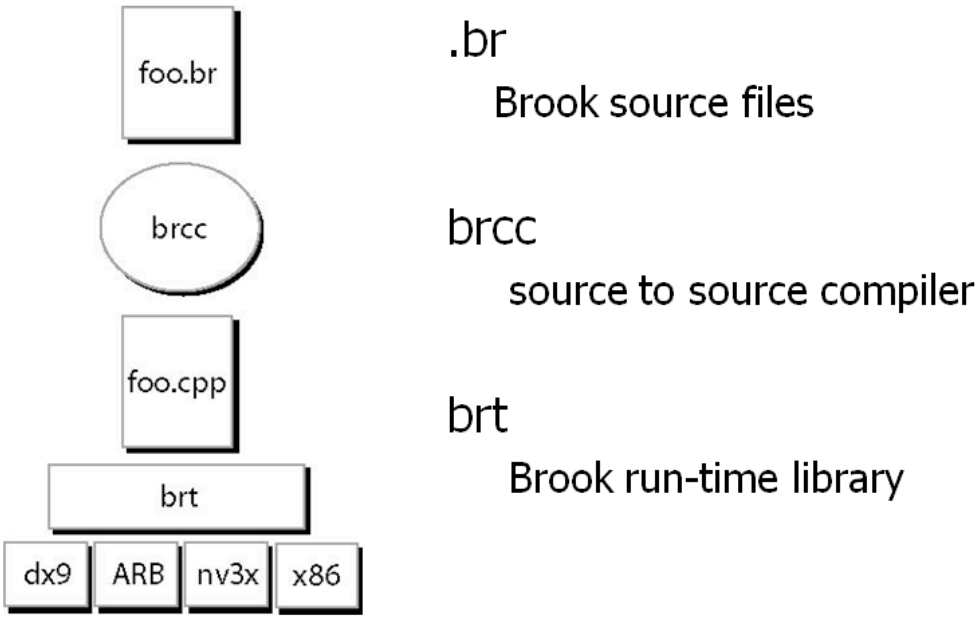


Figure 4.6: BrookGPU flow diagram

Brook is a set of extensions to the C language "C with streams" as its creators at Stanford presented it. Concretely, Brook proposed to encapsulate all the management part of the 3D API and expose the GPU as a coprocessor for parallel calculations. For this, Brook comprises a compiler, which takes a .br file containing C++ code and extensions and generates standard C++ code that will be linked to a run-time library that has various back-ends (DirectX, OpenGL ARB, OpenGL NV3x, x86).

But BrookGPU has a very big problem. One over which BrookGPU's developers had no control, compatibility. It's not strange for GPU manufacturers to improve their drivers regularly, furthermore given the heavy competition between them. While these updates are an improvement for gamers, they could

break Brook's compatibility overnight. That made it hard to imagine using the API in industrial-quality code intended for deployment.

BrookGPU has remained the attention of curious researches and programmers but is doomed to death because of not having big enterprises support.

4.2.5. Intel - 'Ct' Larrabee

Larrabee is the codename of the current Intel developing processor with which expect to get into graphic device cards market, at the moment under nVidia and AMD/ATi control in 98%.

Intel Larrabee is at present being developed but on a SIGGRAPH 2008 paper they revealed some interesting data about it. It seems that is some kind of AMD/ATi Fusion, mixing CPU and GPU in the same unit with processor that could contain, in the future, 48 cores something not surprising if we take into account the evolution of two years ago.

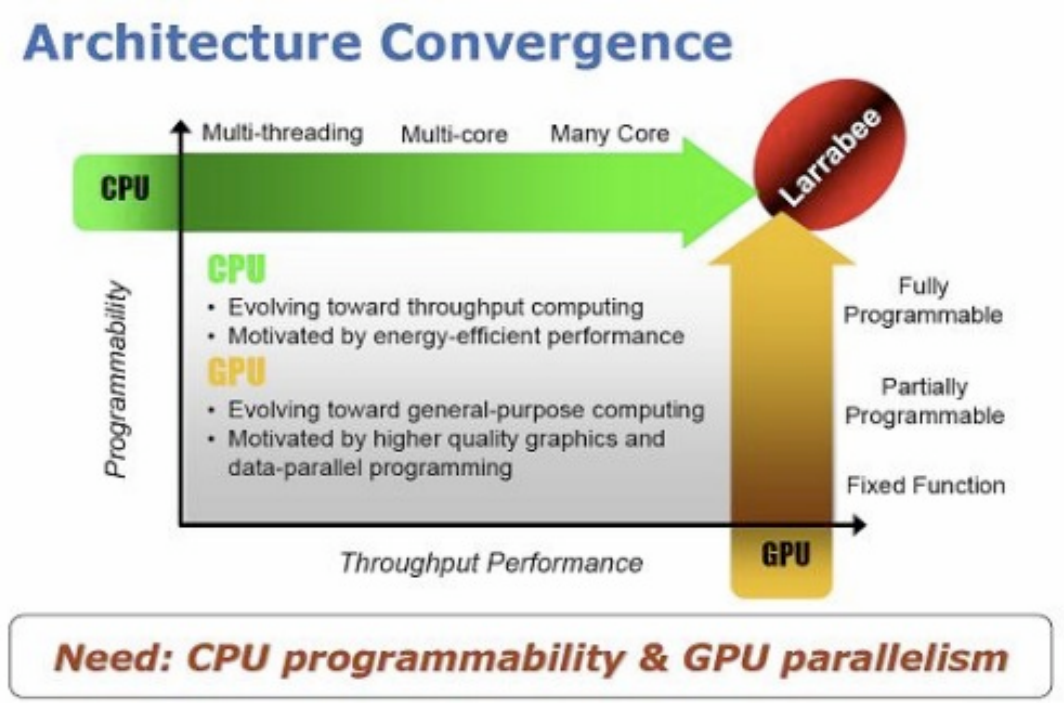


Figure 4.7: Intel's demagogy about CPU and GPU

Intel Larrabee will mean one big step ahead over current systems. Both Intel with Larrabee and AMD/ATi with Fusion are walking in the same direction unifying the two most important components of a system: CPU and GPU.

However, Intel is speaking about a long-term basis, instead of AMD/ATi Fusion, where the number of cores are of vital importance for the task distribution (Both task and graphic data) between different execution threads. On the other hand, AMD/ATi Fusion could be integrated in a short-term basis and into no so powerful microprocessors even in netbooks.

According to Intel, Larrabee is based on several Pentium processors with an addition of 64 bits instruction set and multi-threading, as well as one cache for sharing data between cores.

It will be available in the market on 2009-2010. Dates that seem early if we take into account in the situation we are. Where quad-core processors in the present market don't have too much followers (In part due to operating system developers passivity to fit these processors well in the running architecture).

Some preliminary notes had been given in the SIGGRAPH like performance data comparing the increasing of cores vs performance. Proving linear increase:

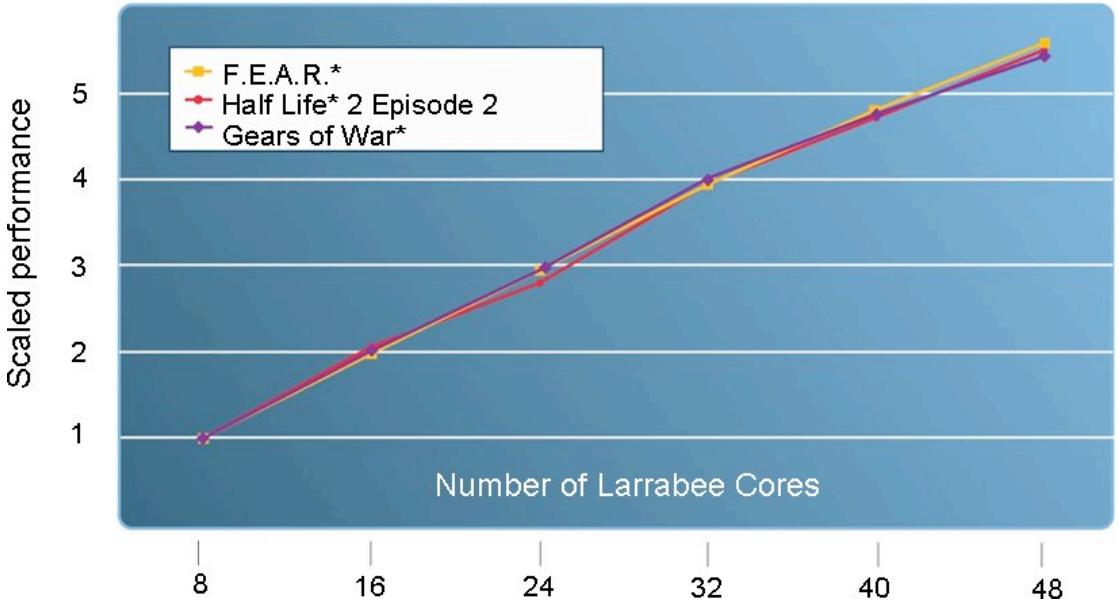


Figure 4.8: Gaming performance tests

It looks promising, but no runtime environment was given which is a suspicion reason. Although Intel is very secretive regarding which are the upcoming Larrabee's features, what is confirmed is the language they are going to use for programming on it. It's called 'Ct' (C/C++ for Throughput Computing).

In their words 'Ct' is for:

> *"One of the main challenges in scaling multi-core for the future is that of migrating programming tools, build environments, and millions of lines of existing code to new parallel programming models or compilers. To help this transition, Intel researchers are developing "Ct," or C/C++ for Throughput Computing."*

And how is going to be done? Creating a library-like solution in the same way as BrookGPU does. There is a first step parsing to analyze if 'Ct' code exists. This is done using special Ct-based parallel data types. After that the code identified by 'Ct' is used for parallelizing it. For example we have this:

```
TVEC<F64> A,expv,product,SMVP;
A       = 5;
expv    = [1 2 3 4 5];
```

```
product = A*expv; // product = [5 10 15 20 25]
SMVP    = addReduce(product) // SMVP = 5+10+15+20+25 = 75
```

This is the simplest example that can be written. But teaches how is the idea of 'Ct' to cover the most basic part of a programming language (data types) and try to parallelize the operations between these.

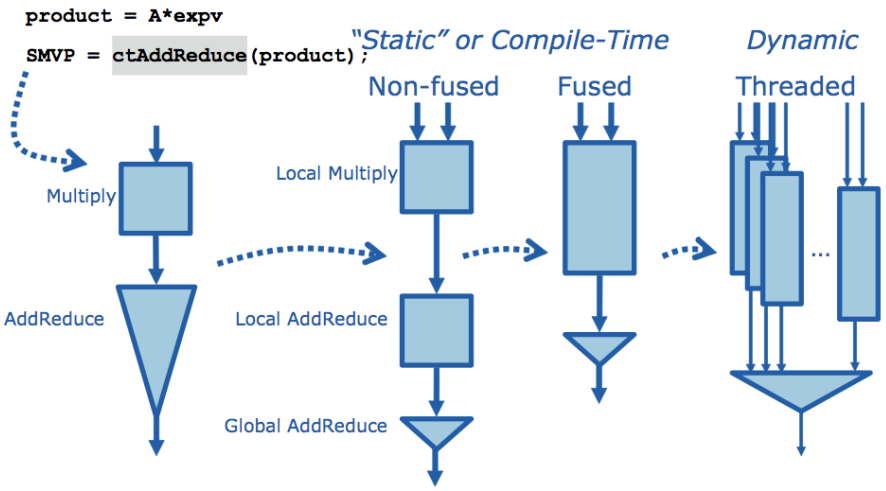


Figure 4.9: 'Ct' parallelizing method

Below a figure taken from 'Ct' whitepaper is shown which simplifies the flow diagram of a program parsing, compiling/linking and execution:

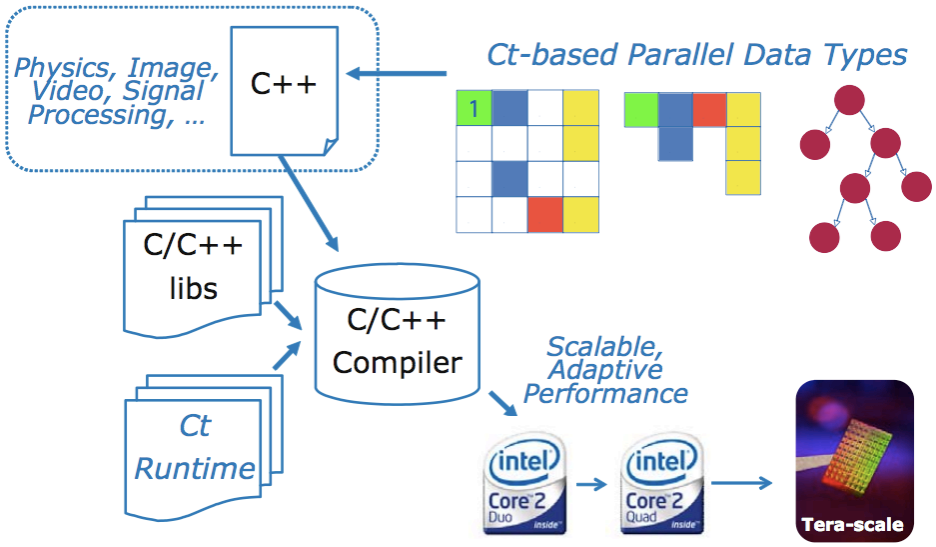


Figure 4.10: 'Ct' flow diagram

And finally a performance test using 'Ct' in Intel Xeon processor E5345 platform (two 2.33GHz quad-core processors, 4GB memory):

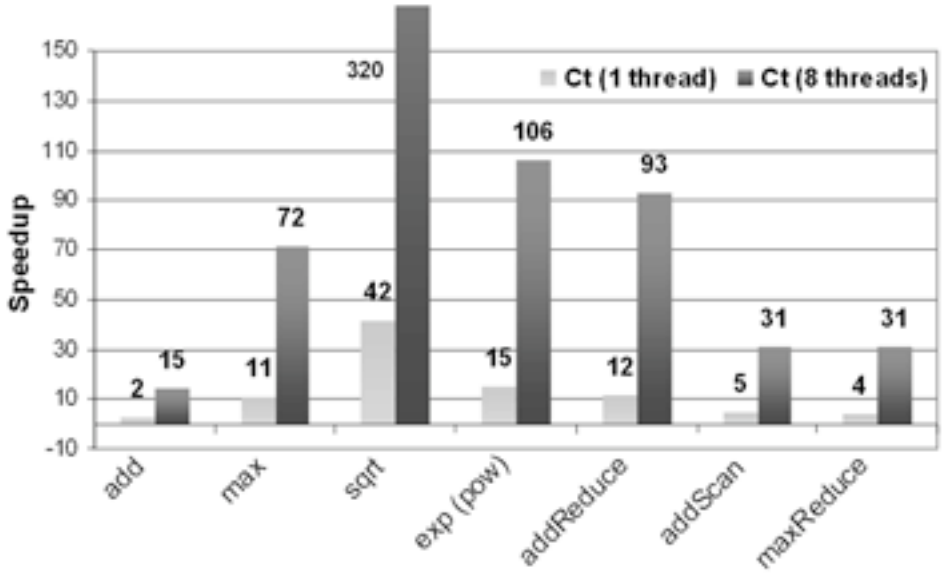


Figure 4.11: Basic operations performance tests

If we calculate the speedup of each task dividing the multi-threaded by the single-threaded we obtain an almost uniform random parameter between [6,8]. This means that if we increase the number of cores the performance is increased almost linearly.

4.2.6. IBM – CellBE

SONY TOSHIBA IBM **Cell BE**

Cell is a microprocessor architecture developed jointly by Sony, Toshiba and IBM an alliance known as "STI". It started on 2001 and needed four developing years for having a working prototype with 400 employees working around the world and US$400 million.

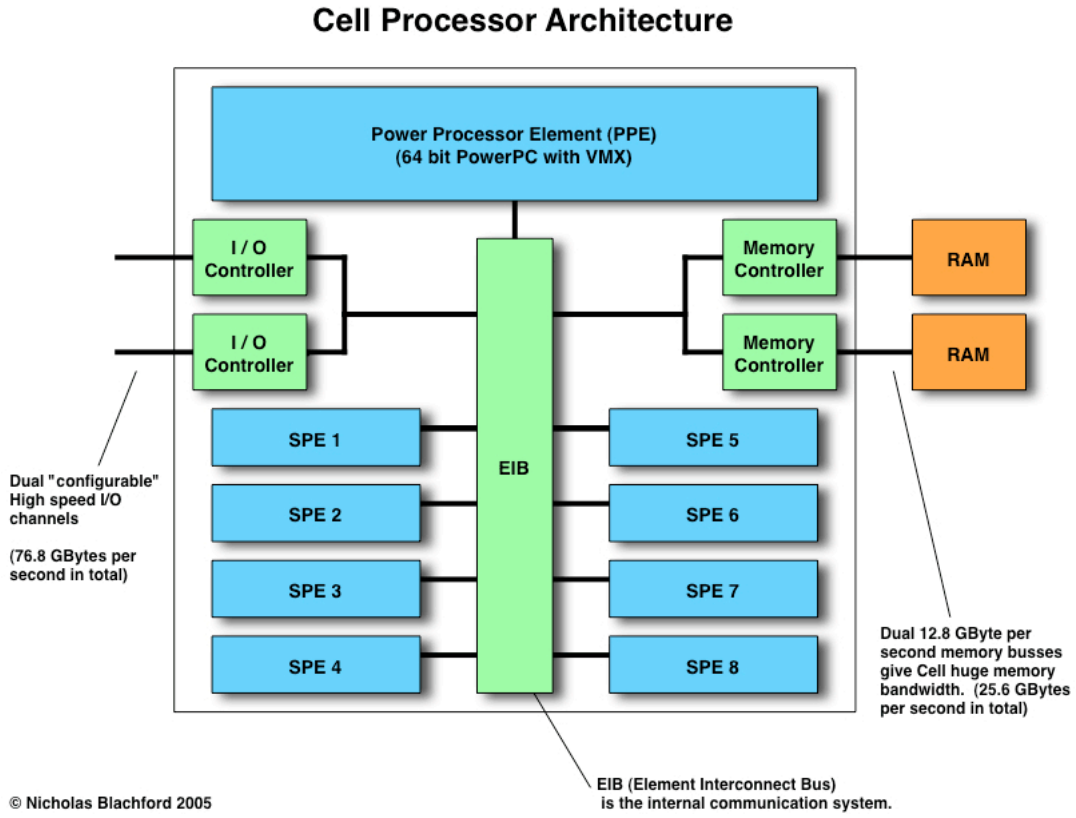


Figure 4.12: Cell BE processor

The current chip is composed of one 64-bit PowerPC Processing Element (PPE) and 8 specialized co-processors called Synergistic Processing Elements (SPE). The PPE and SPEs are linked together by an internal high speed bus called Element Interconnect Bus (EIB).

The PowerPC Processing Element (PPE) follows the 64-bit PowerPC AS architecture, as the PowerPC 970 CPU (also known as the G5) and all recent IBM POWER™ processors also use. Like the 970, it can use the VMX (AltiVec) vector instructions to parallelize arithmetic operations.

The SPEs are composed of a Synergistic Processing Unit (SPU), and a SMF unit (DMA, MMU, and bus interface). A SPE is a RISC processor with 128-bit SIMD

organization for single and double precision instructions. Each SPE contains a 256 KB instruction and data local memory area (called local store) which is visible to the PPE and can be addressed directly by software. The local store does not operate like a superscalar CPU cache since it is neither transparent to software nor does it contain hardware structures that predict what data to load.

The EIB is a circular bus made of two channels in opposite directions each. It enables communication between the PPE and SPEs. It is also connected to the L2 cache, the memory controller, and the FlexIO for external communications.

Nowadays, Cell BE is at least as famous as nVidia CUDA capable graphic device cards for parallel application development. But they have different influencing area. Cell BE is more focused in industrial area composed by HPC (High Performance Computing), upcoming Video/Audio systems, powerful consoles, where has the sector won. nVidia instead is the leader in personal computing (and now its getting known by researches, universities, etc.) and is migrating its useful SDK to Home HPC world creating different kind of systems based on Tesla cards.

A fast comparison could be done between these two supercomputing giants:

	Magnetar TDX Pro (nVidia Tesla HPC)	**PS3**
Price	11.100 €	465 €
Single Precision	2.8 TeraFLOPS	256 GigaFLOPS
Double Precision	230 GigaFLOPS	25 GigaFLOPS

Table 4.1: Comparison between nVidia Tesla and Sony PS3

Spending the same quantity of money buying one nVidia Tesla HPC we could buy (11.100/465) = 23.8 PS3s. Knowing that the theoretical GigaFLOPS increases linearly, with the same quantity of money PS3 arrives to 256*23.8 = 6.111 GigaFLOPS. **2.18** times faster than nVidia Tesla HPC in single precision computation and **2.58** times faster in double precision.

To get this HPC monster working, IBM released an SDK that is continuously updating and has a lot of useful libraries as well as a great amount and variety of documentation.

This release helped to independent groups to create their own supercomputer (Universitat Politecnica de Catalunya is migrating to PowerXCell, an evolution of Cell BE, based systems). Consequently, these groups (Universities, HPCs, Independent PS3 users, etc.) are contributing to the Cell community and it is growing fast with a lot of software progresses, like CellSs (Cell Superscalar).

.

5. CUDA: In-depth analysis

5.1. Brief description

As explained in the background of this thesis, CUDA is a SDK developed by nVidia to let developers the possibility of programming into CUDA-capable nVidia devices. A parallel programming model and software environment designed to overcome the challenge of transparently scale its parallelism to leverage the increasing number of processor cores, while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

At its engine there are three main abstractions, hierarchy of thread groups, shared memories and synchronization between threads. To use all of them together CUDA gives a minimal set of C extensions.

But not all that glitters is gold. CUDA provides the software development kit for create our parallel made applications but doesn't magically convert our sequential thinking habit into optimized code. Although this SDK helps a lot and reduces dramatically the learning curve, we need to become accustomed with parallel code deployment.

Once we think up how to create our application for be able to run in a parallel architecture the benefits came alone. We can see the peak of GigaFLOPS obtained with one nVidia GPU comparing with an Intel CPU:

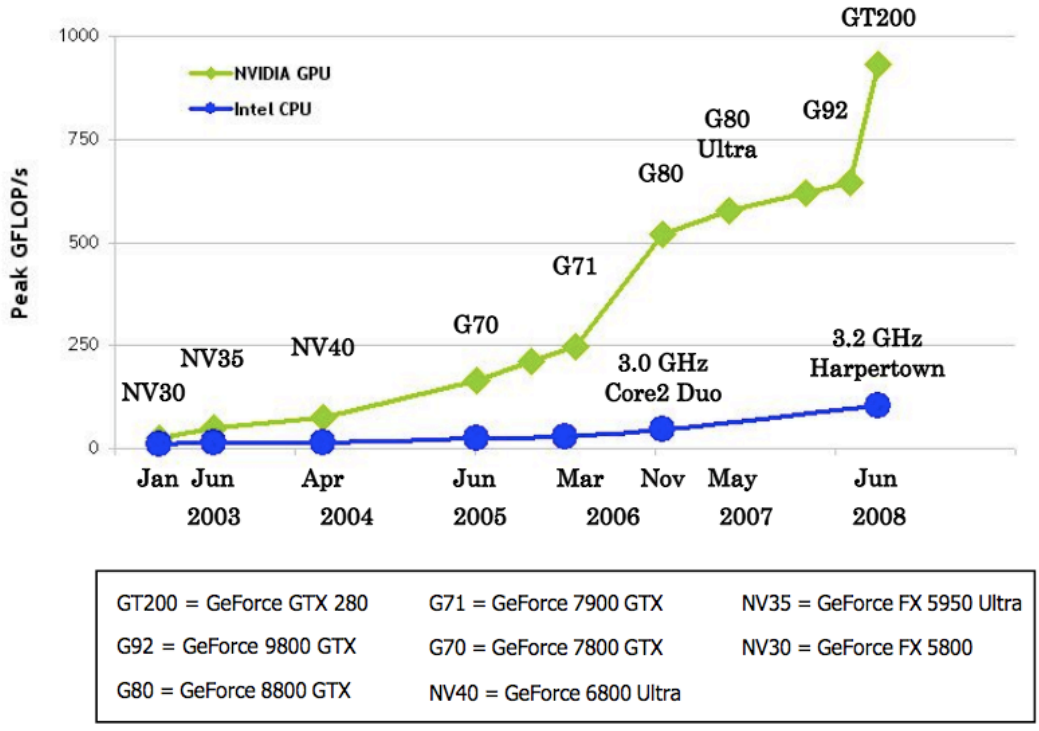


Figure 5.1: Chronology of CPU and GPU improvements

Why weren't they handicapped in the same way as their rivals who design CPUs? The reason is very simple: CPUs are designed to get maximum performance from a stream of instructions, which operates on diverse data (such as integers and floating-point calculations) and performs random memory accesses, branching, etc. Up to that point, architects were working to extract more parallelism of instructions – that is, to launch as many instructions as possible in parallel. Accordingly, the Pentium introduced superscalar execution, making it possible to launch two instructions per cycle under certain conditions. The Pentium Pro ushered in out-of-order execution of instructions in order to make optimum use of calculating units. The problem is that there's a limit to the parallelism that is possible to get out of a sequential stream of instructions, and consequently, blindly increasing the number of calculating units is useless, since they remain unused most of the time.

In the other hand, the operation of a GPU is too simple. The job consists of taking a group of polygons, on the one hand, and generating a group of pixels on the other. The polygons and pixels are independent of each other, and so can be processed by parallel units. That means that a GPU can afford to devote a large part of its die to calculating units which, unlike those of a CPU, will actually be used.

The difference is shown in the next figure:

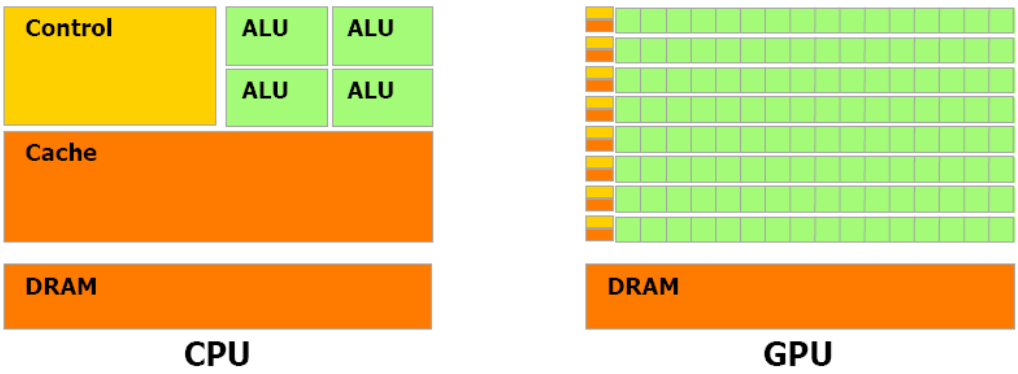


Figure 5.2: Difference between CPU and GPU architecture

5.2. Programming Model

CUDA as all novel SDKs has created its own programming model based on the requirements that this kind of SDK should has. Normally, the hardware itself imposes the restrictions and the needs of a SDK. As shown in the last figure, the GPU is composed with a lot of ALU's prepared to process some graphic data. To manage these ALUs uses threads that can be coded for our own purposes.

But the threads are organized in one way that it is easy for the device to launch and maintain under control. In the same manner, these are arranged in one

way that makes the life of a 3D programmer easier, creating the concept of grids and blocks.

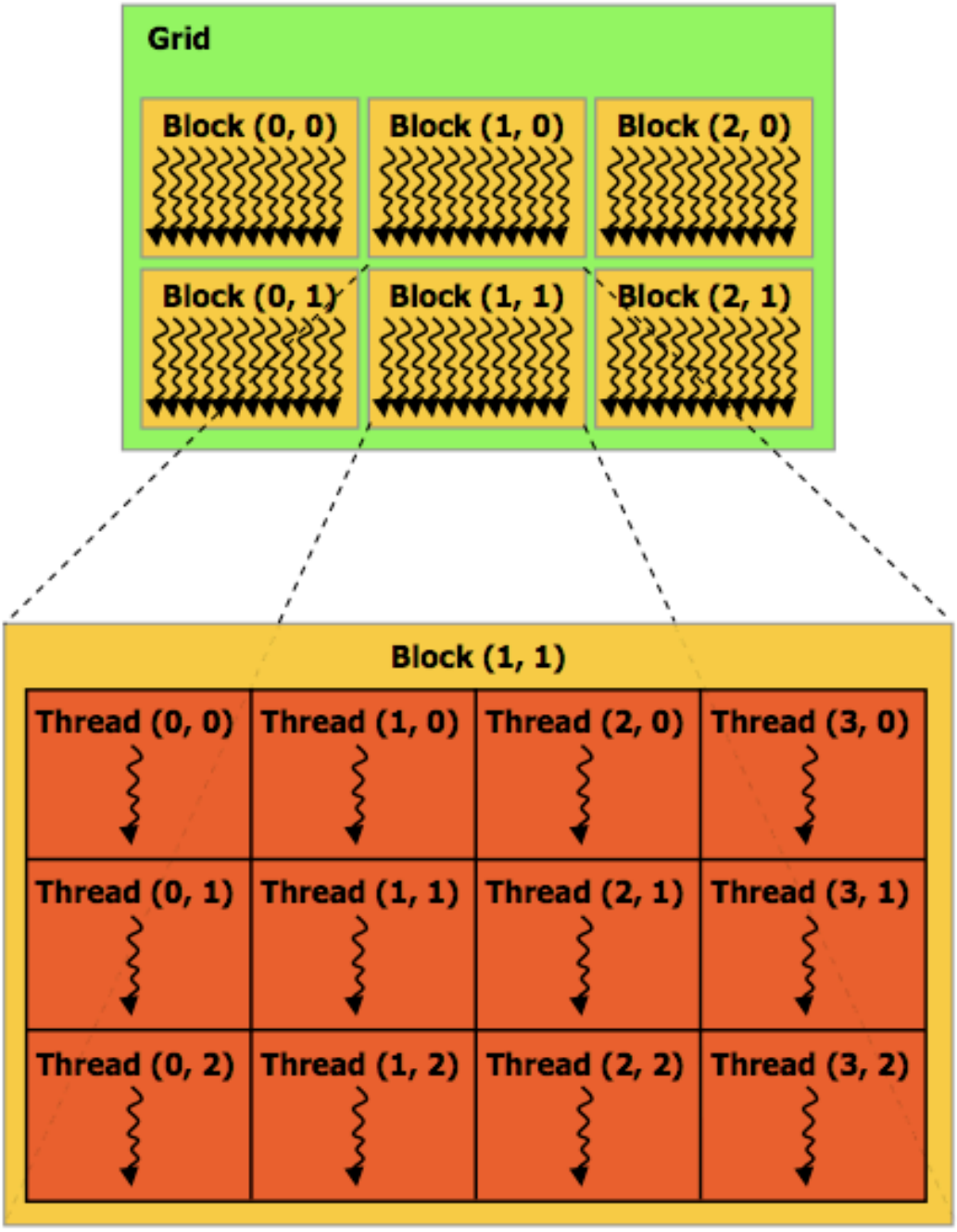


Figure 5.3: nVidia device grid composition

In the last figure we can identify the thread hierarchy. First it is the *Grid* a two-dimensional parameter containing blocks inside. Then we have *Blocks*, a three-dimensional parameter in which we have a defined number of threads. Finally, the *Threads* themselves where we put the tasks we want to launch. The threads are a three-dimensional parameters too. The reason why is arranged in a three-dimensional way is because, in the first implementation of the SDK, was

thought to be running two/three dimensional application and an abstraction which helps was needed.

How we put threads doing some job? For that was implemented the kernel. The kernel is similar as a function is in C, which helps launching a task to the GPU with an appropriate configuration. An easy way to understand:

```
dim3 numBlocks(1,1,1);
dim3 numThreads(1,1,1);
uint sharedMemsize = 2048;
myKernelTask<<< numBlocks,numThreads,sharedMemsize>>>(arg1,arg2,...);
```

A lot of kernels could be coded in our C program but only one of them is going to be using the GPU at the same time. This implies a queue to be implemented in the GPU card to maintain the kernel execution order as they were inserted in the queue. The queue is a FIFO (First In First Out) based one, it's up to the developer create a priority based algorithm if needed.

What the parameter `sharedMemsize` really does is going to be explained in the memory model section.

5.2.1. Host and Device

CUDA assumes that the threads may execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case when the kernels execute on a GPU and the rest of the C program continues executing on a CPU.

It also assumes that both the host and the device maintain their own DRAM referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime environment. This includes device memory allocation and deallocation, as well as data transfer between host and device memory.

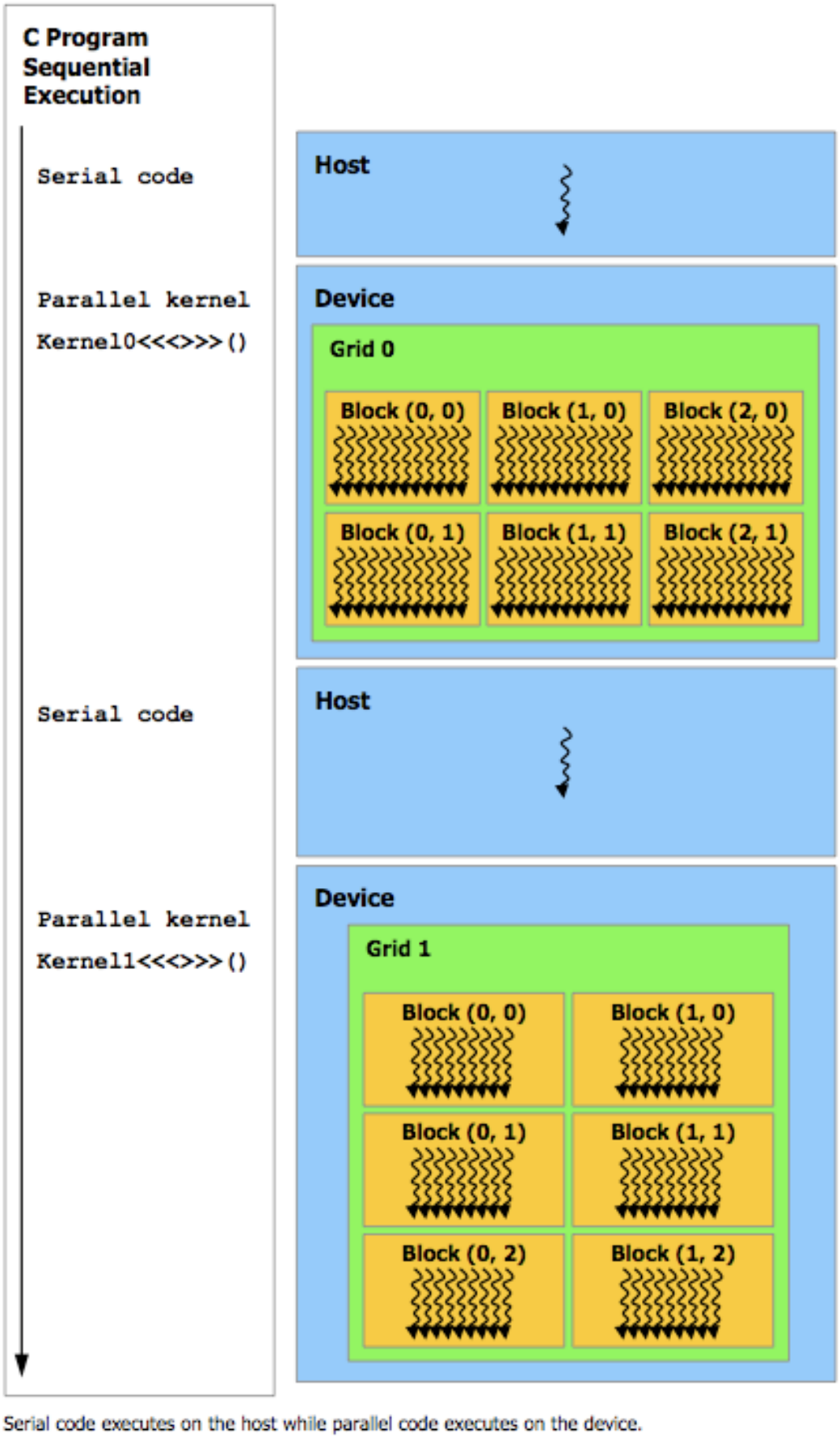


Figure 5.4: nVidia CUDA execution flow

5.2.2. Memory model

There are several types of memory that a developer can use. Each one has its pros and cons. Lets analyze through the next figure:

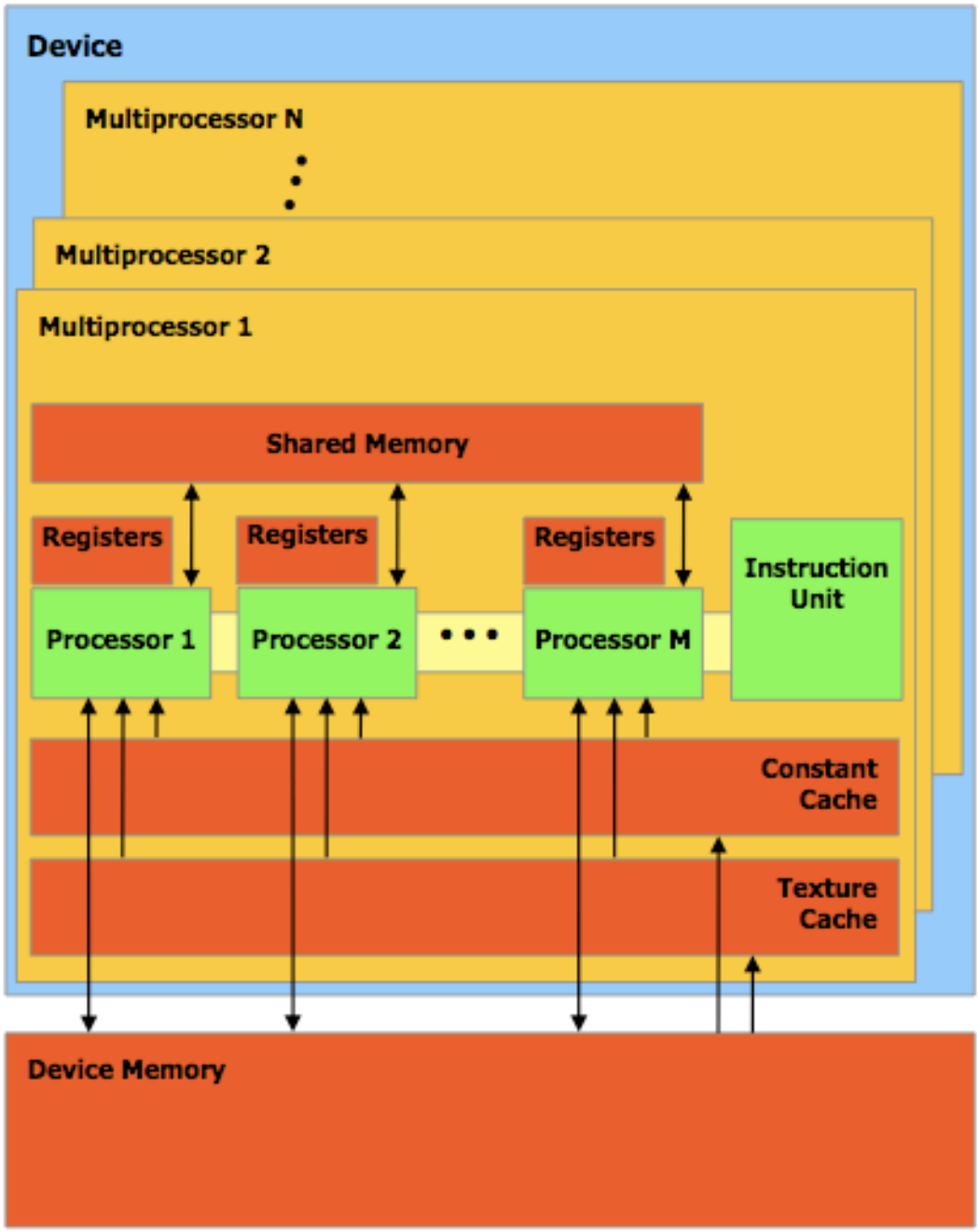


Figure 5.5: nVidia device architecture in depth

First, we can see that there is a *Device Memory* usually called as global memory. This is shared by all the threads and we need to take account how our kernel is getting information from it. Because it could be extremely slow if more than one thread are trying to access to a specific memory address at the same time. This memory it's readable and writable by all the running threads.

Shared Memory is like *Device Memory* but it differs in the quantity of threads that could get access to it. It is deductible with the figure up, that *Shared Memory* is only shared by the threads in the multiprocessor. Namely, its changes could only be seen by the threads within the multiprocessor (within the block when we configure our kernel) and is used for the communication between threads. As well as global memory, shared memory is readable and writable by all the threads within the block.

Before the explanation of another type of memory we need to make one thing clear. There is a big confusion between what is local memory and registers. In the nVidia forums usually when people is talking about registers want to say local memory, and vice versa. But they are not the same.

Local Memory is an abstraction made by CUDA to be able to differentiate between global memory and local memory of each thread, but in essence, they reside in the same memory place (global memory). It was created to fulfil as well as possible the needs of the developer.

Instead, *Registers* are memories owned by each thread. There aren't so much but they help getting our application faster. Normally, automatic variables declared in a kernel reside in registers, which provide very fast access. In some cases the compiler might choose to place these variables in local memory, which might be the case when:

- There are too many register variables
- An array contains more than four elements
- Some structure or array would consume too much register space
- When the compiler cannot determine if an array is indeed with constant quantities.

Finally, there are two read only address spaces that are vestiges of the graphical nature of the GPU – the constant and texture memories.

Constant Memory can be filled up with constant values before launching a kernel and it cannot be modified during the execution. It is like the *#define* statement in C that cannot be modified after it have been pre-processed.

The *Texture Cache* is a place where the 2D textures are put there to have a fast access to them. It's vastly larger than *Constant Memory* and has two-dimensional locality (traditional caches have locality in a single dimension). It is not our interest when we are developing not-OpenGL/DirectX kind of applications.

But there are big differences between memories as we can see in the next table:

Name	Size	Speed	Type	Alloc. Place
Registers	8192/ MP	Fast	R/W	MP
Shared Memory	16 Kb	Fast	R/W	MP
Local Memory	Device's Mem.	Slow	R/W	DRAM

Constant Memory	64 Kb	Fast	R	DRAM
Texture Cache	16 Kb/MP	Fast	R	DRAM
Global Memory	Device's Mem.	Slow	R/W	DRAM

MP = MultiProcessor (This values could change with device model)

Table 5.1: Memory types comparison

5.3. CUDA API

The CUDA API encompasses a whole ecosystem of software. Its hear is the CUDA C which is compiled with nvcc (nVidia compiler, based on Open64 back-end). CUDA is not C, is a variant of C extensions.

To be able of executing CUDA-kind code it's needed the installation of CUDA driver, which is now included in all nVidia graphic drivers. The CUDA run-time, which is a dynamic compiler (JiT – Just in Time compiler, yes! Based on Toyota method) that can target the underlying hardware, is an optional component. Finally, the API includes math libraries, FFT (Fast Fourier Transform), BLAS (Basic Linear Algebra Subprograms) and DPP (Data Parallel Primitives). Again, these are optional.

nvcc the most important part of the coding process, can output three different targets:

- *PTX: Parallel Thread eXecution*
- *CUDA binaries*
- *Standard C*

PTX is a virtual instruction set designed as an input set for the dynamic compiler. The CUDA run-time layer (JiT) converts, compiling the *PTX* file into native operations for whatever it is the GPU hardware a user has installed. The loveliness of this approach is that it's thought for the upcoming years, keeping in mind the importance of backwards compatibility, longevity, scalability and high performance.

PTX running in the JiT layer is obviously the best approach to get the highest performance. Some software companies prefer to give up some performance in exchange for easy to validate behaviour. To fulfil the needs of this area nvcc could deliver a *CUDA binary* file and avoid the dynamic compilar of weird behaviour. The user who uses *.cubin* files is tied up to execute in a specific environment, that is to say, a concrete GPU and nVidia driver version.

Finally, there is the option of getting a *Standard C* output. This could be redirected to the actual well-known compilers like Intel (icc), GNU (gcc). What's the benefit of doing this? It's easy to understand. When programming CUDA compatible programs, code is made in the way it's easy to optimize. It can certainly be assured that it will considerably improve the scalability for

multi core CPUs. In the documentation is showed an improvement of 4x speed up.

5.3.1. Compute Capabilites

As explained before, CUDA was designed thinking in the longevity of the released packages. The capabilities of CUDA devices are described as a revision number that is closely linked with the version of CUDA that could execute. The first digit indicates the core architecture and the second digit indicates more negligible improvements.

On July 2006 CUDA released its first SDK version, 0.2 Beta. In this time there have been 3 minor revisions each trending to more general purpose functionality:

- **1.1:**
  - Atomic functions operating on 32 bit words in global memory.
- **1.2:**
  - Atomic functions operating on 32 bit words in shared memory.
  - Atomic functions operating on 64 bit words in global memory.
  - Two new warp voting functions
  - Support for the GT200 micro-architecture.
- **1.3:**
  - Support for double precision floating point values.

Compute capability version is not related to the CUDA version. At present, nVidia has released CUDA 2.0 but there is no compute capabilities update. For example, GeForce GTX 280, 260 and Tesla S1070, C1060 are the only with compute capability 1.3. There are not compute capability 1.2 devices.

The lack of Compute 1.2 devices today seems to indicate that at least one future GPU will omit double precision floating point to reduce costs.

5.4. Example

The simplest example but the most didactic is the next one that shows in a simple manner how to structure the code and the sections that really matter.

```
#include <stdio.h>
#include <assert.h>

#define NUMTHREADSPERBLOCK 10

__constant__ int kte_dev [NUMTHREADSPERBLOCK];
             int kte_host[NUMTHREADSPERBLOCK];

void __device__ addFunc(int idx, int *data) {
    data[idx] += kte_dev[idx];
}
```

```
__global__ void myKernel(int *deviceVar) {
    extern __shared__ int s_data[];
    // This is not necessary because we have only one block full of
    // threads but it's illustrative (blockDim.x = numThreadsPerBlock)
    // (blockDim.x = numThreadsPerBlock)*(blockIdx.x = 0) = 0
    int idx = blockDim.x *blockIdx.x + threadIdx.x;

    // Transfer data to shared memory
    s_data[idx] = deviceVar[idx];
    // Execute the addition function
    addFunc(idx, s_data);
    // Transfer data from shared memory
    deviceVar[idx] = s_data[idx];
}

int main(int argc, char **argv) {
    // Allocated in the host memory
    int *hostVar;
    int numThreadsPerBlock = NUMTHREADSPERBLOCK;
    int memSize = numThreadsPerBlock * sizeof(int);

    // Initialize and copy it to Constant Cache
    for(int i = 0; i < numThreadsPerBlock; i++) {
        kte_host[i] = i;
    }
    cudaMemcpyToSymbol(kte_dev, kte_host, sizeof(kte_host));
    // Allocated in the device memory
    int *deviceVar;
    // Alloc memory in host
    hostVar = (int *)malloc(memSize);
    cudaMalloc((void **)&deviceVar, memSize);
    // Fill with 0's
    memset(hostVar,0,memSize);
    // Copy it to device's memory
    cudaMemcpy(deviceVar, hostVar, memSize, cudaMemcpyHostToDevice);
    // Launch kernel 1 block NUMTHREADSPERBLOCK
    myKernel<<1,numThreadsPerBlock,memSize>>(deviceVar);
    // Wait until process it's finished
    // This is not necessary because the cudaMemcpy below is a blocking
    // function which will stop running the CPU task until deviceVar is
    // unblocked
    cudaThreadSynchronize();
    // Copy it back to host's memory
    cudaMemcpy(hostVar, deviceVar, memSize, cudaMemcpyDeviceToHost);
    // Test if something was wrong
    for(int i = 0; i < numThreadsPerBlock ; i++) {
        assert(hostVar[i] == i);
    }
    // Free host's memory
    free(hostVar);
}
```

6. Blue Midnight Wish

6.1. Brief explanation

BMW (BLUE MIDNIGHT WISH) was presented on October 2008 as a candidate for SHA-3 NIST (National Institute of Standards and Technology) hash competition.

This hash competition was thought in the same way as AES was, with a detailed timeline and a lot of deadlines before deciding which is going to be the next standard hash algorithm. In the appendix, you can find the SHA-3 hash competition timeline where the green blocks mean that the stage has been completed. This list is provisional and is subject to change at NIST's discretion.

Like a lot of algorithm developments, BMW has gone through a lot of changes during its creation timeline. It's based on Turbo SHA-2, predecessor of BMW, and has four different output sizes: 224, 256, 384, 512 bits.

Algortihm	Message size l (in bits)	Block size m (in bits)	Word size w (in bits)	Endianess	Digest size n (in bits)	Support of "one-pass" streaming mode
BMW224	$< 2^{64}$	512	32	Little-endian	224	Yes
BMW256	$< 2^{64}$	512	32	Little-endian	256	Yes
BMW384	$< 2^{64}$	1024	64	Litlle-endian	384	Yes
BMW512	$< 2^{64}$	1024	64	Little-endian	512	Yes

Table 6.1: Basic properties of all four variants of the BLUE MIDNIGHT WISH

Our purpose isn't to explain how BMW works but it's advisable to have a global idea which will help in some subsequent descriptions. BMW follows the general design patterns with three important stages in its cryptographic algorithm process:

- Pre-processing.
  - Pad the message M.
  - Parse the padded message into N, m-bit message blocks, $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$
  - Set initial values.
- Hash computation recursively (shown in figures below)
- Output the hash taking the least *n* significant bits.

A more detailed vision could be taken in the figure below:

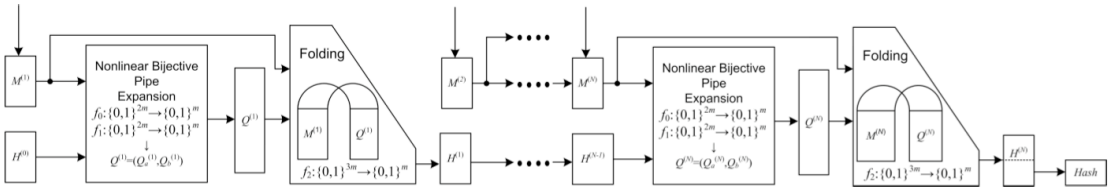


Figure 6.1: A graphic representation of BLUE MIDNIGHT WISH hash algorithm

A different point of view of the same description could also be seen in the figure below:

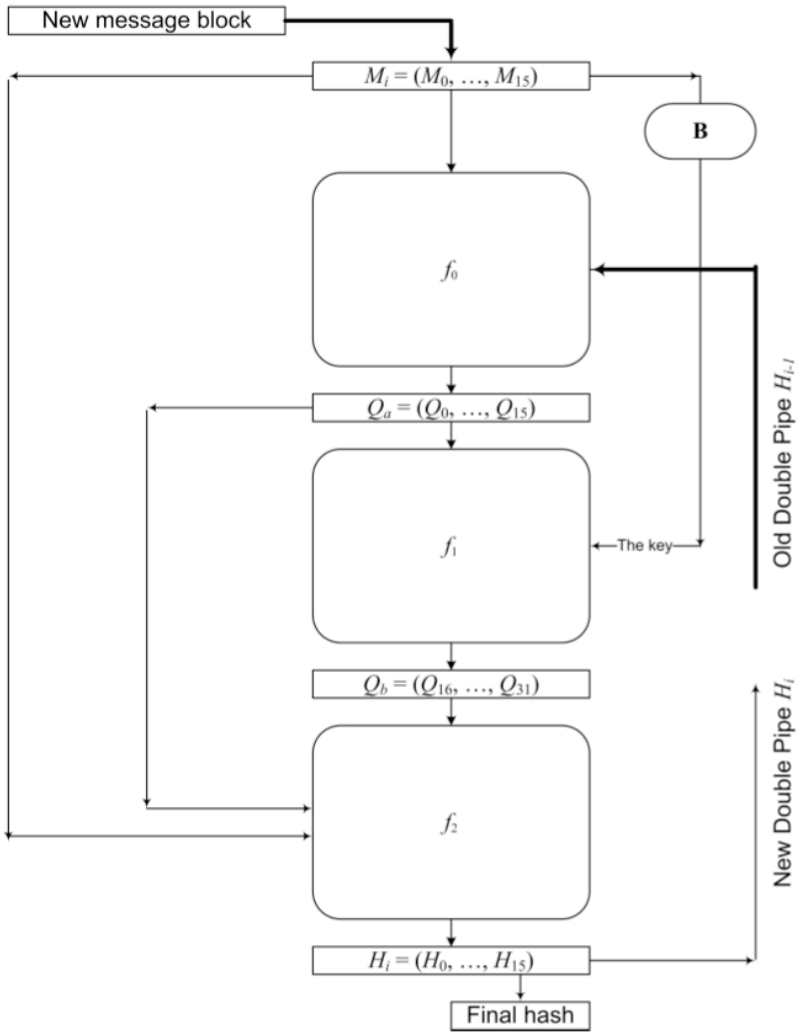


Figure 6.2: Graphical representation of the compression function in BLUE MIDNIGHT WISH

As it can be observed it is a recursive algorithm that depends with the last double pipe calculated before. The first double pipe is defined in the pre-processing stage with static values in an incremental way.

The most important parts are the blocks defined as f_0, f_1 and f_2. Without going into BMW in depth the blocks convert and reduce the ingoing message using a lot of non-linear mathematical procedures via low-cost bit operations. Finally, we get a "final hash" that is a modified version of the initial value using message blocks as changing parameter.

6.2. Identifying parallel blocks

Our purpose is to use CUDA and try to implement BMW on it. To reach this aim first we need to identify which parts are those that could be parallelized. This work is done in the paper submitted to NIST (Section 4.5: Internal Parallelizability of BLUE MIDNIGHT WISH).

The compression function (f_0, f_1 and f_2) allows a very high level of parallelization. This can be achieved with upcoming multicore systems, both CPU and GPU. Nowadays, a feasible option is only the GPU because they have at least the quantity of threads that the compression function needs.

The process consists in three main parts but not all of them can be parallelized. Theoretically all of them could be parallelized but f_1 is more a sequential calculation method instead of parallel one. It was decided to consider only the options involving to f_0 and f_2.

From the NIST submitted paper we have these identified parallelizable points:

- **Computing f_0:**
  - **Step 1:** Computation of all 16 parts of W_0, W_1, …, W_{15}, in the $_i$th iteration.
  - **Step 2:** Computing the values of all 16 parts of Q_a.
- **Computing f_1:**
  - **Step 1:** It has 16 expansion steps and each step depends from the previous one. But every expansion step have an internal structure that can be parallelized, and a pipelined setup can compute parts from the next expansion steps that do not depend on the previous expansion value.
- **Computing f_2:**
  - **Step 1:** This step can be computed together with the computation of Step 1 of the function f_1.
  - **Step 2 (First half):** Computation of the first 8 words H_0, H_1, …, H_8, in the $_i$th iteration.
  - **Step 2 (Second half):** Computation of the last 8 words H_8, H_9, …, H_{15}, in the $_i$th iteration.

Knowing the internal working flow or pipeline of CUDA, as explained before, it was decided to don't implement some parallel parts to get the best performance with nVidia graphic device cards.

Another decision was taken with the last computation (f_2), where instead of launching 8 computations in each moment, it was decided to launch 16 doing some changes in the calculation order but obtaining the same result.

In the appendix, it's framed by a red box the parts that involves to the f_0 and f_2 calculations. How many threads are launched in each step are also shown in the figures.

6.3. Implementation

Knowing how to proceed using the methodology planned in the introduction, two different proposals were designed. The significant difference between these two was how to do the compression procedure. Lets analyze the two designs.

6.3.1. Design #1

We known that the hash obtained from a message before getting the final output needs to split the message into small blocks. Using a loop and recursively with the last compression procedure calculation we update the output at each time and after the last calculation we get the hash.

In this design, the loop is coded inside an unique CUDA kernel. This design includes the f_0, f_1 and f_2 calculation in the same kernel. As said in the section "Identifying parallel blocks" f_0 and f_1 can be calculated paralelly but what about f_2? Because of its sequential behaviour it is calculated with only one CUDA thread that slows down the calculation a lot (in this case slows down f_{CPU}/f_{CORE} = 3Ghz/550Mhz = 5.45 times). As summary below the pros and cons:

Pros	Cons
Negligible memory transfer latency because it's done only once	If a huge message is sent to the GPU could crash because of memory overflow
It is called the kernel only once, avoiding a lot of kernel call delays	The sequential part slows down the compression procedure at least 5.45 times

Table 6.2: Design #1 pros and cons

6.3.2. Design #2

Two kernels were created. The first one calculates f_0 block and the second f_2. Each loop iteration compression procedure is called. This means that at each iteration of a piece of message block needs to be transferred to the device memory as well as kernel calling. This happens two times for each parallel block (f_0 and f_2).

Pros	Cons
All the calculations are done in parallel	At each loop iteration memory needs to be transferred to device.
It avoids the f_1 sequential block	At each loop iteration two kernels needs to be called.

Table 6.3: Design #2 pros and cons

6.3.3. How-To

The two designs were implemented but only one is added to the appendix because of the difference in performance between two proposals. The first design was much slower than the second one and all the efforts were focused in the multi-kernel solution. A shortened version of the GPU working compression function is the next:

```
#define Compression256()\
{\
    cudaMemcpy(data32_dev, data32, 16*sizeof(u_int32_t), cudaMemcpyHostToDevice);\
    cudaMemcpy(p256_dev, p256, 16*sizeof(u_int32_t), cudaMemcpyHostToDevice);\
    kernel_f0<<<1,16>>>(data32_dev, p256_dev);\
    cudaMemcpy(p256, p256_dev, 32*sizeof(u_int32_t), cudaMemcpyDeviceToHost);\

    […] // f1 sequential block

    cudaMemcpy(data32_dev, data32, 16*sizeof(u_int32_t), cudaMemcpyHostToDevice);\
    cudaMemcpy(p256_dev, p256, 16*sizeof(u_int32_t), cudaMemcpyHostToDevice);\
    kernel_f2<<<1,16>>>(XL32, XH32, data32_dev, p256_dev);\
    cudaMemcpy(p256, p256_dev, 32*sizeof(u_int32_t), cudaMemcpyDeviceToHost);\
}
```

To implement this multi-kernel proposal, two different kernels were needed as it can be seen in the code above. The code below are the implemented kernel bodies.

- **Kernel 1: Calculation of f_0:**

```
__global__ void kernel_f0(u_int32_t *data32, u_int32_t *p256) {
    u_int32_t thread = threadIdx.x;
    u_int32_t idx=thread*5,sign=thread*4;

           /*  Mix the message block with the previous double pipe.       */
    p256[thread]    ^= data32[thread];
    __syncthreads();
    p256[16+thread]  =                 p256[idx_f0[idx + 0]] +
                       sign_f0[sign+0]*p256[idx_f0[idx + 1]] +
                       sign_f0[sign+1]*p256[idx_f0[idx + 2]] +
                       sign_f0[sign+2]*p256[idx_f0[idx + 3]] +
                       sign_f0[sign+3]*p256[idx_f0[idx + 4]];
    __syncthreads();
    switch(thread % 5)
    {
        case 0: p256[thread] = s32_0(p256[16+thread]); break;
        case 1: p256[thread] = s32_1(p256[16+thread]); break;
        case 2: p256[thread] = s32_2(p256[16+thread]); break;
        case 3: p256[thread] = s32_3(p256[16+thread]); break;
        case 4: p256[thread] = s32_4(p256[16+thread]); break;
    }
}
```

- **Kernel 1: Calculation of f_2:**

```
__global__ void kernel_f2(u_int32_t XL32_host, u_int32_t XH32_host, u_int32_t *data32, u_int32_t
*p256) {
    u_int32_t thread = threadIdx.x;
    u_int32_t XL32 = XL32_host;
    u_int32_t XH32 = XH32_host;
```

```
    /*  Compute the double chaining pipe for the next message block. */
    switch(thread)
    {
        case 0:  p256[0] =  (shl(XH32, 5) ^ shr(p256[16],5) ^ data32[ 0]) + (    XL32    ^
p256[24] ^ p256[ 0]); break;
        case 1:  p256[1] =  (shr(XH32, 7) ^ shl(p256[17],8) ^ data32[ 1]) + (    XL32    ^
p256[25] ^ p256[ 1]); break;
        case 2:  p256[2] =  (shr(XH32, 5) ^ shl(p256[18],5) ^ data32[ 2]) + (    XL32    ^
p256[26] ^ p256[ 2]); break;
        case 3:  p256[3] =  (shr(XH32, 1) ^ shl(p256[19],5) ^ data32[ 3]) + (    XL32    ^
p256[27] ^ p256[ 3]); break;
        case 4:  p256[4] =  (shr(XH32, 3) ^     p256[20]    ^ data32[ 4]) + (    XL32    ^
p256[28] ^ p256[ 4]); break;
        case 5:  p256[5] =  (shl(XH32, 6) ^ shr(p256[21],6) ^ data32[ 5]) + (    XL32    ^
p256[29] ^ p256[ 5]); break;
        case 6:  p256[6] =  (shr(XH32, 4) ^ shl(p256[22],6) ^ data32[ 6]) + (    XL32    ^
p256[30] ^ p256[ 6]); break;
        case 7:  p256[7] =  (shr(XH32,11) ^ shl(p256[23],2) ^ data32[ 7]) + (    XL32    ^
p256[31] ^ p256[ 7]); break;
        case 8:  p256[ 8] = (    XH32     ^     p256[24]    ^ data32[ 8]) + (shl(XL32,8) ^
p256[23] ^ p256[ 8]); break;
        case 9:  p256[ 9] = (    XH32     ^     p256[25]    ^ data32[ 9]) + (shr(XL32,6) ^
p256[16] ^ p256[ 9]); break;
        case 10: p256[10] = (    XH32     ^     p256[26]    ^ data32[10]) + (shl(XL32,6) ^
p256[17] ^ p256[10]); break;
        case 11: p256[11] = (    XH32     ^     p256[27]    ^ data32[11]) + (shl(XL32,4) ^
p256[18] ^ p256[11]); break;
        case 12: p256[12] = (    XH32     ^     p256[28]    ^ data32[12]) + (shr(XL32,3) ^
p256[19] ^ p256[12]); break;
        case 13: p256[13] = (    XH32     ^     p256[29]    ^ data32[13]) + (shr(XL32,4) ^
p256[20] ^ p256[13]); break;
        case 14: p256[14] = (    XH32     ^     p256[30]    ^ data32[14]) + (shr(XL32,7) ^
p256[21] ^ p256[14]); break;
        case 15: p256[15] = (    XH32     ^     p256[31]    ^ data32[15]) + (shr(XL32,2) ^
p256[22] ^ p256[15]); break;
    }
    __syncthreads();
    if(thread < 8) p256[thread+8] += rotl32(p256[(thread+4)%8],thread+9);
}
```

In this implementation no shared memory was used. It was empirically tested the speedup of the kernel calculation in each one is negligible. The commented lines inside the kernels belong to the shared memory versions (In the appendix).

7. Results

Any solution of proposal couldn't be useful without a validation of the results. With the implementation of proposal a benchmarking tool was coded too. In this code were analyzed the most important bottlenecks of the compression procedure like:

- Copy data from host memory to device memory
- Kernel Execution
- Copy data from device memory to host memory

In many documents over the net it's said that using shared memory improves the speedup of our kernel, that's not happening in our case.

7.1. With Shared Memory

```
Timing_12.58.36-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004113
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007767
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.042342
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016823
    TOTAL TIME: 0.071044
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003870
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007768
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.061432
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015077
    TOTAL TIME: 0.088147
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001427
  --Compute f2:
    TOTAL TIME: 0.001403
```

- **kernel_f0: 0.071044 ms**
  - Copy data from host memory to device memory: **0.004113** + **0.007767** = **0.01188 ms**
  - Kernel Execution: **0.042342 ms**
  - Copy data from device memory to host memory: **0.016823 ms**

- **kernel_f2: 0.088147 ms**
  - Copy data from host memory to device memory: **0.003870** + **0.007768** = **0.011638 ms**
  - Kernel Execution: **0.061432 ms**
  - Copy data from device memory to host memory: **0.088147 ms**

All of this without taking the constant memory transferring into account. The benchmarking of complete hash computation is also done. Below a figure showing the difference between CPU and GPU performance made with an average of 10 samples (GPU: green, CPU: blue, the samples can be found in the appendix):

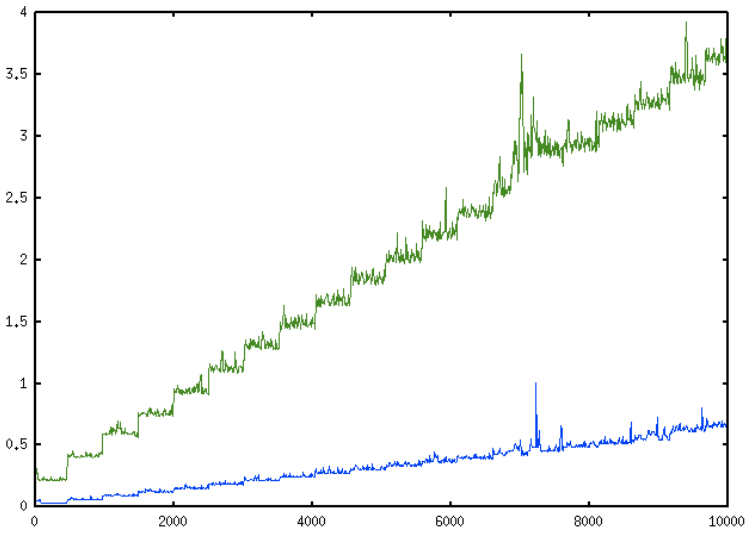


Figure 7.1: With shared memory benchmarking average

7.2. Without Shared Memory

```
Timing_12.55.32-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004083
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007808
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.041249
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.019024
```

```
    TOTAL TIME: 0.072163
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003826
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007720
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.059499
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.023448
    TOTAL TIME: 0.094493
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001429
  --Compute f2:
    TOTAL TIME: 0.001420
```

- **kernel_f0: 0.072163 ms**
  - Copy data from host memory to device memory: **0.004083** + **0.007808** = **0.011891 ms**
  - Kernel Execution: **0.041249 ms**
  - Copy data from device memory to host memory: **0.019024 ms**

- **kernel_f2: 0.094493 ms**
  - Copy data from host memory to device memory: **0.003826** + **0.007720** = **0.011546 ms**
  - Kernel Execution: **0.059499 ms**
  - Copy data from device memory to host memory: **0.023448 ms**

Below the corresponding average figure:

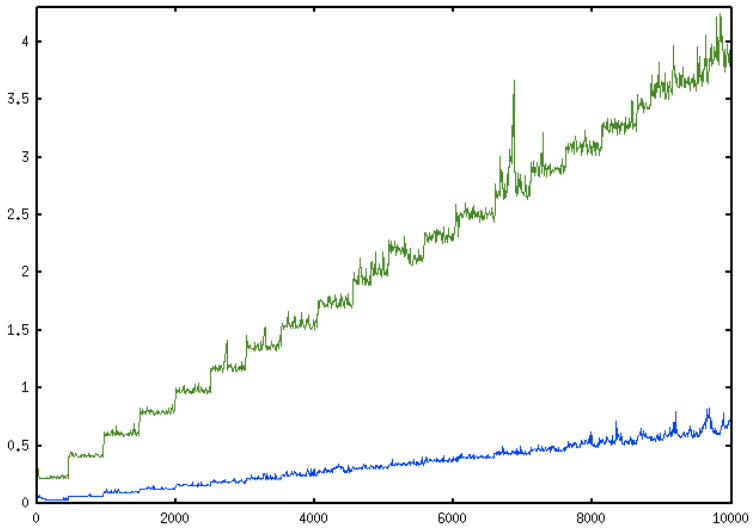


Figure 7.2: Without shared memory benchmarking average

We can calculate how much the GPU code slows down the computation:

- **With shared:**
  - f_0**:** 0.071044/0.001427 = 49.78 times slower.
  - f_2**:** 0.088147/0.001403 = 62.86 times slower.
- **Without shared:**
  - f_0**:** 0.072163/0.001429 = 50.49 times slower.
  - f_2**:** 0.094493/0.001420 = 66.54 times slower.

Using the shared capable version or not, the results are not good enough to switch CPU based code into the GPU one. But, why is this happening? There are several reasons why this hash algorithm is not going to get optimized in a GPU version code:

- **Recursive behaviour:** Because the iterative conduct added with the need of the last hash computation creates a big bottleneck avoiding the possibility to launch more and more threads in parallel.
- **Few threads quantity:** In this implementation it can't be used more than 16 threads. In concrete, this GPU is capable of executing 512 threads in parallel. If threads quantity is increased the occupancy is getting closer to 100% usage.
- **Many if statements:** Given the non-linear behaviour of the BMW algorithm an if statements are needed into it. This slows down a lot because the threads are diverged and the GPU core syncing method waits until all of them converge into the same point. In other words, in most of the cases if statements need to be avoided.
- **Not expensive calculations:** The CPU based hash algorithm calculation is cheap in computational cost terms. Only calling to the kernel in the GPU overcome the microseconds used by the CPU to execute the computation. Without taking into account the memory transferring.
- **Memory transferring:** In the GPU before doing any calculation the data needs to be transferred to the device memory. This adds delay to the final hash computation that isn't made up for parallelized version speed up.

All these reasons together don't give the possibility to have a hopeful code for working on it.

8. Future Work

Knowing the reasons why the algorithm is not getting the hoped results, it's time to think about what could be useful in the upcoming actions.

The most important bottleneck or reason why the algorithm is not working as well as wanted is that in computational terms the algorithm is costless and the time spent calling to the kernel overcome the time taken by the CPU executing the whole algorithm.

A lot of cryptographic algorithms are complex compute-intensive problems that would benefit the parallel computation of these multi-core systems. This is not the case of BLUE MIDNIGHT WISH which is complex but not compute-intensive.

If something could be said to improve the algorithm is to avoid its recursive behaviour. We can see it in the figure below how it works:

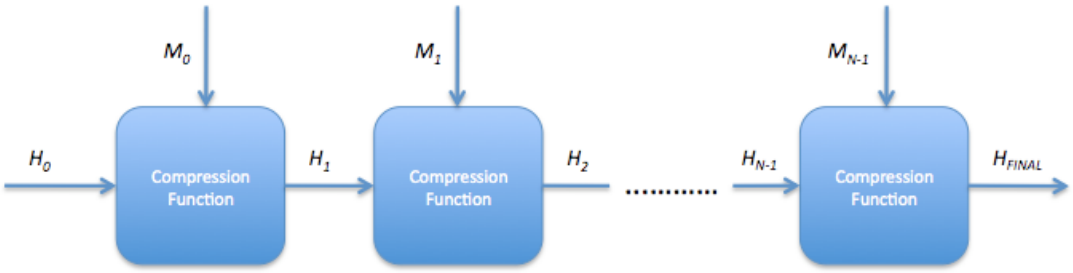


Figure 8.1: Recursive flow diagram

Avoiding this behaviour will give the possibility of optimizing as much as the GPU systems permit. Something like what the next figure shows would be the ideal solution to get incredible high performance speed up.

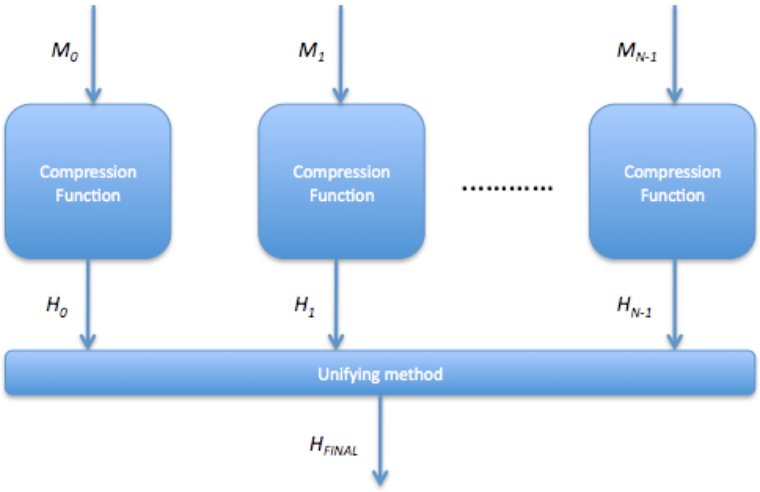


Figure 8.2: Parallel capable flow diagram

This is parallel in all of its aspects. Because it can be launched in parallel and after the next compression function does not need to wait for the previous one.

9. Conclusion

Nowadays, the GPGPU technology is at a very early stage because none one standard have been adopted and there are a lot of diverged developments over the net. During 2009-2010 a lot of new technologies based on the knowledge obtained in the early ages will appear. As well as OpenCL standard, which will help to get all the forces involved in the same standard and try to group all the developments together following the same way.

Like always, Sony and its alliance will continue in their way (although they are in the OpenCL standard consortium). This is not as important as it could sound because the controlling bubble of Cell-kind processor will not bump into the GPGPU's bubble. They are involved in supercomputing world for enterprises and not home available computation. Although perhaps after seeing the comparison between nVidia CUDA capable devices and PS3 they could be hard to beat competitors.

A lot of forums on the net started discussions based on the really non-sense sentences like "CPU is dead, long life to GPU!". This is not really true. CPU will use the capabilities of GPU to increase its performance in a lot of aspects but will never replace the CPUs at all.

It's hard to predict how is going to evolve upcoming technologies of GPGPUs but what is clear is that GPGPU will be the key point of creating the best partner for the CPU.

In our case, BMW is not improved with CUDA and a specific device card. Although the answer is negative, it still has a significant scientific value. The point is that my work acknowledges viewpoints and standings of a part of the cryptographic community that is doubtful that the cryptographic primitives will benefit when executed in parallel in many cores in one CPU.

Indeed, my experiments show that the communication costs between cores in CUDA outweigh by big margin the computational costs done inside one core (processor) unit. Probably a proper way how to use numerous cores in one CUDA processor would be to use specially designed modes of operation for hash function (like tree hashing), but that problem can be left for future work.

The GPGPU computing is not the panacea but maybe it will lead to gold computing ages. The future will tell if these novel technologies will be an historic change in the computing world or not.

10. Bibliography

[1] D. GLIGOROSKI AND S. J. KNAPSKOG, "**Turbo SHA-2**",
<http://eprint.iacr.org/2007/403>

[2] D. GLIGOROSKI, V. KLIMA, S. J. KNAPSKOG, M. EL-HADEDY, J. AMUNDSEN AND S. F. MJØLSNES, "**Blue Midnight Wish**",
<http://www.item.ntnu.no/people/personalpages/fac/danilog/blue_midnight_wish>

[3] GPGPU, "**General-Purpose Computation Using Graphics Hardware**",
<http://www.gpgpu.org/>.

[4] WIKIPEDIA, "**Miscellaneus GPGPU implementations**",
<http://en.wikipedia.org/wiki/GPGPU>,
<http://en.wikipedia.org/wiki/Larrabee_(GPU)>,
<http://en.wikipedia.org/wiki/BrookGPU>,
<http://en.wikipedia.org/wiki/OpenCL>

[5] WIKIPEDIA, "**Intel Processors Comparison**",
<http://en.wikipedia.org/wiki/List_of_Intel_Core_2_microprocessors>,
<http://en.wikipedia.org/wiki/List_of_Intel_Pentium_4_microprocessors>

[6] NVIDIA, "**CUDA: Quickstart guide**",
<http://www.nvidia.com/object/cuda_develop.html>

[7] NVIDIA, "**CUDA: Programming guide**",
<http://www.nvidia.com/object/cuda_develop.html>

[8] NVIDIA, "**CUDA: Reference manual**",
<http://www.nvidia.com/object/cuda_develop.html>

[9] NVIDIA, "**GeForce 8800 & NVIDIA CUDA: A New Architecture for Computing on the GPU**",
<http://www.ddj.com/cpp/207200659>

[10] L. HOWES (NVIDIA), "**Load structured data efficiently with CUDA**",
<http://www.ddj.com/cpp/207200659>

[11] DR. DOBB'S, "**CUDA, Supercomputing for the masses**",
<http://www.ddj.com/cpp/207200659>

[12] T. R. HALFHILL, "**Parallel processing with CUDA**",
<http://www.mdronline.com/>

[13] INTEL, "**Ct: Flexible Parallel Programming for Terascale Architectures**",
<http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf>

[14] INTEL, "**Ct: C for Throughput Computing**",
<http://techresearch.intel.com/articles/Tera-Scale/1514.htm>

[15] E. DAVIS (INTEL), "**Tera Tera Tera**",

<http://bt.pa.msu.edu/TM/BocaRaton2006/abstracts/davis.pdf>

[16] DR. DOBB'S, "**Ct In Action**",
<http://www.ddj.com/architect/212700261>

[17] K. MACKEY, "**Clearing up the confusion over Intel's Larrabee**",
<http://arstechnica.com/hardware/news/2007/04/clearing-up-the-confusion-over-intels-larrabee.ars>,
<http://arstechnica.com/hardware/news/2007/06/clearing-up-the-confusion-over-intels-larrabee-part-ii.ars>

[18] IBM, "**Cell Broadband Engine**",
<http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine>

[19] H. P. HOFSTEE, "**Introduction to Cell Broadband Engine**",
<http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/D21E662845B95D4F872570AB0055404D>

[20] N. TREVETT (KHRONOS GROUP), "**OpenCL, The Open Standard for Heterogeneous Parallel Programming**",
<http://www.khronos.org/developers/library/overview/opencl_overview.pdf>,
<http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf >

[21] T. MATTSON AND L. MEADOWS, "**A 'hands-on' introduction to OpenMP**",
<http://openmp.org/wp/2008/11/sc08-openmp-hands-on-tutorial-available/>

[22] D. NEGRUT, "**High Performance Computing for Engineering Applications**",
<http://sbel.wisc.edu/Courses/ME964/2008/index.htm>

[23] P. LEONARD, "**Parallel Computing: Strategies for Enterprise Software Development**",
<http://www.cs.colorado.edu/events/colloquia/current/leonard.html>

[24] P. LEONARD, "**The Multi-Core Dilemma**",
<http://softwareblogs.intel.com/2007/03/14/the-multi-core-dilemma-by-patrick-leonard/>

11. Appendix

11.1. SHA-3 hash competition timeline

1Q = January – March
2Q = April – June
3Q = July – September
4Q = October – December

2006

August	Second Cryptographic Hash Workshop: Assess current status, discuss hash function development strategy, and encourage further research.
4Q	Draft the preliminary minimum acceptability requirements, submission requirements, and evaluation criteria for candidate hash functions.

2007

1Q	Publish the preliminary minimum acceptability requirements, submission requirements, and evaluation criteria for public comments. Present the draft minimum acceptability requirements, submission requirements, and evaluation criteria for candidate hash functions at the RSA Conference and at FSE 2007.

2008

4Q	Submission deadline for new hash functions.

2009

2Q	Review submitted functions, and select candidates that meet basic submission requirements. Host the First Hash Function Candidate Conference to announce first round candidates. Submitters present their functions at the workshop. Call for public comments on the first round candidates.

2010

2Q	Public comment period ends. Note: Depending on the number and quality of the submissions, NIST may either extend the length of the initial public comment period to allow sufficient time for the public analysis of the candidate algorithms, or may include additional rounds of analysis in order to successively reduce the number of candidate algorithms for further consideration as finalist algorithms. In these cases, NIST may host multiple workshops to discuss analysis results on candidate algorithms until it is ready to select the finalists. Note that subsequent dates in the timeline assume that the initial comment period will not be extended or additional rounds will not be required.
2Q	Hold the Second Hash Function Candidate Conference. Discuss

	the analysis results on the submitted candidates. Submitters may identify possible improvements for their algorithms.
3Q	Address the public comments on the submitted candidates; select the finalists. Prepare a report to explain the selection. Announce the finalists. Publish the selection report.
4Q	Submitters of the finalist candidates announce any tweaks to their submissions. Final round begins.

2011

4Q	Public comment period for the final round ends.

2012

1Q	Host the Final Hash Function Candidate Conference. Submitters of the finalist algorithms discuss the comments on their submissions.
2Q	Address public comments, and select the winner. Prepare a report to describe the final selection(s). Announce the new hash function(s).
3Q	Draft the revised hash standard. Publish the draft standard for public comments.
4Q	Public comment period ends. Address public comments. Send the proposed standard to the Secretary of Commerce for signature.

11.2. Implemented parallel blocks

11.2.1. Computation of f_0

$$f_0 : \{0,1\}^{2m} \rightarrow \{0,1\}^m$$

Input: Message block $M^{(i)} = (M_0^{(i)}, M_1^{(i)}, \ldots, M_{15}^{(i)})$, and the previous double pipe $H^{(i-1)} = (H_0^{(i-1)}, H_1^{(i-1)}, \ldots, H_{15}^{(i-1)})$.
Output: First part of the quadruple pipe $Q_a^{(i)} = (Q_0^{(i)}, Q_1^{(i)}, \ldots, Q_{15}^{(i)})$.

1. Bijective transform of $M^{(i)} \oplus H^{(i-1)}$:

Step 1: 16 threads

$$\begin{aligned}
W_0^{(i)} &= (M_5^{(i)} \oplus H_5^{(i-1)}) - (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) + (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\
W_1^{(i)} &= (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_8^{(i)} \oplus H_8^{(i-1)}) + (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) + (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) - (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\
W_2^{(i)} &= (M_0^{(i)} \oplus H_0^{(i-1)}) + (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_9^{(i)} \oplus H_9^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) + (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\
W_3^{(i)} &= (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_1^{(i)} \oplus H_1^{(i-1)}) + (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) \\
W_4^{(i)} &= (M_1^{(i)} \oplus H_1^{(i-1)}) + (M_2^{(i)} \oplus H_2^{(i-1)}) + (M_9^{(i)} \oplus H_9^{(i-1)}) - (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) - (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\
W_5^{(i)} &= (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_2^{(i)} \oplus H_2^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) + (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\
W_6^{(i)} &= (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) \\
W_7^{(i)} &= (M_1^{(i)} \oplus H_1^{(i-1)}) - (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) - (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\
W_8^{(i)} &= (M_2^{(i)} \oplus H_2^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) - (M_6^{(i)} \oplus H_6^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) - (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\
W_9^{(i)} &= (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_3^{(i)} \oplus H_3^{(i-1)}) + (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\
W_{10}^{(i)} &= (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_1^{(i)} \oplus H_1^{(i-1)}) - (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\
W_{11}^{(i)} &= (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_2^{(i)} \oplus H_2^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) + (M_9^{(i)} \oplus H_9^{(i-1)}) \\
W_{12}^{(i)} &= (M_1^{(i)} \oplus H_1^{(i-1)}) + (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_9^{(i)} \oplus H_9^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) \\
W_{13}^{(i)} &= (M_2^{(i)} \oplus H_2^{(i-1)}) + (M_4^{(i)} \oplus H_4^{(i-1)}) + (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) + (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) \\
W_{14}^{(i)} &= (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) + (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) \\
W_{15}^{(i)} &= (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) - (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_9^{(i)} \oplus H_9^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)})
\end{aligned}$$

2. Further bijective transform of $W_j^{(i)}, j = 0, \ldots, 15$:

Step 2: 16 threads

$$\begin{aligned}
Q_0^{(i)} &= s_0(W_0^{(i)}); & Q_1^{(i)} &= s_1(W_1^{(i)}); & Q_2^{(i)} &= s_2(W_2^{(i)}); & Q_3^{(i)} &= s_3(W_3^{(i)}); \\
Q_4^{(i)} &= s_4(W_4^{(i)}); & Q_5^{(i)} &= s_0(W_5^{(i)}); & Q_6^{(i)} &= s_1(W_6^{(i)}); & Q_7^{(i)} &= s_2(W_7^{(i)}); \\
Q_8^{(i)} &= s_3(W_8^{(i)}); & Q_9^{(i)} &= s_4(W_9^{(i)}); & Q_{10}^{(i)} &= s_0(W_{10}^{(i)}); & Q_{11}^{(i)} &= s_1(W_{11}^{(i)}); \\
Q_{12}^{(i)} &= s_2(W_{12}^{(i)}); & Q_{13}^{(i)} &= s_3(W_{13}^{(i)}); & Q_{14}^{(i)} &= s_4(W_{14}^{(i)}); & Q_{15}^{(i)} &= s_0(W_{15}^{(i)});
\end{aligned}$$

11.2.1. Computation of f_2

Folding $f_2 : \{0,1\}^{3m} \rightarrow \{0,1\}^m$

Input: Message block $M^{(i)} = (M_0^{(i)}, M_1^{(i)}, \ldots, M_{15}^{(i)})$,
quadruple pipe $Q^{(i)} = (Q_0^{(i)}, Q_1^{(i)}, \ldots, Q_{15}^{(i)}, Q_{16}^{(i)}, \ldots, Q_{31}^{(i)})$.
Output: New double pipe $H^{(i)} = (H_0^{(i)}, H_1^{(i)}, \ldots, H_{15}^{(i)})$.

1. Compute the cumulative temporary variables XL and XH.

$$XL = Q_{16}^{(i)} \oplus Q_{17}^{(i)} \oplus \ldots \oplus Q_{23}^{(i)}$$
$$XH = XL \oplus Q_{24}^{(i)} \oplus Q_{25}^{(i)} \oplus \ldots \oplus Q_{31}^{(i)}$$

2. Compute the new double pipe $H^{(i)}$:

Step 1: 16 threads

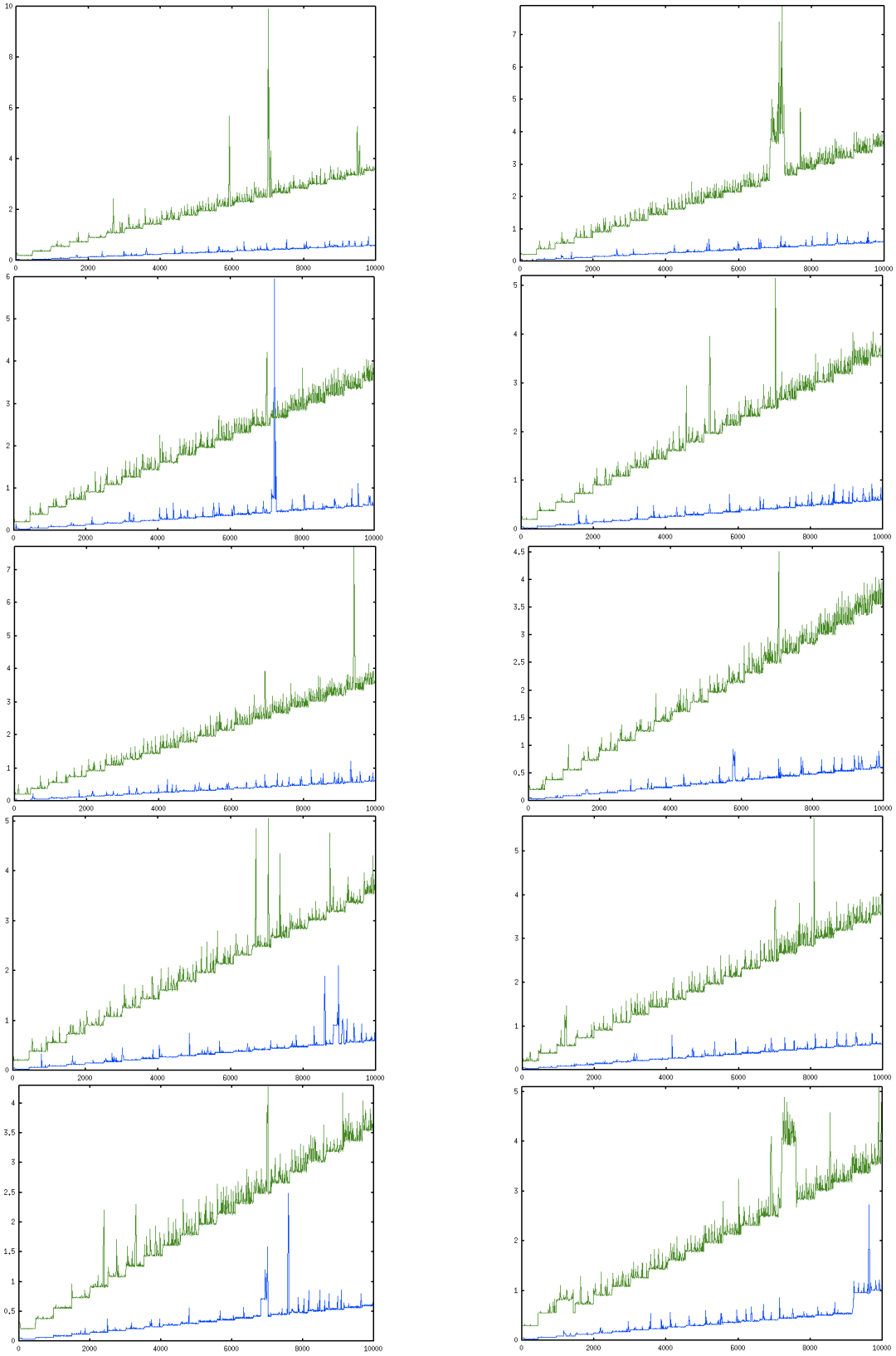
$$\begin{aligned}
H_0^{(i)} &= \left(SHL^5(XH) \oplus SHR^5(Q_{16}^{(i)}) \oplus M_0^{(i)}\right) + \left(XL \oplus Q_{24}^{(i)} \oplus Q_0^{(i)}\right)\\
H_1^{(i)} &= \left(SHR^7(XH) \oplus SHL^8(Q_{17}^{(i)}) \oplus M_1^{(i)}\right) + \left(XL \oplus Q_{25}^{(i)} \oplus Q_1^{(i)}\right)\\
H_2^{(i)} &= \left(SHR^5(XH) \oplus SHL^5(Q_{18}^{(i)}) \oplus M_2^{(i)}\right) + \left(XL \oplus Q_{26}^{(i)} \oplus Q_2^{(i)}\right)\\
H_3^{(i)} &= \left(SHR^1(XH) \oplus SHL^5(Q_{19}^{(i)}) \oplus M_3^{(i)}\right) + \left(XL \oplus Q_{27}^{(i)} \oplus Q_3^{(i)}\right)\\
H_4^{(i)} &= \left(SHR^3(XH) \oplus Q_{20}^{(i)} \oplus M_4^{(i)}\right) + \left(XL \oplus Q_{28}^{(i)} \oplus Q_4^{(i)}\right)\\
H_5^{(i)} &= \left(SHL^6(XH) \oplus SHR^6(Q_{21}^{(i)}) \oplus M_5^{(i)}\right) + \left(XL \oplus Q_{29}^{(i)} \oplus Q_5^{(i)}\right)\\
H_6^{(i)} &= \left(SHR^4(XH) \oplus SHL^6(Q_{22}^{(i)}) \oplus M_6^{(i)}\right) + \left(XL \oplus Q_{30}^{(i)} \oplus Q_6^{(i)}\right)\\
H_7^{(i)} &= \left(SHR^{11}(XH) \oplus SHL^2(Q_{23}^{(i)}) \oplus M_7^{(i)}\right) + \left(XL \oplus Q_{31}^{(i)} \oplus Q_7^{(i)}\right)
\end{aligned}$$

Step 2: 8 of 16 threads

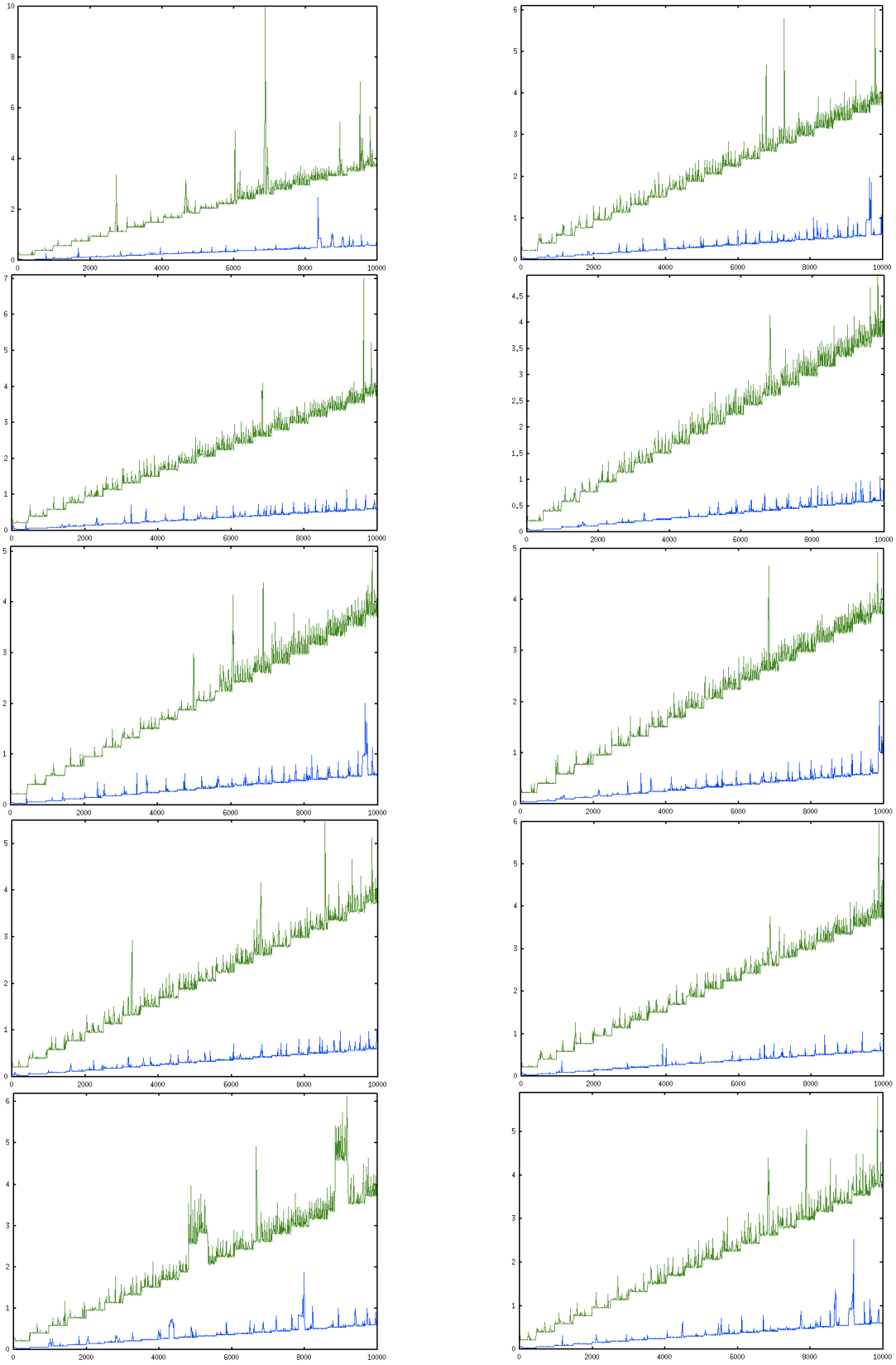
$$\begin{aligned}
H_8^{(i)} &= ROTL^9(H_4^{(i)}) + \left(XH \oplus Q_{24}^{(i)} \oplus M_8^{(i)}\right) + \left(SHL^8(XL) \oplus Q_{23}^{(i)} \oplus Q_8^{(i)}\right)\\
H_9^{(i)} &= ROTL^{10}(H_5^{(i)}) + \left(XH \oplus Q_{25}^{(i)} \oplus M_9^{(i)}\right) + \left(SHR^6(XL) \oplus Q_{16}^{(i)} \oplus Q_9^{(i)}\right)\\
H_{10}^{(i)} &= ROTL^{11}(H_6^{(i)}) + \left(XH \oplus Q_{26}^{(i)} \oplus M_{10}^{(i)}\right) + \left(SHL^6(XL) \oplus Q_{17}^{(i)} \oplus Q_{10}^{(i)}\right)\\
H_{11}^{(i)} &= ROTL^{12}(H_7^{(i)}) + \left(XH \oplus Q_{27}^{(i)} \oplus M_{11}^{(i)}\right) + \left(SHL^4(XL) \oplus Q_{18}^{(i)} \oplus Q_{11}^{(i)}\right)\\
H_{12}^{(i)} &= ROTL^{13}(H_0^{(i)}) + \left(XH \oplus Q_{28}^{(i)} \oplus M_{12}^{(i)}\right) + \left(SHR^3(XL) \oplus Q_{19}^{(i)} \oplus Q_{12}^{(i)}\right)\\
H_{13}^{(i)} &= ROTL^{14}(H_1^{(i)}) + \left(XH \oplus Q_{29}^{(i)} \oplus M_{13}^{(i)}\right) + \left(SHR^4(XL) \oplus Q_{20}^{(i)} \oplus Q_{13}^{(i)}\right)\\
H_{14}^{(i)} &= ROTL^{15}(H_2^{(i)}) + \left(XH \oplus Q_{30}^{(i)} \oplus M_{14}^{(i)}\right) + \left(SHR^7(XL) \oplus Q_{21}^{(i)} \oplus Q_{14}^{(i)}\right)\\
H_{15}^{(i)} &= ROTL^{16}(H_3^{(i)}) + \left(XH \oplus Q_{31}^{(i)} \oplus M_{15}^{(i)}\right) + \left(SHR^2(XL) \oplus Q_{22}^{(i)} \oplus Q_{15}^{(i)}\right)
\end{aligned}$$

11.3. Hash Benchmarks

11.3.1. With Shared Memory



11.3.1. Without Shared Memory



11.4. Compression function timing averages

11.4.1. With Shared Memory

```
Timing_12.58.21-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004080
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007842
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.042444
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016691
    TOTAL TIME: 0.071058
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003846
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007797
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.061484
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015070
    TOTAL TIME: 0.088197
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001444
  --Compute f2:
    TOTAL TIME: 0.001420

Timing_12.58.29-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004154
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007810
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.042450
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016771
    TOTAL TIME: 0.071185
```

```
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003830
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007786
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.061280
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015093
    TOTAL TIME: 0.087989
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001466
  --Compute f2:
    TOTAL TIME: 0.001431

Timing_12.58.36-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004113
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007767
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.042342
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016823
    TOTAL TIME: 0.071044
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003870
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007768
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.061432
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015077
    TOTAL TIME: 0.088147
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001427
  --Compute f2:
    TOTAL TIME: 0.001403

Timing_12.58.44-12Feb2009.txt:

GPU
```

```
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004133
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007796
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.043016
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016717
    TOTAL TIME: 0.071662
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003897
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007753
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.062323
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015118
    TOTAL TIME: 0.089091
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001449
  --Compute f2:
    TOTAL TIME: 0.001424

Timing_12.58.51-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004187
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007854
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.043089
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016732
    TOTAL TIME: 0.071862
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003877
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007749
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.062509
```

```
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015061
    TOTAL TIME: 0.089196
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001443
  --Compute f2:
    TOTAL TIME: 0.001435

Timing_12.58.58-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004111
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007792
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.042583
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016741
    TOTAL TIME: 0.071227
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003848
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007771
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.061858
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015125
    TOTAL TIME: 0.088601
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001421
  --Compute f2:
    TOTAL TIME: 0.001406

Timing_12.59.06-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004077
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007806
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
```

```
        time: 0.042433
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016778
    TOTAL TIME: 0.071094
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003873
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007769
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.061320
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015021
    TOTAL TIME: 0.087982
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001433
  --Compute f2:
    TOTAL TIME: 0.001422

Timing_12.59.13-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004061
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007813
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.042503
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016655
    TOTAL TIME: 0.071032
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003804
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007783
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.061448
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015101
    TOTAL TIME: 0.088136
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001434
  --Compute f2:
```

```
    TOTAL TIME: 0.001406

Timing_12.59.20-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004078
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007802
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.042440
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016658
    TOTAL TIME: 0.070977
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003811
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007749
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.061450
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015081
    TOTAL TIME: 0.088091
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001437
  --Compute f2:
    TOTAL TIME: 0.001409

Timing_12.59.28-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004115
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007817
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.042358
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.016720
    TOTAL TIME: 0.071010
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003828
```

```
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007717
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.061282
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.015080
    TOTAL TIME: 0.087907
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001475
  --Compute f2:
    TOTAL TIME: 0.001451
```

11.4.2. Without Shared Memory

```
Timing_12.55.16-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004146
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007809
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.041207
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.018972
    TOTAL TIME: 0.072134
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003834
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007762
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.059848
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.023426
    TOTAL TIME: 0.094870
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001394
  --Compute f2:
    TOTAL TIME: 0.001375

Timing_12.55.25-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004189
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007822
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.041337
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.019014
    TOTAL TIME: 0.072363
  --Compute f2:
```

```
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003873
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007756
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.059801
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.023388
    TOTAL TIME: 0.094819
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001419
  --Compute f2:
    TOTAL TIME: 0.001390

Timing_12.55.32-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004083
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007808
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.041249
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.019024
    TOTAL TIME: 0.072163
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003826
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007720
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.059499
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.023448
    TOTAL TIME: 0.094493
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001429
  --Compute f2:
    TOTAL TIME: 0.001420

Timing_12.55.40-12Feb2009.txt:

GPU
**Compression256()
```

```
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004158
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007779
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.041192
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.019050
    TOTAL TIME: 0.072180
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003853
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007721
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.059527
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.023480
    TOTAL TIME: 0.094582
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001416
  --Compute f2:
    TOTAL TIME: 0.001397

Timing_12.55.48-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004161
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007774
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.041171
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.019020
    TOTAL TIME: 0.072127
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003800
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007724
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.059663
      cudaMemcpyDeviceToHost:
```

```
        p256_dev->p256 (128 bytes)
        time: 0.023436
    TOTAL TIME: 0.094623
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001417
  --Compute f2:
    TOTAL TIME: 0.001403

Timing_12.55.55-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004139
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007806
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.041350
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.019088
    TOTAL TIME: 0.072383
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003816
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007747
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.059547
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.023443
    TOTAL TIME: 0.094552
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001434
  --Compute f2:
    TOTAL TIME: 0.001424

Timing_12.56.03-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004118
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007774
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.041161
```

```
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.019004
    TOTAL TIME: 0.072058
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003802
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007743
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.059298
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.023495
    TOTAL TIME: 0.094336
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001422
  --Compute f2:
    TOTAL TIME: 0.001400

Timing_12.56.10-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004149
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007759
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.041260
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.018981
    TOTAL TIME: 0.072149
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003818
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007728
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.059723
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.023441
    TOTAL TIME: 0.094711
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001427
  --Compute f2:
    TOTAL TIME: 0.001407
```

```
Timing_12.56.18-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004128
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007771
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.041299
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.019006
    TOTAL TIME: 0.072203
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003828
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007708
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.059858
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.023380
    TOTAL TIME: 0.094775
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001404
  --Compute f2:
    TOTAL TIME: 0.001412

Timing_12.56.25-12Feb2009.txt:

GPU
**Compression256()
  --Compute f0:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.004326
      cudaMemcpyHostToDevice:
        p256->p256_dev (128 bytes)
        time: 0.007905
      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):
        time: 0.042749
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.019343
    TOTAL TIME: 0.074323
  --Compute f2:
      cudaMemcpyHostToDevice:
        data32->data32_dev (64 bytes)
        time: 0.003961
      cudaMemcpyHostToDevice:
```

```
        p256->p256_dev (128 bytes)
        time: 0.007764
      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):
        time: 0.061921
      cudaMemcpyDeviceToHost:
        p256_dev->p256 (128 bytes)
        time: 0.023565
    TOTAL TIME: 0.097211
CPU
**Compression256()
  --Compute f0:
    TOTAL TIME: 0.001419
  --Compute f2:
    TOTAL TIME: 0.001393
```

11.5. Source code

11.5.1. Timer.h

```
#ifndef TIMER_H_
#define TIMER_H_

#include <cuda_runtime.h>
#include <cutil.h>
#include <stdio.h>
#include <stdlib.h>

class Timer {
private:
    unsigned int timer;
public:
    Timer() {
        CUT_SAFE_CALL(cutCreateTimer(&timer));
    };
    void reset(void) {
        CUT_SAFE_CALL(cutResetTimer(timer));
    };
    void start(void) {
        CUT_SAFE_CALL(cutStartTimer(timer));
    };
    void stop(void) {
        CUT_SAFE_CALL(cutStopTimer(timer));
    };
    float gettime(void) {
        return cutGetTimerValue(timer);
    };
};

#define BLOCKS 256

class Timing {
private:
    Timer timer;
    float *timing;
    int   *counter;
public:
    Timing(int n) {
        timing  = new float[n];
        counter = new int[n];
        for(int i = 0 ; i < n ; i++) {
            timing[i]  = 0.0;
            counter[i] = 0;
        };
    };
    Timing() {
        delete [] timing;
        delete [] counter;
    };
    void DoTiming1(void) {
    #ifdef _DEBUG
        timer.reset();
        timer.start();
    #endif
    };
    void DoTiming2(int i) {
    #ifdef _DEBUG
        timing[i] = timing[i] + timer.gettime();
        counter[i]++;
```

```
#endif
    };
    float* gettiming(void) {
        return timing;
    };
    int* getcounter(void) {
        return counter;
    };
};

#endif
```

11.5.2. main.cpp

```
#include <cuda_runtime.h>
#include <cutil.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "Timer.h"
#include "bmw/BlueMidnightWish.h"

#define FILENAME                    "Benchmark"
#define MAX_FILENAME_SIZE           2048

#define BENCHMARK_CPU               0
#define BENCHMARK_GPU               1

#define BUFFER_LENGTH               10000
#define BENCHMARK_STEP              10

char str[MAX_FILENAME_SIZE];
unsigned char buffer[BUFFER_LENGTH];

#ifdef _DEBUG
extern float *timing_gpu_256;
extern float *timing_cpu_256;
extern int *counter_gpu_256;
extern int *counter_cpu_256;
//extern float *timing_gpu_512;
//extern float *timing_cpu_512;
#endif

//
// Fill buffer with random bytes
//
unsigned int rand32(void) {
    static unsigned int r4,r_cnt = -1,w = 521288629,z = 362436069;

    z = 36969 * (z & 65535) + (z >> 16);
    w = 18000 * (w & 65535) + (w >> 16);

    r_cnt = 0; r4 = (z << 16) + w; return r4;
}

unsigned char rand8(void) {
    static unsigned int r4,r_cnt = 4;

    if(r_cnt == 4){
        r4 = rand32();
        r_cnt = 0;
    }

    return (char)(r4 >> (8 * r_cnt++));
}

void BlockRandomFill(void) {
    unsigned int  i;
    for(i = 0; i < BUFFER_LENGTH; ++i) buffer[i] = rand8();
}
```

```
//
// Get formatted date and time
//
char* getDate(void) {
    struct tm *ptr;
    time_t tm;
    char *str;

    str = (char *)malloc(25*sizeof(char));

    tm = time(NULL);
    ptr = localtime(&tm);
    strftime(str,25,"%H.%M.%S-%d%b%Y",ptr);

    return str;
}

//
// Init & Close CUDA
//
#if __DEVICE_EMULATION__

bool InitCUDA(int argc, char **argv){return true;}
bool CloseCUDA(int argc, char **argv){return true;}

#else
bool InitCUDA(int argc, char **argv)
{
    int count = 0;
    int i = 0;

    cudaGetDeviceCount(&count);
    if(count == 0) {
        fprintf(stderr, "There is no device.\n");
        return false;
    }

    for(i = 0; i < count; i++) {
        cudaDeviceProp prop;
        if(cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
            if(prop.major >= 1) {
                break;
            }
        }
    }
    if(i == count) {
        fprintf(stderr, "There is no device supporting CUDA.\n");
        return false;
    }
    cudaSetDevice(i);

    printf("CUDA initialized.\n");
    return true;
}

void CloseCUDA(int argc, char **argv){CUT_EXIT(argc,argv);return;}

#endif

//
// Benchmark and store output in Octave-Matlab compatible format
//
void Benchmark(char *date, int cpu_or_gpu) {
    FILE *fd;
```

```
    unsigned int i,j,k,size,hsizes[] = {224,256,384,512};
    unsigned char fail=0,hv[BlueMidnightWish512_BLOCK_SIZE];

    Timer timer;

    for(i=0 ; i<4 ; i++) {
        sprintf(str,"data/%s_%s_%i_%s.dat\0", FILENAME, (cpu_or_gpu) ? "GPU":"CPU",
hsizes[i],date);
        fd = fopen(str,"w");
        for(j=1 ; j<BUFFER_LENGTH; j+=BENCHMARK_STEP,fail=0) {
            timer.reset();
            timer.start();
            if(cpu_or_gpu == 0) {
                 if(HashCPU(hsizes[i],buffer,j,hv) != SUCCESS){fail = 1;}
            }
            else {
                 if(HashGPU(hsizes[i],buffer,j,hv) != SUCCESS){fail = 1;}
            }
            timer.stop();

            if(fail) { fprintf(fd,"%10i FAILED\n",j); }
            else {
                fprintf(fd,"%10i %2.8f\n",j,timer.gettime());
            }
        }
        fclose(fd);
    }
}

void DoBenchmarking(void) {
    char *date = getDate();

    Benchmark(date, BENCHMARK_CPU);
    Benchmark(date, BENCHMARK_GPU);

#ifdef _DEBUG
    FILE *fd_timing;
    sprintf(str,"data/Timing_%s.txt\0",date);
    fd_timing = fopen(str,"w");

    float t_gpu_256[8];
    float t_cpu_256[2];

    t_gpu_256[0] = timing_gpu_256[0]/(float)counter_gpu_256[0];
    t_gpu_256[1] = timing_gpu_256[1]/(float)counter_gpu_256[1];
    t_gpu_256[2] = timing_gpu_256[2]/(float)counter_gpu_256[2];
    t_gpu_256[3] = timing_gpu_256[3]/(float)counter_gpu_256[3];
    t_gpu_256[4] = timing_gpu_256[4]/(float)counter_gpu_256[4];
    t_gpu_256[5] = timing_gpu_256[5]/(float)counter_gpu_256[5];
    t_gpu_256[6] = timing_gpu_256[6]/(float)counter_gpu_256[6];
    t_gpu_256[7] = timing_gpu_256[7]/(float)counter_gpu_256[7];

    t_cpu_256[0] = timing_cpu_256[0]/(float)counter_cpu_256[0];
    t_cpu_256[1] = timing_cpu_256[1]/(float)counter_cpu_256[1];

    fprintf(fd_timing, "Timing average calculation by Gorka Lertxundi Osa\n");
    fprintf(fd_timing, "------------------------------------------------\n");
    fprintf(fd_timing, "GPU\n");
    fprintf(fd_timing, "**Compression256()\n");
    fprintf(fd_timing, "  --Compute f0:\n");
    fprintf(fd_timing, "      cudaMemcpyHostToDevice:\n");
    fprintf(fd_timing, "        data32->data32_dev (%i bytes)\n",16*sizeof(u_int32_t));
    fprintf(fd_timing, "        time: %f\n",t_gpu_256[0]);
    fprintf(fd_timing, "      cudaMemcpyHostToDevice:\n");
    fprintf(fd_timing, "        p256->p256_dev (%i bytes)\n",32*sizeof(u_int32_t));
```

```
    fprintf(fd_timing, "        time: %f\n",t_gpu_256[1]);
    fprintf(fd_timing, "      kernel_256_f0<<<1,16,2048>>>(data32_dev, p256_dev):\n");
    fprintf(fd_timing, "        time: %f\n",t_gpu_256[2]);
    fprintf(fd_timing, "      cudaMemcpyDeviceToHost:\n");
    fprintf(fd_timing, "        p256_dev->p256 (%i bytes)\n",32*sizeof(u_int32_t));
    fprintf(fd_timing, "        time: %f\n",t_gpu_256[3]);
    fprintf(fd_timing, "    TOTAL TIME: %f\n", t_gpu_256[0]+
                                               t_gpu_256[1]+
                                               t_gpu_256[2]+
                                               t_gpu_256[3]);
    fprintf(fd_timing, "  --Compute f2:\n");
    fprintf(fd_timing, "      cudaMemcpyHostToDevice:\n");
    fprintf(fd_timing, "        data32->data32_dev (%i bytes)\n",16*sizeof(u_int32_t));
    fprintf(fd_timing, "        time: %f\n",t_gpu_256[4]);
    fprintf(fd_timing, "      cudaMemcpyHostToDevice:\n");
    fprintf(fd_timing, "        p256->p256_dev (%i bytes)\n",32*sizeof(u_int32_t));
    fprintf(fd_timing, "        time: %f\n",t_gpu_256[5]);
    fprintf(fd_timing, "      kernel_256_f2<<<1,16,2048>>>(XL32, XH32, data32_dev, p256_dev):\n");
    fprintf(fd_timing, "        time: %f\n",t_gpu_256[6]);
    fprintf(fd_timing, "      cudaMemcpyDeviceToHost:\n");
    fprintf(fd_timing, "        p256_dev->p256 (%i bytes)\n",32*sizeof(u_int32_t));
    fprintf(fd_timing, "        time: %f\n",t_gpu_256[7]);
    fprintf(fd_timing, "    TOTAL TIME: %f\n", t_gpu_256[4]+
                                               t_gpu_256[5]+
                                               t_gpu_256[6]+
                                               t_gpu_256[7]);

    fprintf(fd_timing, "CPU\n");
    fprintf(fd_timing, "**Compression256()\n");
    fprintf(fd_timing, "  --Compute f0:\n");
    fprintf(fd_timing, "    TOTAL TIME: %f\n",t_cpu_256[0]);
    fprintf(fd_timing, "  --Compute f2:\n");
    fprintf(fd_timing, "    TOTAL TIME: %f\n",t_cpu_256[1]);

    fclose(fd_timing);
#endif
}

int main(int argc, char *argv[]) {
    if(!InitCUDA(argc, argv)) {
        return 0;
    }

    BlockRandomFill();
    DoBenchmarking();

    CloseCUDA(argc, argv);

    return 0;
}
```

11.5.3. BlueMidnightWish.h

```
// We use this type definition to ensure that
// "unsigned long" on 32-bit and 64-bit little-endian
// operating systems are 4 bytes long.
#if defined( _MSC_VER )
typedef unsigned long u_int32_t;
typedef unsigned long long u_int64_t;
#else
#include <sys/types.h>
#endif

// General SHA-3 definitions
typedef unsigned char BitSequence;
typedef u_int64_t DataLength; // a typical 64-bit value
typedef enum { SUCCESS = 0, FAIL = 1, BAD_HASHLEN = 2, BAD_CONSECUTIVE_CALL_TO_UPDATE = 3 }
HashReturn;
// Blue Midnight Wish allows to call Update() function consecutively only if the total length of
stored
// unprocessed data and the new supplied data is less than or equal to the BLOCK_SIZE on which the
// compression functions operates. Otherwise BAD_CONSECUTIVE_CALL_TO_UPDATE is returned.


// Specific algorithm definitions
#define BlueMidnightWish224_DIGEST_SIZE  28
#define BlueMidnightWish224_BLOCK_SIZE   64
#define BlueMidnightWish256_DIGEST_SIZE  32
#define BlueMidnightWish256_BLOCK_SIZE   64
#define BlueMidnightWish384_DIGEST_SIZE  48
#define BlueMidnightWish384_BLOCK_SIZE  128
#define BlueMidnightWish512_DIGEST_SIZE  64
#define BlueMidnightWish512_BLOCK_SIZE  128


// Here we define the default Blue Midnight Wish tunable security parameters.
// The parameters are named EXPAND_1_ROUNDS and EXPAND_2_ROUNDS.
// Since Blue Midnight Wish has 16 rounds in its message expansion part, the
// following relation for these parameters should be satisfied:
//
//          EXPAND_1_ROUNDS + EXPAND_2_ROUNDS = 16
//
// Blue Midnight Wish in its message expansion part uses 2 different functions:
// expand_1 and expand_2.
//
// expand_1 is the more complex and more time consuming, but offers the fastest
// diffusion of bit differences and produces variables that have the most complex
// nonlinear relations with previous 16 variables in the message expansion part.
//
// expand_2 is faster than expand_1, and uses faster and simpler functions than
// expand_1. The produced variables still have complex nonlinear relations with
// previous 16 variables in the message expansion part.
//
#define EXPAND_1_ROUNDS 2
#define EXPAND_2_ROUNDS 14


typedef struct
{
    u_int32_t DoublePipe[32];
    BitSequence LastPart[BlueMidnightWish256_BLOCK_SIZE * 2];
} Data256;
typedef struct
```

```
{
    u_int64_t DoublePipe[32];
    BitSequence LastPart[BlueMidnightWish512_BLOCK_SIZE * 2];
} Data512;

typedef struct {
    int hashbitlen;

    // + algorithm specific parameters
    u_int64_t bits_processed;
    union
    {
        Data256  p256[1];
        Data512  p512[1];
    } pipe[1];
    int unprocessed_bits;
} hashState;

HashReturn InitCPU(hashState *state, int hashbitlen);
HashReturn UpdateCPU(hashState *state, const BitSequence *data, DataLength databitlen);
HashReturn FinalCPU(hashState *state, BitSequence *hashval);
HashReturn HashCPU(int hashbitlen, const BitSequence *data, DataLength databitlen, BitSequence
*hashval);

HashReturn InitGPU(hashState *state, int hashbitlen);
HashReturn UpdateGPU(hashState *state, const BitSequence *data, DataLength databitlen);
HashReturn FinalGPU(hashState *state, BitSequence *hashval);
HashReturn HashGPU(int hashbitlen, const BitSequence *data, DataLength databitlen, BitSequence
*hashval);
```

11.5.4. BlueMidnigthWish_CPU.cpp

See bibliography of "**Blue Midnight Wish**". It is the same as *BlueMidnightWish.c* source code. It only differs that it uses some timing functions to know how much time spends doing the compression function.

11.5.5. BlueMidnightWish_GPU.cpp

```
#include <string.h>
#include "BlueMidnightWish.h"

#include <cuda_runtime.h>
#include <cutil.h>

#include "../Timer.h"

#define rotl32(x,n)   (((x) << n) | ((x) >> (32 - n)))
#define rotr32(x,n)   (((x) >> n) | ((x) << (32 - n)))
//#define rotl32 _lrotl
//#define rotr32 _lrotr

#define rotl64(x,n)   (((x) << n) | ((x) >> (64 - n)))
#define rotr64(x,n)   (((x) >> n) | ((x) << (64 - n)))
//#define rotl64 _rotl64
//#define rotr64 _rotr64

#define shl(x,n)      ((x) << n)
#define shr(x,n)      ((x) >> n)

/* BlueMidnightWish224 initial double chaining pipe */
const u_int32_t i224p2[16] =
{   0x00010203ul, 0x04050607ul, 0x08090a0bul, 0x0c0d0e0ful,
    0x10111213ul, 0x14151617ul, 0x18191a1bul, 0x1c1d1e1ful,
    0x20212223ul, 0x24252627ul, 0x28292a2bul, 0x2c2d2e2ful,
    0x30313233ul, 0x24353637ul, 0x38393a3bul, 0x3c3d3e3ful,
};

/* BlueMidnightWish256 initial double chaining pipe */
const u_int32_t i256p2[16] =
{   0x40414243ul, 0x44454647ul, 0x48494a4bul, 0x4c4d4e4ful,
    0x50515253ul, 0x54555657ul, 0x58595a5bul, 0x5c5d5e5ful,
    0x60616263ul, 0x64656667ul, 0x68696a6bul, 0x6c6d6e6ful,
    0x70717273ul, 0x74757677ul, 0x78797a7bul, 0x7c7d7e7ful,
};

/* BlueMidnightWish384 initial double chaining pipe */
const u_int64_t i384p2[16] =
{
    0x0001020304050607ull, 0x08090a0b0c0d0e0full,
    0x1011121314151617ull, 0x18191a1b1c1d1e1full,
    0x2021222324252627ull, 0x28292a2b2c2d2e2full,
    0x3031323324353637ull, 0x38393a3b3c3d3e3full,
    0x4041424344454647ull, 0x48494a4b4c4d4e4full,
    0x5051525354555657ull, 0x58595a5b5c5d5e5full,
    0x6061626364656667ull, 0x68696a6b6c6d6e6full,
    0x7071727374757677ull, 0x78797a7b7c7d7e7full
};

/* BlueMidnightWish512 initial double chaining pipe */
const u_int64_t i512p2[16] =
{
    0x8081828384858687ull, 0x88898a8b8c8d8e8full,
    0x9091929394959697ull, 0x98999a9b9c9d9e9full,
    0xa0a1a2a3a4a5a6a7ull, 0xa8a9aaabacadaeafull,
    0xb0b1b2b3b4b5b6b7ull, 0xb8b9babbbcbdbebfull,
    0xc0c1c2c3c4c5c6c7ull, 0xc8c9cacbcccdcecfull,
    0xd0d1d2d3d4d5d6d7ull, 0xd8d9dadbdcdddedfull,
    0xe0e1e2e3e4e5e6e7ull, 0xe8e9eaebecedeeefull,
```

```
    0xf0f1f2f3f4f5f6f7ull, 0xf8f9fafbfcfdfeffull
};

#define hashState224(x)  ((x)->pipe->p256)
#define hashState256(x)  ((x)->pipe->p256)
#define hashState384(x)  ((x)->pipe->p512)
#define hashState512(x)  ((x)->pipe->p512)

/* Components used for 224 and 256 bit version */
#define s32_0(x)  (shr((x), 1) ^ shl((x), 3) ^ rotl32((x),  4) ^ rotl32((x), 19))
#define s32_1(x)  (shr((x), 1) ^ shl((x), 2) ^ rotl32((x),  8) ^ rotl32((x), 23))
#define s32_2(x)  (shr((x), 2) ^ shl((x), 1) ^ rotl32((x), 12) ^ rotl32((x), 25))
#define s32_3(x)  (shr((x), 2) ^ shl((x), 2) ^ rotl32((x), 15) ^ rotl32((x), 29))
#define s32_4(x)  (shr((x), 1) ^ (x))
#define s32_5(x)  (shr((x), 2) ^ (x))
#define r32_01(x) rotl32((x),  3)
#define r32_02(x) rotl32((x),  7)
#define r32_03(x) rotl32((x), 13)
#define r32_04(x) rotl32((x), 16)
#define r32_05(x) rotl32((x), 19)
#define r32_06(x) rotl32((x), 23)
#define r32_07(x) rotl32((x), 27)

Timing t_gpu_256(8);

#ifdef _DEBUG
float  *timing_gpu_256 = t_gpu_256.gettiming();
int  *counter_gpu_256 = t_gpu_256.getcounter();
#endif

/*
**
** CUDA code
**
*/

unsigned char __constant__ idx_f0[80];
unsigned char __constant__ sign_f0[64];
/*
unsigned char __constant__ idx_f2_sh0[16];
unsigned char __constant__ idx_f2_sh0value[16];
unsigned char __constant__ idx_f2_sh1value[16];
unsigned char __constant__ idx_f2_sh2[16];
unsigned char __constant__ idx_f2_sh2value[16];
unsigned char __constant__ idx_f2_p256value[16];
*/
u_int32_t *data32_dev, *p256_dev;

const unsigned char
idx_f0_host[80] = { 5, 6,10,13,14,
                    6, 8,11,14,15,
                    0, 7, 9,12,15,
                    0, 1, 8,10,13,
                    1, 2, 9,11,14,
                    3, 2,10,12,15,
                    4, 0, 3,11,13,
                    1, 4, 5,12,14,
                    2, 5, 6,13,15,
                    0, 3, 6, 7,14,
                    8, 1, 4, 7,15,
                    8, 0, 2, 5, 9,
                    1, 3, 6, 9,10,
                    2, 4, 7,10,11,
                    3, 5, 8,11,12,
                    2, 4, 6, 9,13 };
```

```
const char
sign_f0_host[64] = { -1,+1,+1,+1,
                     -1,+1,+1,-1,
                     +1,+1,-1,+1,
                     -1,+1,-1,+1,
                     +1,+1,-1,-1,
                     -1,+1,-1,+1,
                     -1,-1,-1,+1,
                     -1,-1,-1,-1,
                     -1,-1,+1,-1,
                     -1,+1,-1,+1,
                     -1,-1,-1,+1,
                     -1,-1,-1,+1,
                     +1,-1,-1,+1,
                     +1,+1,+1,+1,
                     -1,+1,-1,-1,
                     -1,-1,-1,+1 };
/*
const unsigned char
idx_f2_sh0_host[16]       = { 0, 1, 1, 1,
                              1, 0, 1, 1,
                              1, 1, 1, 1,
                              1, 1, 1, 1 };
const unsigned char
idx_f2_sh0value_host[16]  = { 5, 7, 5, 1,
                              3, 6, 4,11,
                              0, 0, 0, 0,
                              0, 0, 0, 0 };
const unsigned char
idx_f2_sh1value_host[16]  = { 5, 8, 5, 5,
                              0, 6, 6, 2,
                              0, 0, 0, 0,
                              0, 0, 0, 0 };
const unsigned char
idx_f2_sh2_host[16]       = { 1, 1, 1, 1,
                              1, 1, 1, 1,
                              0, 1, 0, 0,
                              1, 1, 1, 1 };
const unsigned char
idx_f2_sh2value_host[16]  = { 8, 6, 6, 4,
                              3, 4, 7, 2,
                              0, 0, 0, 0,
                              0, 0, 0, 0 };
const unsigned char
idx_f2_p256value_host[16] = {24,25,26,27,
                             28,29,30,31,
                             23,16,17,18,
                             19,20,21,22 };
*/

void AllocCUDA(void) {
    cudaMalloc((void**)&data32_dev,16*sizeof(u_int32_t));
    cudaMalloc((void**)&p256_dev,32*sizeof(u_int32_t));

    cudaMemcpyToSymbol(idx_f0, idx_f0_host, sizeof(idx_f0));
    cudaMemcpyToSymbol(sign_f0, sign_f0_host, sizeof(sign_f0));
/*
    cudaMemcpyToSymbol(idx_f2_sh0, idx_f2_sh0_host, sizeof(idx_f2_sh0));
    cudaMemcpyToSymbol(idx_f2_sh0value, idx_f2_sh0value_host, sizeof(idx_f2_sh0value));
    cudaMemcpyToSymbol(idx_f2_sh1value, idx_f2_sh1value_host, sizeof(idx_f2_sh1value));
    cudaMemcpyToSymbol(idx_f2_sh2, idx_f2_sh2_host, sizeof(idx_f2_sh2));
    cudaMemcpyToSymbol(idx_f2_sh2value, idx_f2_sh2value_host, sizeof(idx_f2_sh2value));
    cudaMemcpyToSymbol(idx_f2_p256value, idx_f2_p256value_host, sizeof(idx_f2_p256value));
*/
```

```
}

void FreeCUDA(void) {
    cudaFree(data32_dev);
    cudaFree(p256_dev);
}

__global__ void kernel_f0(u_int32_t *data32, u_int32_t *p256) {
    u_int32_t thread = threadIdx.x;
    u_int32_t idx=thread*5,sign=thread*4;

    //extern __shared__ u_int32_t s_data[];
    //u_int32_t *data32,*p256;

    //data32 = s_data;
    //p256   = &data32[16];

    //data32[thread]   = data32_dev[thread];
    //p256[thread]     = p256_dev[thread];
    //p256[thread+16]  = p256_dev[thread+16];
    //__syncthreads();
    /*  Mix the message block with the previous double pipe.       */
    p256[thread]    ^= data32[thread];
    __syncthreads();
    p256[16+thread]  =                 p256[idx_f0[idx + 0]] +
                       sign_f0[sign+0]*p256[idx_f0[idx + 1]] +
                       sign_f0[sign+1]*p256[idx_f0[idx + 2]] +
                       sign_f0[sign+2]*p256[idx_f0[idx + 3]] +
                       sign_f0[sign+3]*p256[idx_f0[idx + 4]];
    __syncthreads();
    switch(thread % 5)
    {
        case 0: p256[thread] = s32_0(p256[16+thread]); break;
        case 1: p256[thread] = s32_1(p256[16+thread]); break;
        case 2: p256[thread] = s32_2(p256[16+thread]); break;
        case 3: p256[thread] = s32_3(p256[16+thread]); break;
        case 4: p256[thread] = s32_4(p256[16+thread]); break;
    }
    //p256_dev[thread]    = p256[thread];
    //p256_dev[thread+16] = p256[thread+16];
}

__global__ void kernel_f2(u_int32_t XL32_host, u_int32_t XH32_host, u_int32_t *data32, u_int32_t
*p256) {
    u_int32_t thread = threadIdx.x;
    u_int32_t XL32 = XL32_host;
    u_int32_t XH32 = XH32_host;
    //u_int32_t XL32_sh[2][16];
    //u_int32_t XH32_sh[2][16];

    //extern __shared__ u_int32_t s_data[];
    //u_int32_t *data32,*p256;

    //data32 = s_data;
    //p256   = &data32[16];

    //data32[thread]   = data32_dev[thread];
    //p256[thread]     = p256_dev[thread];
    //p256[thread+16]  = p256_dev[thread+16];

    /*
    XL32_sh[0][thread] = shl(XL32, thread);
    XL32_sh[1][thread] = shr(XL32, thread);
    XH32_sh[0][thread] = shl(XH32, thread);
    XH32_sh[1][thread] = shr(XH32, thread);
```

```
    if(idx_f2_sh0[thread]) {
        p256[thread] = (XH32_sh[idx_f2_sh0[thread]][idx_f2_sh0value[thread]]  ^
                        shr(p256[thread+16],idx_f2_sh1value[thread])          ^
                        data32[thread])                                       +
                       (XL32_sh[idx_f2_sh2[thread]][idx_f2_sh2value[thread]]  ^
                        p256[idx_f2_p256value[thread]] ^ p256[thread]);
    }
    else {
        p256[thread] = (XH32_sh[idx_f2_sh0[thread]][idx_f2_sh0value[thread]]  ^
                        shr(p256[thread+16],idx_f2_sh1value[thread])          ^
                        data32[thread])                                       +
                       (XL32_sh[idx_f2_sh2[thread]][idx_f2_sh2value[thread]]  ^
                        p256[idx_f2_p256value[thread]] ^ p256[thread]);
    }
    */

    /*  Compute the double chaining pipe for the next message block. */
    switch(thread)
    {
        case 0:  p256[0] =  (shl(XH32, 5) ^ shr(p256[16],5) ^ data32[ 0]) + (    XL32    ^
p256[24] ^ p256[ 0]); break;
        case 1:  p256[1] =  (shr(XH32, 7) ^ shl(p256[17],8) ^ data32[ 1]) + (    XL32    ^
p256[25] ^ p256[ 1]); break;
        case 2:  p256[2] =  (shr(XH32, 5) ^ shl(p256[18],5) ^ data32[ 2]) + (    XL32    ^
p256[26] ^ p256[ 2]); break;
        case 3:  p256[3] =  (shr(XH32, 1) ^ shl(p256[19],5) ^ data32[ 3]) + (    XL32    ^
p256[27] ^ p256[ 3]); break;
        case 4:  p256[4] =  (shr(XH32, 3) ^     p256[20]    ^ data32[ 4]) + (    XL32    ^
p256[28] ^ p256[ 4]); break;
        case 5:  p256[5] =  (shl(XH32, 6) ^ shr(p256[21],6) ^ data32[ 5]) + (    XL32    ^
p256[29] ^ p256[ 5]); break;
        case 6:  p256[6] =  (shr(XH32, 4) ^ shl(p256[22],6) ^ data32[ 6]) + (    XL32    ^
p256[30] ^ p256[ 6]); break;
        case 7:  p256[7] =  (shr(XH32,11) ^ shl(p256[23],2) ^ data32[ 7]) + (    XL32    ^
p256[31] ^ p256[ 7]); break;
        case 8:  p256[ 8] = (    XH32     ^     p256[24]    ^ data32[ 8]) + (shl(XL32,8) ^
p256[23] ^ p256[ 8]); break;
        case 9:  p256[ 9] = (    XH32     ^     p256[25]    ^ data32[ 9]) + (shr(XL32,6) ^
p256[16] ^ p256[ 9]); break;
        case 10: p256[10] = (    XH32     ^     p256[26]    ^ data32[10]) + (shl(XL32,6) ^
p256[17] ^ p256[10]); break;
        case 11: p256[11] = (    XH32     ^     p256[27]    ^ data32[11]) + (shl(XL32,4) ^
p256[18] ^ p256[11]); break;
        case 12: p256[12] = (    XH32     ^     p256[28]    ^ data32[12]) + (shr(XL32,3) ^
p256[19] ^ p256[12]); break;
        case 13: p256[13] = (    XH32     ^     p256[29]    ^ data32[13]) + (shr(XL32,4) ^
p256[20] ^ p256[13]); break;
        case 14: p256[14] = (    XH32     ^     p256[30]    ^ data32[14]) + (shr(XL32,7) ^
p256[21] ^ p256[14]); break;
        case 15: p256[15] = (    XH32     ^     p256[31]    ^ data32[15]) + (shr(XL32,2) ^
p256[22] ^ p256[15]); break;
    }
    __syncthreads();
    if(thread < 8) p256[thread+8] += rotl32(p256[(thread+4)%8],thread+9);

    //p256_dev[thread]    = p256[thread];
    //p256_dev[thread+16] = p256[thread+16];
}

#define Compression256()\
{\
    t_gpu_256.DoTiming1();\
    cudaMemcpy(data32_dev, data32, 16*sizeof(u_int32_t), cudaMemcpyHostToDevice);\
    t_gpu_256.DoTiming2(0);\
```

```
    t_gpu_256.DoTiming1();\
    cudaMemcpy(p256_dev, p256, 16*sizeof(u_int32_t), cudaMemcpyHostToDevice);\
    t_gpu_256.DoTiming2(1);\
    t_gpu_256.DoTiming1();\
    kernel_f0<<<1,16>>>(data32_dev, p256_dev);\
    t_gpu_256.DoTiming2(2);\
    t_gpu_256.DoTiming1();\
    cudaMemcpy(p256, p256_dev, 32*sizeof(u_int32_t), cudaMemcpyDeviceToHost);\
    t_gpu_256.DoTiming2(3);\
\
    /* This is the Message expansion. */\
    /* It has 16 rounds.              */\
    p256[16] = s32_1(p256[ 0])    + s32_2(p256[ 1]) + s32_3(p256[ 2]) + s32_0(p256[ 3])\
             + s32_1(p256[ 4])    + s32_2(p256[ 5]) + s32_3(p256[ 6]) + s32_0(p256[ 7])\
             + s32_1(p256[ 8])    + s32_2(p256[ 9]) + s32_3(p256[10]) + s32_0(p256[11])\
             + s32_1(p256[12])    + s32_2(p256[13]) + s32_3(p256[14]) + s32_0(p256[15])\
             + 0x55555550ul + data32[0]      + data32[3]      - data32[10];\
    XL32 =  p256[16];\
    p256[17] = s32_1(p256[ 1])    + s32_2(p256[ 2]) + s32_3(p256[ 3]) + s32_0(p256[ 4])\
             + s32_1(p256[ 5])    + s32_2(p256[ 6]) + s32_3(p256[ 7]) + s32_0(p256[ 8])\
             + s32_1(p256[ 9])    + s32_2(p256[10]) + s32_3(p256[11]) + s32_0(p256[12])\
             + s32_1(p256[13])    + s32_2(p256[14]) + s32_3(p256[15]) + s32_0(p256[16])\
             + 0x5aaaaaa5ul + data32[1]      + data32[4]      - data32[11];\
    XL32 ^= p256[17];\
    TempEven32 = p256[14]  + p256[12]  + p256[10]  + p256[ 8]  + p256[ 6]  + p256[ 4]  + p256[
2];\
    TempOdd32  = p256[15]  + p256[13]  + p256[11]  + p256[ 9]  + p256[ 7]  + p256[ 5]  + p256[
3];\
\
    /* expand32_22(18); */\
    p256[18] = TempEven32 + r32_01(p256[ 3])  + r32_02(p256[ 5])\
                          + r32_03(p256[ 7])  + r32_04(p256[ 9])\
                          + r32_05(p256[11])  + r32_06(p256[13])\
                          + r32_07(p256[15])  + s32_5( p256[16] ) + s32_4( p256[17])\
                          + 0x5ffffffaul + data32[2] + data32[5] - data32[12];\
    XL32 ^= p256[18];\
    /* expand32_21(19); */\
    p256[19] = TempOdd32  + r32_01(p256[ 4])  + r32_02(p256[ 6])\
                          + r32_03(p256[ 8])  + r32_04(p256[10])\
                          + r32_05(p256[12])  + r32_06(p256[14])\
                          + r32_07(p256[16])  + s32_5( p256[17] ) + s32_4( p256[18])\
                          + 0x6555554ful + data32[3] + data32[6] - data32[13];\
    XL32 ^= p256[19];\
    TempEven32+=p256[16]; TempEven32-=p256[ 2];\
    /* expand32_22(20); */\
    p256[20] = TempEven32 + r32_01(p256[ 5])  + r32_02(p256[ 7])\
                          + r32_03(p256[ 9])  + r32_04(p256[11])\
                          + r32_05(p256[13])  + r32_06(p256[15])\
                          + r32_07(p256[17])  + s32_5( p256[18] ) + s32_4( p256[19])\
                          + 0x6aaaaaa4ul + data32[4] + data32[7] - data32[14];\
    XL32 ^= p256[20];\
    TempOdd32 +=p256[17]; TempOdd32 -=p256[ 3];\
    /* expand32_21(21); */\
    p256[21] = TempOdd32  + r32_01(p256[ 6])  + r32_02(p256[ 8])\
                          + r32_03(p256[10])  + r32_04(p256[12])\
                          + r32_05(p256[14])  + r32_06(p256[16])\
                          + r32_07(p256[18])  + s32_5( p256[19] ) + s32_4( p256[20])\
                          + 0x6ffffff9ul + data32[5] + data32[8] - data32[15];\
    XL32 ^= p256[21];\
    TempEven32+=p256[18]; TempEven32-=p256[ 4];\
    /* expand32_22(22); */\
    p256[22] = TempEven32 + r32_01(p256[ 7])  + r32_02(p256[ 9])\
                          + r32_03(p256[11])  + r32_04(p256[13])\
                          + r32_05(p256[15])  + r32_06(p256[17])\
                          + r32_07(p256[19])  + s32_5( p256[20] ) + s32_4( p256[21])\
```

```
                    + 0x7555554eul + data32[6] + data32[9] - data32[ 0];\
XL32 ^= p256[22];\
TempOdd32 +=p256[19]; TempOdd32 -=p256[ 5];\
/* expand32_21(23); */\
p256[23] = TempOdd32  + r32_01(p256[ 8])  + r32_02(p256[10])\
                    + r32_03(p256[12])  + r32_04(p256[14])\
                    + r32_05(p256[16])  + r32_06(p256[18])\
                    + r32_07(p256[20])  + s32_5( p256[21] ) + s32_4( p256[22])\
                    + 0x7aaaaaa3ul + data32[7] + data32[10] - data32[ 1];\
XL32 ^= p256[23];\
TempEven32+=p256[20]; TempEven32-=p256[ 6];\
/* expand32_22(24); */\
p256[24] = TempEven32 + r32_01(p256[ 9])  + r32_02(p256[11])\
                    + r32_03(p256[13])  + r32_04(p256[15])\
                    + r32_05(p256[17])  + r32_06(p256[19])\
                    + r32_07(p256[21])  + s32_5( p256[22] ) + s32_4( p256[23])\
                    + 0x7ffffff8ul + data32[8] + data32[11] - data32[ 2];\
XH32 =  XL32^p256[24];\
TempOdd32 +=p256[21]; TempOdd32 -=p256[ 7];\
/* expand32_21(25); */\
p256[25] = TempOdd32   + r32_01(p256[10])  + r32_02(p256[12])\
                    + r32_03(p256[14])  + r32_04(p256[16])\
                    + r32_05(p256[18])  + r32_06(p256[20])\
                    + r32_07(p256[22])  + s32_5( p256[23] ) + s32_4( p256[24])\
                    + 0x8555554dul + data32[9] + data32[12] - data32[ 3];\
XH32 ^= p256[25];\
TempEven32+=p256[22]; TempEven32-=p256[ 8];\
/* expand32_22(26); */\
p256[26] = TempEven32 + r32_01(p256[11])  + r32_02(p256[13])\
                    + r32_03(p256[15])  + r32_04(p256[17])\
                    + r32_05(p256[19])  + r32_06(p256[21])\
                    + r32_07(p256[23])  + s32_5( p256[24] ) + s32_4( p256[25])\
                    + 0x8aaaaaa2ul + data32[10] + data32[13] - data32[ 4];\
XH32 ^= p256[26];\
TempOdd32 +=p256[23]; TempOdd32 -=p256[ 9];\
/* expand32_21(27); */\
p256[27] = TempOdd32  + r32_01(p256[12])  + r32_02(p256[14])\
                    + r32_03(p256[16])  + r32_04(p256[18])\
                    + r32_05(p256[20])  + r32_06(p256[22])\
                    + r32_07(p256[24])  + s32_5( p256[25] ) + s32_4( p256[26])\
                    + 0x8ffffff7ul + data32[11] + data32[14] - data32[ 5];\
XH32 ^= p256[27];\
TempEven32+=p256[24]; TempEven32-=p256[10];\
/* expand32_22(28); */\
p256[28] = TempEven32 + r32_01(p256[13])  + r32_02(p256[15])\
                    + r32_03(p256[17])  + r32_04(p256[19])\
                    + r32_05(p256[21])  + r32_06(p256[23])\
                    + r32_07(p256[25])  + s32_5( p256[26] ) + s32_4( p256[27])\
                    + 0x9555554cul + data32[12] + data32[15] - data32[ 6];\
XH32 ^= p256[28];\
TempOdd32 +=p256[25]; TempOdd32 -=p256[11];\
/* expand32_21(29); */\
p256[29] = TempOdd32  + r32_01(p256[14])  + r32_02(p256[16])\
                    + r32_03(p256[18])  + r32_04(p256[20])\
                    + r32_05(p256[22])  + r32_06(p256[24])\
                    + r32_07(p256[26])  + s32_5( p256[27] ) + s32_4( p256[28])\
                    + 0x9aaaaaa1ul + data32[13] + data32[ 0] - data32[ 7];\
XH32 ^= p256[29];\
TempEven32+=p256[26]; TempEven32-=p256[12];\
/* expand32_22(30); */\
p256[30] = TempEven32 + r32_01(p256[15])  + r32_02(p256[17])\
                    + r32_03(p256[19])  + r32_04(p256[21])\
                    + r32_05(p256[23])  + r32_06(p256[25])\
                    + r32_07(p256[27])  + s32_5( p256[28] ) + s32_4( p256[29])\
                    + 0x9ffffff6ul + data32[14] + data32[ 1] - data32[ 8];\
```

```
    XH32 ^= p256[30];\
    TempOdd32 +=p256[27]; TempOdd32 -=p256[13];\
    /* expand32_21(31); */\
    p256[31] = TempOdd32  + r32_01(p256[16])  + r32_02(p256[18])\
                          + r32_03(p256[20])  + r32_04(p256[22])\
                          + r32_05(p256[24])  + r32_06(p256[26])\
                          + r32_07(p256[28])  + s32_5( p256[29] ) + s32_4( p256[30])\
                          + 0xa555554bul + data32[15] + data32[ 2] - data32[ 9];\
    XH32 ^= p256[31];\
\
    t_gpu_256.DoTiming1();\
    cudaMemcpy(data32_dev, data32, 16*sizeof(u_int32_t), cudaMemcpyHostToDevice);\
    t_gpu_256.DoTiming2(4);\
    t_gpu_256.DoTiming1();\
    cudaMemcpy(p256_dev, p256, 16*sizeof(u_int32_t), cudaMemcpyHostToDevice);\
    t_gpu_256.DoTiming2(5);\
    t_gpu_256.DoTiming1();\
    kernel_f2<<<1,16>>>(XL32, XH32, data32_dev, p256_dev);\
    t_gpu_256.DoTiming2(6);\
    t_gpu_256.DoTiming1();\
    cudaMemcpy(p256, p256_dev, 32*sizeof(u_int32_t), cudaMemcpyDeviceToHost);\
    t_gpu_256.DoTiming2(7);\
}

#define s64_0(x)  (shr((x), 1) ^ shl((x), 3) ^ rotl64((x),  4) ^ rotl64((x), 37))
#define s64_1(x)  (shr((x), 1) ^ shl((x), 2) ^ rotl64((x), 13) ^ rotl64((x), 43))
#define s64_2(x)  (shr((x), 2) ^ shl((x), 1) ^ rotl64((x), 19) ^ rotl64((x), 53))
#define s64_3(x)  (shr((x), 2) ^ shl((x), 2) ^ rotl64((x), 28) ^ rotl64((x), 59))
#define s64_4(x)  (shr((x), 1) ^ (x))
#define s64_5(x)  (shr((x), 2) ^ (x))
#define r64_01(x) rotl64((x),  5)
#define r64_02(x) rotl64((x), 11)
#define r64_03(x) rotl64((x), 27)
#define r64_04(x) rotl64((x), 32)
#define r64_05(x) rotl64((x), 37)
#define r64_06(x) rotl64((x), 43)
#define r64_07(x) rotl64((x), 53)

#define Compression512()\
{\
    p512_00=p512[ 0]^data64[ 0]; p512_01=p512[ 1]^data64[ 1]; p512_02=p512[ 2]^data64[ 2];
p512_03=p512[ 3]^data64[ 3];\
    p512_04=p512[ 4]^data64[ 4]; p512_05=p512[ 5]^data64[ 5]; p512_06=p512[ 6]^data64[ 6];
p512_07=p512[ 7]^data64[ 7];\
    p512_08=p512[ 8]^data64[ 8]; p512_09=p512[ 9]^data64[ 9]; p512_10=p512[10]^data64[10];
p512_11=p512[11]^data64[11];\
    p512_12=p512[12]^data64[12]; p512_13=p512[13]^data64[13]; p512_14=p512[14]^data64[14];
p512_15=p512[15]^data64[15];\
\
    p512_16 = ( p512_05-p512_07+p512_10+p512_13+p512_14);\
    p512_17 = ( p512_06-p512_08+p512_11+p512_14-p512_15);\
    p512_18 = ( p512_00+p512_07+p512_09-p512_12+p512_15);\
    p512_19 = ( p512_00-p512_01+p512_08-p512_10+p512_13);\
    p512_20 = ( p512_01+p512_02+p512_09-p512_11-p512_14);\
    p512_21 = ( p512_03-p512_02+p512_10-p512_12+p512_15);\
    p512_22 = ( p512_04-p512_00-p512_03-p512_11+p512_13);\
    p512_23 = ( p512_01-p512_04-p512_05-p512_12-p512_14);\
    p512_24 = ( p512_02-p512_05-p512_06+p512_13-p512_15);\
    p512_25 = ( p512_00-p512_03+p512_06-p512_07+p512_14);\
    p512_26 = ( p512_08-p512_01-p512_04-p512_07+p512_15);\
    p512_27 = ( p512_08-p512_00-p512_02-p512_05+p512_09);\
    p512_28 = ( p512_01+p512_03-p512_06-p512_09+p512_10);\
    p512_29 = ( p512_02+p512_04+p512_07+p512_10+p512_11);\
    p512_30 = ( p512_03-p512_05+p512_08-p512_11-p512_12);\
    p512_31 = ( p512_12-p512_04-p512_06-p512_09+p512_13);\
```

```
\
    p512_00  = s64_0(p512_16);\
    p512_01  = s64_1(p512_17);\
    p512_02  = s64_2(p512_18);\
    p512_03  = s64_3(p512_19);\
    p512_04  = s64_4(p512_20);\
    p512_05  = s64_0(p512_21);\
    p512_06  = s64_1(p512_22);\
    p512_07  = s64_2(p512_23);\
    p512_08  = s64_3(p512_24);\
    p512_09  = s64_4(p512_25);\
    p512_10  = s64_0(p512_26);\
    p512_11  = s64_1(p512_27);\
    p512_12  = s64_2(p512_28);\
    p512_13  = s64_3(p512_29);\
    p512_14  = s64_4(p512_30);\
    p512_15  = s64_0(p512_31);\
\
    /* This is the Message expansion. */\
    /* It has 16 rounds.              */\
    p512_16  = s64_1(p512_00)    + s64_2(p512_01) + s64_3(p512_02) + s64_0(p512_03)\
             + s64_1(p512_04)    + s64_2(p512_05) + s64_3(p512_06) + s64_0(p512_07)\
             + s64_1(p512_08)    + s64_2(p512_09) + s64_3(p512_10) + s64_0(p512_11)\
             + s64_1(p512_12)    + s64_2(p512_13) + s64_3(p512_14) + s64_0(p512_15)\
             + 0x5555555555555550ull + data64[0]      + data64[3]      - data64[10];\
    XL64 =  p512_16;\
    p512_17  = s64_1(p512_01)    + s64_2(p512_02) + s64_3(p512_03) + s64_0(p512_04)\
             + s64_1(p512_05)    + s64_2(p512_06) + s64_3(p512_07) + s64_0(p512_08)\
             + s64_1(p512_09)    + s64_2(p512_10) + s64_3(p512_11) + s64_0(p512_12)\
             + s64_1(p512_13)    + s64_2(p512_14) + s64_3(p512_15) + s64_0(p512_16)\
             + 0x5aaaaaaaaaaaaaa5ull + data64[1]      + data64[4]      - data64[11];\
    XL64 ^= p512_17;\
    TempEven64 = p512_14  + p512_12  + p512_10  + p512_08  + p512_06  + p512_04  + p512_02;\
    TempOdd64  = p512_15  + p512_13  + p512_11  + p512_09  + p512_07  + p512_05  + p512_03;\
\
    /* expand64_22(18); */\
    p512_18  = TempEven64 + r64_01(p512_03)  + r64_02(p512_05)\
                          + r64_03(p512_07)  + r64_04(p512_09)\
                          + r64_05(p512_11)  + r64_06(p512_13)\
                          + r64_07(p512_15)  + s64_5( p512_16 ) + s64_4( p512_17)\
                          + 0x5fffffffffffffaull + data64[2] + data64[5] - data64[12];\
    XL64 ^= p512_18;\
    /* expand64_21(19); */\
    p512_19 = TempOdd64   + r64_01(p512_04)  + r64_02(p512_06)\
                          + r64_03(p512_08)  + r64_04(p512_10)\
                          + r64_05(p512_12)  + r64_06(p512_14)\
                          + r64_07(p512_16)  + s64_5( p512_17 ) + s64_4( p512_18)\
                          + 0x655555555555554full + data64[3] + data64[6] - data64[13];\
    XL64 ^= p512_19;\
    TempEven64+=p512_16; TempEven64-=p512_02;\
    /* expand64_22(20); */\
    p512_20  = TempEven64 + r64_01(p512_05)  + r64_02(p512_07)\
                          + r64_03(p512_09)  + r64_04(p512_11)\
                          + r64_05(p512_13)  + r64_06(p512_15)\
                          + r64_07(p512_17)  + s64_5( p512_18 ) + s64_4( p512_19)\
                          + 0x6aaaaaaaaaaaaaa4ull + data64[4] + data64[7] - data64[14];\
    XL64 ^= p512_20;\
    TempOdd64 +=p512_17; TempOdd64 -=p512_03;\
    /* expand64_21(21); */\
    p512_21 = TempOdd64   + r64_01(p512_06)  + r64_02(p512_08)\
                          + r64_03(p512_10)  + r64_04(p512_12)\
                          + r64_05(p512_14)  + r64_06(p512_16)\
                          + r64_07(p512_18)  + s64_5( p512_19 ) + s64_4( p512_20)\
                          + 0x6ffffffffffffff9ull + data64[5] + data64[8] - data64[15];\
    XL64 ^= p512_21;\
```

```
TempEven64+=p512_18; TempEven64-=p512_04;\
/* expand64_22(22); */\
p512_22  = TempEven64 + r64_01(p512_07)  + r64_02(p512_09)\
                      + r64_03(p512_11)  + r64_04(p512_13)\
                      + r64_05(p512_15)  + r64_06(p512_17)\
                      + r64_07(p512_19)  + s64_5( p512_20 ) + s64_4( p512_21)\
                      + 0x755555555555554eull + data64[6] + data64[9] - data64[ 0];\
XL64 ^= p512_22;\
TempOdd64 +=p512_19; TempOdd64 -=p512_05;\
/* expand64_21(23); */\
p512_23 = TempOdd64   + r64_01(p512_08)  + r64_02(p512_10)\
                      + r64_03(p512_12)  + r64_04(p512_14)\
                      + r64_05(p512_16)  + r64_06(p512_18)\
                      + r64_07(p512_20)  + s64_5( p512_21 ) + s64_4( p512_22)\
                      + 0x7aaaaaaaaaaaaaa3ull + data64[7] + data64[10] - data64[ 1];\
XL64 ^= p512_23;\
TempEven64+=p512_20; TempEven64-=p512_06;\
/* expand64_22(24); */\
p512_24  = TempEven64 + r64_01(p512_09)  + r64_02(p512_11)\
                      + r64_03(p512_13)  + r64_04(p512_15)\
                      + r64_05(p512_17)  + r64_06(p512_19)\
                      + r64_07(p512_21)  + s64_5( p512_22 ) + s64_4( p512_23)\
                      + 0x7ffffffffffffff8ull + data64[8] + data64[11] - data64[ 2];\
XH64 =  XL64^p512_24;\
TempOdd64 +=p512_21; TempOdd64 -=p512_07;\
/* expand64_21(25); */\
p512_25 = TempOdd64   + r64_01(p512_10)  + r64_02(p512_12)\
                      + r64_03(p512_14)  + r64_04(p512_16)\
                      + r64_05(p512_18)  + r64_06(p512_20)\
                      + r64_07(p512_22)  + s64_5( p512_23 ) + s64_4( p512_24)\
                      + 0x855555555555554dull + data64[9] + data64[12] - data64[ 3];\
XH64 ^= p512_25;\
TempEven64+=p512_22; TempEven64-=p512_08;\
/* expand64_22(26); */\
p512_26  = TempEven64 + r64_01(p512_11)  + r64_02(p512_13)\
                      + r64_03(p512_15)  + r64_04(p512_17)\
                      + r64_05(p512_19)  + r64_06(p512_21)\
                      + r64_07(p512_23)  + s64_5( p512_24 ) + s64_4( p512_25)\
                      + 0x8aaaaaaaaaaaaaa2ull + data64[10] + data64[13] - data64[ 4];\
XH64 ^= p512_26;\
TempOdd64 +=p512_23; TempOdd64 -=p512_09;\
/* expand64_21(27); */\
p512_27 = TempOdd64   + r64_01(p512_12)  + r64_02(p512_14)\
                      + r64_03(p512_16)  + r64_04(p512_18)\
                      + r64_05(p512_20)  + r64_06(p512_22)\
                      + r64_07(p512_24)  + s64_5( p512_25 ) + s64_4( p512_26)\
                      + 0x8fffffffffffff7ull + data64[11] + data64[14] - data64[ 5];\
XH64 ^= p512_27;\
TempEven64+=p512_24; TempEven64-=p512_10;\
/* expand64_22(28); */\
p512_28  = TempEven64 + r64_01(p512_13)  + r64_02(p512_15)\
                      + r64_03(p512_17)  + r64_04(p512_19)\
                      + r64_05(p512_21)  + r64_06(p512_23)\
                      + r64_07(p512_25)  + s64_5( p512_26 ) + s64_4( p512_27)\
                      + 0x955555555555554cull + data64[12] + data64[15] - data64[ 6];\
XH64 ^= p512_28;\
TempOdd64 +=p512_25; TempOdd64 -=p512_11;\
/* expand64_21(29); */\
p512_29 = TempOdd64   + r64_01(p512_14)  + r64_02(p512_16)\
                      + r64_03(p512_18)  + r64_04(p512_20)\
                      + r64_05(p512_22)  + r64_06(p512_24)\
                      + r64_07(p512_26)  + s64_5( p512_27 ) + s64_4( p512_28)\
                      + 0x9aaaaaaaaaaaaaa1ull + data64[13] + data64[ 0] - data64[ 7];\
XH64 ^= p512_29;\
TempEven64+=p512_26; TempEven64-=p512_12;\
```

```
    /* expand64_22(30); */\
    p512_30  = TempEven64 + r64_01(p512_15)  + r64_02(p512_17)\
              + r64_03(p512_19)  + r64_04(p512_21)\
                          + r64_05(p512_23)  + r64_06(p512_25)\
                          + r64_07(p512_27)  + s64_5( p512_28 ) + s64_4( p512_29)\
                          + 0x9fffffffffffff6ull + data64[14] + data64[ 1] - data64[ 8];\
    XH64 ^= p512_30;\
    TempOdd64 +=p512_27; TempOdd64 -=p512_13;\
    /* expand64_21(31); */\
    p512_31 = TempOdd64   + r64_01(p512_16)  + r64_02(p512_18)\
                          + r64_03(p512_20)  + r64_04(p512_22)\
                          + r64_05(p512_24)  + r64_06(p512_26)\
                          + r64_07(p512_28)  + s64_5( p512_29 ) + s64_4( p512_30)\
                          + 0xa55555555555554bull + data64[15] + data64[ 2] - data64[ 9];\
    XH64 ^= p512_31;\
\
    /*  Compute the double chaining pipe for the next message block. */\
    p512[0] =                       (shl(XH64, 5) ^ shr(p512_16,5) ^ data64[ 0]) + (    XL64    ^
p512_24 ^ p512_00);\
    p512[1] =                       (shr(XH64, 7) ^ shl(p512_17,8) ^ data64[ 1]) + (    XL64    ^
p512_25 ^ p512_01);\
    p512[2] =                       (shr(XH64, 5) ^ shl(p512_18,5) ^ data64[ 2]) + (    XL64    ^
p512_26 ^ p512_02);\
    p512[3] =                       (shr(XH64, 1) ^ shl(p512_19,5) ^ data64[ 3]) + (    XL64    ^
p512_27 ^ p512_03);\
    p512[4] =                       (shr(XH64, 3) ^     p512_20    ^ data64[ 4]) + (    XL64    ^
p512_28 ^ p512_04);\
    p512[5] =                       (shl(XH64, 6) ^ shr(p512_21,6) ^ data64[ 5]) + (    XL64    ^
p512_29 ^ p512_05);\
    p512[6] =                       (shr(XH64, 4) ^ shl(p512_22,6) ^ data64[ 6]) + (    XL64    ^
p512_30 ^ p512_06);\
    p512[7] =                       (shr(XH64,11) ^ shl(p512_23,2) ^ data64[ 7]) + (    XL64    ^
p512_31 ^ p512_07);\
\
    p512[ 8] = rotl64(p512[4], 9) + (    XH64     ^     p512_24    ^ data64[ 8]) + (shl(XL64,8) ^
p512_23 ^ p512_08);\
                p512[ 9] = rotl64(p512[5],10) + (    XH64     ^     p512_25    ^ data64[ 9]) +
(shr(XL64,6) ^ p512_16 ^ p512_09);\
    p512[10] = rotl64(p512[6],11) + (    XH64     ^     p512_26    ^ data64[10]) + (shl(XL64,6) ^
p512_17 ^ p512_10);\
    p512[11] = rotl64(p512[7],12) + (    XH64     ^     p512_27    ^ data64[11]) + (shl(XL64,4) ^
p512_18 ^ p512_11);\
    p512[12] = rotl64(p512[0],13) + (    XH64     ^     p512_28    ^ data64[12]) + (shr(XL64,3) ^
p512_19 ^ p512_12);\
    p512[13] = rotl64(p512[1],14) + (    XH64     ^     p512_29    ^ data64[13]) + (shr(XL64,4) ^
p512_20 ^ p512_13);\
    p512[14] = rotl64(p512[2],15) + (    XH64     ^     p512_30    ^ data64[14]) + (shr(XL64,7) ^
p512_21 ^ p512_14);\
    p512[15] = rotl64(p512[3],16) + (    XH64     ^     p512_31    ^ data64[15]) + (shr(XL64,2) ^
p512_22 ^ p512_15);\
}



HashReturn InitGPU(hashState *state, int hashbitlen)
{
    switch(hashbitlen)
    {
        case 224:
        state->hashbitlen = 224;
        // #1 Between comments #1 and #2 add algorithm specific initialization
        state->bits_processed = 0;
        state->unprocessed_bits = 0;
        memcpy(hashState224(state)->DoublePipe, i224p2,  16 * sizeof(u_int32_t));
        // #2 Between comments #1 and #2 add algorithm specific initialization
```

```
        return(SUCCESS);

        case 256:
        state->hashbitlen = 256;
        // #1 Between comments #1 and #2 add algorithm specific initialization
        state->bits_processed = 0;
        state->unprocessed_bits = 0;
        memcpy(hashState256(state)->DoublePipe, i256p2,  16 * sizeof(u_int32_t));
        // #2 Between comments #1 and #2 add algorithm specific initialization
        return(SUCCESS);

        case 384:
        state->hashbitlen = 384;
        // #1 Between comments #1 and #2 add algorithm specific initialization
        state->bits_processed = 0;
        state->unprocessed_bits = 0;
        memcpy(hashState384(state)->DoublePipe, i384p2,  16 * sizeof(u_int64_t));
        // #2 Between comments #1 and #2 add algorithm specific initialization
        return(SUCCESS);

        case 512:
        state->hashbitlen = 512;
        // #1 Between comments #1 and #2 add algorithm specific initialization
        state->bits_processed = 0;
        state->unprocessed_bits = 0;
        memcpy(hashState224(state)->DoublePipe, i512p2,  16 * sizeof(u_int64_t));
        // #2 Between comments #1 and #2 add algorithm specific initialization
        return(SUCCESS);

        default:    return(BAD_HASHLEN);
    }
}


HashReturn UpdateGPU(hashState *state, const BitSequence *data, DataLength databitlen)
{
    u_int32_t *data32, *p256;
    u_int32_t XL32, XH32, TempEven32, TempOdd32;

    u_int64_t *data64, *p512;
    u_int64_t XL64, XH64, TempEven64, TempOdd64;
    u_int64_t p512_00, p512_01, p512_02, p512_03, p512_04, p512_05, p512_06, p512_07;
    u_int64_t p512_08, p512_09, p512_10, p512_11, p512_12, p512_13, p512_14, p512_15;
    u_int64_t p512_16, p512_17, p512_18, p512_19, p512_20, p512_21, p512_22, p512_23;
    u_int64_t p512_24, p512_25, p512_26, p512_27, p512_28, p512_29, p512_30, p512_31;

    int LastBytes;

    switch(state->hashbitlen)
    {
        case 224:
        case 256:
            if (state->unprocessed_bits > 0)
            {
                if ( state->unprocessed_bits + databitlen > BlueMidnightWish256_BLOCK_SIZE * 8)
                {
                    return BAD_CONSECUTIVE_CALL_TO_UPDATE;
                }
                else
                {
                    LastBytes = (int)databitlen >> 3; // LastBytes = databitlen / 8
                    memcpy(hashState256(state)->LastPart + (state->unprocessed_bits >> 3), data,
LastBytes );
                    state->unprocessed_bits += (int)databitlen;
                    databitlen = state->unprocessed_bits;
```

```
                    data32 = (u_int32_t *)hashState256(state)->LastPart;
                }
            }
            else
                data32 = (u_int32_t *)data;

            p256   = hashState256(state)->DoublePipe;
            while (databitlen >= BlueMidnightWish256_BLOCK_SIZE * 8)
            {
                databitlen -= BlueMidnightWish256_BLOCK_SIZE * 8;
                // #1 Between comments #1 and #2 add algorithm specifics

                state->bits_processed += BlueMidnightWish256_BLOCK_SIZE * 8;
                Compression256();
                data32 += 16;
            }
            state->unprocessed_bits = (int)databitlen;
            if (databitlen > 0)
            {
                LastBytes = ((~(((- (int)databitlen)>>3) & 0x01ff)) + 1) & 0x01ff;  // LastBytes =
Ceil(databitlen / 8)
                memcpy(hashState256(state)->LastPart, data32, LastBytes );
            }
            // #2 Between comments #1 and #2 add algorithm specifics
            return(SUCCESS);


        case 384:
        case 512:
            if (state->unprocessed_bits > 0)
            {
                if ( state->unprocessed_bits + databitlen > BlueMidnightWish512_BLOCK_SIZE * 8)
                {
                    return BAD_CONSECUTIVE_CALL_TO_UPDATE;
                }
                else
                {
                    LastBytes = (int)databitlen >> 3; // LastBytes = databitlen / 8
                    memcpy(hashState512(state)->LastPart + (state->unprocessed_bits >> 3), data,
LastBytes );
                    state->unprocessed_bits += (int)databitlen;
                    databitlen = state->unprocessed_bits;
                    data64 = (u_int64_t *)hashState512(state)->LastPart;
                }
            }
            else
                data64 = (u_int64_t *)data;

            p512   = hashState512(state)->DoublePipe;
            while (databitlen >= BlueMidnightWish512_BLOCK_SIZE * 8)
            {
                databitlen -= BlueMidnightWish512_BLOCK_SIZE * 8;
                // #1 Between comments #1 and #2 add algorithm specifics

                state->bits_processed += BlueMidnightWish512_BLOCK_SIZE * 8;
                Compression512();
                data64 += 16;
            }
            state->unprocessed_bits = (int)databitlen;
            if (databitlen > 0)
            {
                LastBytes = ((~(((- (int)databitlen)>>3) & 0x03ff)) + 1) & 0x03ff; // LastBytes =
Ceil(databitlen / 8)
                memcpy(hashState512(state)->LastPart, data64, LastBytes );
            }
```

```
            // #2 Between comments #1 and #2 add algorithm specifics
            return(SUCCESS);

        default:    return(BAD_HASHLEN); //This should never happen
    }
}


HashReturn FinalGPU(hashState *state, BitSequence *hashval)
{
    u_int32_t *data32, *p256;
    u_int32_t XL32, XH32, TempEven32, TempOdd32;

    u_int64_t *data64, *p512;
    u_int64_t XL64, XH64, TempEven64, TempOdd64;
    u_int64_t p512_00, p512_01, p512_02, p512_03, p512_04, p512_05, p512_06, p512_07;
    u_int64_t p512_08, p512_09, p512_10, p512_11, p512_12, p512_13, p512_14, p512_15;
    u_int64_t p512_16, p512_17, p512_18, p512_19, p512_20, p512_21, p512_22, p512_23;
    u_int64_t p512_24, p512_25, p512_26, p512_27, p512_28, p512_29, p512_30, p512_31;

    DataLength databitlen;

    int LastByte, PadOnePosition;

    switch(state->hashbitlen)
    {
        case 224:
        case 256:
            LastByte = (int)state->unprocessed_bits >> 3;
            PadOnePosition = 7 - (state->unprocessed_bits & 0x07);
            hashState256(state)->LastPart[LastByte] = hashState256(state)->LastPart[LastByte] &
(0xff << (PadOnePosition + 1) )\
                                                    ^ (0x01 << PadOnePosition);
            data64 = (u_int64_t *)hashState256(state)->LastPart;

            if (state->unprocessed_bits < 448)
            {
                memset( (hashState256(state)->LastPart) + LastByte + 1, 0x00,
BlueMidnightWish256_BLOCK_SIZE - LastByte - 9 );
                databitlen = BlueMidnightWish256_BLOCK_SIZE * 8;
                data64[7] = state->bits_processed + state->unprocessed_bits;
            }
            else
            {
                memset( (hashState256(state)->LastPart) + LastByte + 1, 0x00,
BlueMidnightWish256_BLOCK_SIZE * 2 - LastByte - 9 );
                databitlen = BlueMidnightWish256_BLOCK_SIZE * 16;
                data64[15] = state->bits_processed + state->unprocessed_bits;
            }

            data32   = (u_int32_t *)hashState256(state)->LastPart;
            p256     = hashState256(state)->DoublePipe;
            while (databitlen >= BlueMidnightWish256_BLOCK_SIZE * 8)
            {
                databitlen -= BlueMidnightWish256_BLOCK_SIZE * 8;
                // #1 Between comments #1 and #2 add algorithm specifics
                Compression256();
                data32 += 16;
            }
            // #2 Between comments #1 and #2 add algorithm specifics
            break;


        case 384:
        case 512:
```

```
            LastByte = (int)state->unprocessed_bits >> 3;
            PadOnePosition = 7 - (state->unprocessed_bits & 0x07);
            hashState512(state)->LastPart[LastByte] = hashState512(state)->LastPart[LastByte] &
(0xff << (PadOnePosition + 1) )\
                                                    ^ (0x01 << PadOnePosition);
            data64 = (u_int64_t *)hashState512(state)->LastPart;

            if (state->unprocessed_bits < 960)
            {
                memset( (hashState512(state)->LastPart) + LastByte + 1, 0x00,
BlueMidnightWish512_BLOCK_SIZE - LastByte - 9 );
                databitlen = BlueMidnightWish512_BLOCK_SIZE * 8;
                data64[15] = state->bits_processed + state->unprocessed_bits;
            }
            else
            {
                memset( (hashState512(state)->LastPart) + LastByte + 1, 0x00,
BlueMidnightWish512_BLOCK_SIZE * 2 - LastByte - 9 );
                databitlen = BlueMidnightWish512_BLOCK_SIZE * 16;
                data64[31] = state->bits_processed + state->unprocessed_bits;
            }

            p512   = hashState512(state)->DoublePipe;
            while (databitlen >= BlueMidnightWish512_BLOCK_SIZE * 8)
            {
                databitlen -= BlueMidnightWish512_BLOCK_SIZE * 8;
                // #1 Between comments #1 and #2 add algorithm specifics
                Compression512();
                data64 += 16;
            }
            break;
            // #2 Between comments #1 and #2 add algorithm specifics

        default:    return(BAD_HASHLEN); //This should never happen
    }


    switch(state->hashbitlen)
    {
        case 224:
            memcpy(hashval, p256 + 9, BlueMidnightWish224_DIGEST_SIZE );
            return(SUCCESS);
        case 256:
            memcpy(hashval, p256 + 8, BlueMidnightWish256_DIGEST_SIZE );
            return(SUCCESS);
        case 384:
            memcpy(hashval, p512 + 10, BlueMidnightWish384_DIGEST_SIZE );
            return(SUCCESS);
        case 512:
            memcpy(hashval, p512 + 8,  BlueMidnightWish512_DIGEST_SIZE );
            return(SUCCESS);
        default:    return(BAD_HASHLEN); //This should never happen
    }
}

HashReturn HashGPU(int hashbitlen, const BitSequence *data, DataLength databitlen, BitSequence
*hashval)
{
    HashReturn qq;
    hashState state;

    AllocCUDA();

    qq = InitGPU(&state, hashbitlen);
    if (qq != SUCCESS) return(qq);
```

```
    qq = UpdateGPU(&state, data, databitlen);
    if (qq != SUCCESS) return(qq);
    qq = FinalGPU(&state, hashval);

    FreeCUDA();

    return(qq);
}
```

11.6. TurboSHA-2 Implementation

11.6.1. TurboSHA2.h

```
#ifndef TURBOSHA2_H__
#define TURBOSHA2_H__

#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>

/* 32 bit Turbo SHA-2 (224/256) */
uint32_t* TurboSHA224(uint8_t *M, uint64_t l);
uint32_t* TurboSHA256(uint8_t *M, uint64_t l);
/* 64 bit Turbo SHA-2 (384/512) */
uint64_t* TurboSHA384(uint64_t **M, uint128_t N);
uint64_t* TurboSHA512(uint64_t **M, uint128_t N);

void test(uint64_t l);

#endif
```

11.6.2. TurboSHA2.c

```
#include "TurboSHA2.h"

#define Ch(x,y,z)     ((x) & (y)) ^ (~(x) & (z))
#define Maj(x,y,z)    ((x) & (y)) ^ (( x) & (z)) ^ ((y) & (z))
#define SHR(x,n)          ((x) >> (n))
/* 32-bit specific functions */
#define ROTR32(x,n)    (((x) >> (n)) | ((x) << (32 - (n))))
#define SUM32_0(x)     ROTR32(x,2)  ^ ROTR32(x,13) ^ ROTR32(x,22)
#define SUM32_1(x)     ROTR32(x,6)  ^ ROTR32(x,11) ^ ROTR32(x,25)
#define VAR32_0(x)     ROTR32(x,7)  ^ ROTR32(x,18) ^ SHR(x,3)
#define VAR32_1(x)     ROTR32(x,17) ^ ROTR32(x,19) ^ SHR(x,10)
/* 64-bit specific functions */
#define ROTR64(x,n)    ((x) >> (n)) | ((x) << (64 - (n)))
#define SUM64_0(x)     ROTR64(x,28) ^ ROTR64(x,34) ^ ROTR64(x,39)
#define SUM64_1(x)     ROTR64(x,14) ^ ROTR64(x,18) ^ ROTR64(x,41)
#define VAR64_0(x)     ROTR64(x,1)  ^ ROTR64(x,8)  ^ SHR(x,7)
#define VAR64_1(x)     ROTR64(x,19) ^ ROTR64(x,61) ^ SHR(x,6)

/* 32/64 bit functions */
/* Message expansion */
#define MSG_EXP(t,W,P)      W[t-16]                       \
                        + VAR32_0(W[t-15])  \
                        + W[t-14]                     \
                        + VAR32_1(W[t-13])  \
                        + W[t-12]                     \
                        + VAR32_0(W[t-11])  \
                        + W[t-10]                     \
                        + VAR32_1(W[t-9])             \
                        + W[t-8]                      \
                        + W[t-7]                      \
                        + VAR32_0(W[t-6])             \
                        + W[t-5]                      \
                        + VAR32_1(W[t-4])             \
                        + W[t-3]                      \
                        + VAR32_1(W[t-2])             \
                        + VAR32_0(W[t-1])             \
                        + P[t-16]

/* Initialize eight working variables: a,b,c,d,e,f,g,h */
#define AH_INIT(v,H,W) v[0] = H[0] + W[31];           \
                        v[1] = H[1] + W[30];          \
                        v[2] = H[2] + W[29];          \
                        v[3] = H[3] + W[28];          \
                        v[4] = H[4] + W[27];          \
                        v[5] = H[5] + W[26];          \
                        v[6] = H[6] + W[25];          \
                        v[7] = H[7] + W[24];

#define PROC(t,v,T,W) T[0] = v[7]                                   \
                        + SUM32_1(v[4])                       \
                        + Ch(v[4],v[5],v[6])          \
                        + (W[t]   ^ W[t+16])          \
                        + (W[t+4] ^ W[t+24])          \
                        + (W[t+8] ^ W[t+20])          \
                        + W[t+12];                            \
                        T[1] =        SUM32_0(v[0])   \
                        + Maj(v[0],v[1],v[2]);                \
                        v[7] = v[6];                          \
                        v[6] = v[5];                          \
                        v[5] = v[4];                          \
                        v[4] = v[3] + T[0];           \
```

```
                        v[3] = v[2];                        \
                        v[2] = v[1];                        \
                        v[1] = v[0];                        \
                        v[0] = T[0] + T[1];

/* Calculate i-th iteration hash value */
#define iTH_HASH(v,H) H[0] = v[0] + H[0];          \
                        H[1] = v[1] + H[1];          \
                        H[2] = v[2] + H[2];          \
                        H[3] = v[3] + H[3];          \
                        H[4] = v[4] + H[4];          \
                        H[5] = v[5] + H[5];          \
                        H[6] = v[6] + H[6];          \
                        H[7] = v[7] + H[7];

/* Initial Hash values */
/* Turbo SHA-224 */
uint32_t H_224[] = { 0xc1059ed8,
                     0x367cd507,
                     0x3070dd17,
                     0xf70e5939,
                     0xffc00b31,
                     0x68581511,
                     0x64f98fa7,
                     0xbefa4fa4
};

/* Turbo SHA-256 */
uint32_t H_256[] = { 0x6a09e667,
                     0xbb67ae85,
                     0x3c6ef372,
                     0xa54ff53a,
                     0x510e527f,
                     0x9b05688c,
                     0x1f83d9ab,
                     0x5be0cd19
};

/* Turbo SHA-384 */
uint64_t H_384[] = { UINT64_C(0xcbbb9d5dc1059ed8),
                     UINT64_C(0x629a292a367cd507),
                     UINT64_C(0x9159015a3070dd17),
                     UINT64_C(0x152fecd8f70e5939),
                     UINT64_C(0x67332667ffc00b31),
                     UINT64_C(0x8eb44a8768581511),
                     UINT64_C(0xdb0c2e0d64f98fa7),
                     UINT64_C(0x47b5481dbefa4fa4)
};

uint64_t H_512[] = { UINT64_C(0x6a09e667f3bcc908),
                     UINT64_C(0xbb67ae8584caa73b),
                     UINT64_C(0x3c6ef372fe94f82b),
                     UINT64_C(0xa54ff53a5f1d36f1),
                     UINT64_C(0x510e527fade682d1),
                     UINT64_C(0x9b05688c2b3e6c1f),
                     UINT64_C(0x1f83d9abfb41bd6b),
                     UINT64_C(0x5be0cd19137e2179)
};

/* Initial double pipe */
/* Turbo SHA-224/256 */
uint32_t P_256[] = { 0x428a2f98,
                     0x71374491,
                     0xb5c0fbcf,
                     0xe9b5dba5,
```

```
                    0x3956c25b,
                    0x59f111f1,
                    0x923f82a4,
                    0xab1c5ed5,
                    0xd807aa98,
                    0x12835b01,
                    0x243185be,
                    0x550c7dc3,
                    0x72be5d74,
                    0x80deb1fe,
                    0x9bdc06a7,
                    0xc19bf174
};

/* Turbo SHA-384/512 */
uint64_t P_512[] = { UINT64_C(0x428a2f98d728ae22),
                     UINT64_C(0x7137449123ef65cd),
                     UINT64_C(0xb5c0fbcfec4d3b2f),
                     UINT64_C(0xe9b5dba58189dbbc),
                     UINT64_C(0x3956c25bf348b538),
                     UINT64_C(0x59f111f1b605d019),
                     UINT64_C(0x923f82a4af194f9b),
                     UINT64_C(0xab1c5ed5da6d8118),
                     UINT64_C(0xd807aa98a3030242),
                     UINT64_C(0x12835b0145706fbe),
                     UINT64_C(0x243185be4ee4b28c),
                     UINT64_C(0x550c7dc3d5ffb4e2),
                     UINT64_C(0x72be5d74f27b896f),
                     UINT64_C(0x80deb1fe3b1696b1),
                     UINT64_C(0x9bdc06a725c71235),
                     UINT64_C(0xc19bf174cf692694)
};

void TurboSHA256_iteration(uint32_t *M, uint32_t *H, uint32_t *P) {
    uint t;
    uint32_t W[32];
    uint32_t vars[8];
    uint32_t T[2];

    /* Step 1 */
    for(t=0  ; t<16 ; t++) { W[t] = M[t];            }
    for(t=16 ; t<32 ; t++) { W[t] = MSG_EXP(t,W,P);  }

    /* Step 2 */
    for(t=0  ; t<16 ; t++) { P[t] = W[t] + W[t+16];  }

    /* Step 3 */
    AH_INIT(vars,H,W);

    /* Step 4 */
    for(t=0  ; t<8 ; t++) { PROC(t,vars,T,W);        }

    /* Step 5 */
    iTH_HASH(vars,H);
}

void TurboSHA512_iteration(uint64_t *M, uint64_t *H, uint64_t *P) {
    uint t;
    uint64_t W[32];
    uint64_t vars[8];
    uint64_t T[2];

    /* Step 1 */
    for(t=0  ; t<16 ; t++) { W[t] = M[t];            }
    for(t=16 ; t<32 ; t++) { W[t] = MSG_EXP(t,W,P);  }
```

```
    /* Step 2 */
    for(t=0  ; t<16 ; t++) { P[t] = W[t] + W[t+16];  }

    /* Step 3 */
    AH_INIT(vars,H,W);

    /* Step 4 */
    for(t=0  ; t<8 ; t++) { PROC(t,vars,T,W);        }

    /* Step 5 */
    iTH_HASH(vars,H);
}

uint32_t* TurboSHA224(uint8_t *M, uint64_t l) {
    int32_t a,k,delay;
    uint8_t *M_tail;
    uint32_t *H,*M_current;
    uint32_t P[16];
    uint64_t i,r,l_bits;

    /* Initialize some variables                            */
    l_bits = l*8;

    /* Initialize hash and double pipe parameters           */
    H = (uint32_t *)malloc(8*sizeof(uint32_t));
    for(i=0 ; i<8  ; i++) {
        H[i] = H_224[i];
        P[i] = P_256[i];
    }
    for(i=8 ; i<16 ; i++) {
        P[i] = P_256[i];
    }

    /* Padding the message                                  */
    /* Solve this equation l+1+k = 448 mod 512 (k unknown) */
    a = 0;
    r = (l % 64);
    do { k = 448 - 1 - r*8 + 512*a; a++; } while(k < 0);

    /* Initialize last Message block                        */
    M_tail = (uint8_t *)malloc(64*a*sizeof(uint8_t));
    memset(&M_tail[0],0,64*a);

    delay = 0;
    memcpy(&M_tail[0]+delay,&M[0]+(l-r),r);
    delay += r;
    M_tail[delay] = 0x80;
    delay += 1;
    memset(&M_tail[0]+delay,0,(k-7)/8);
    delay += (k-7)/8;
    memcpy(&M_tail[0]+delay,&l_bits,8);

    for(i=0 ; i<l-r ; i+=64) TurboSHA256_iteration((uint32_t *)&M[0]+i,H,P);
    for(i=0 ; i<a ; i+=64) TurboSHA256_iteration((uint32_t *)&M_tail[0]+i,H,P);

    return H;
}

uint32_t* TurboSHA256(uint8_t *M, uint64_t l) {
    uint i;
    uint32_t *H;
    uint32_t P[16];

    H = (uint32_t *)malloc(8*sizeof(uint32_t));
```

```
    for(i=0 ; i<8  ; i++) {
        H[i] = H_256[i];
        P[i] = P_256[i];
    }
    for(i=8 ; i<16 ; i++) {
        P[i] = P_256[i];
    }
    for(i=0 ; i<l ; i++) TurboSHA256_iteration(M[i],H,P);

    return H;
}

uint64_t* TurboSHA384(uint64_t **M, uint N) {
    uint i;
    uint64_t *H;
    uint64_t P[16];

    H = (uint64_t *)malloc(8*sizeof(uint64_t));

    for(i=0 ; i<8  ; i++) {
        H[i] = H_384[i];
        P[i] = P_512[i];
    }
    for(i=8 ; i<16 ; i++) {
        P[i] = P_512[i];
    }
    for(i=0 ; i<N ; i++) TurboSHA512_iteration(M[i],H,P);

    return H;
}

uint64_t* TurboSHA512(uint64_t **M, uint N) {
    uint i;
    uint64_t *H;
    uint64_t P[16];

    H = (uint64_t *)malloc(8*sizeof(uint64_t));

    for(i=0 ; i<8  ; i++) {
        H[i] = H_512[i];
        P[i] = P_512[i];
    }
    for(i=8 ; i<16 ; i++) {
        P[i] = P_512[i];
    }
    for(i=0 ; i<N ; i++) TurboSHA512_iteration(M[i],H,P);

    return H;
}

void print32_hex(uint32_t *A, int arrayNum) {
    uint i;
    for(i=0 ; i<arrayNum ; i++) printf("%8x ",A[i]);
    printf("\n");
}

void print64_hex(uint64_t *A, int arrayNum) {
    uint i;
    for(i=0 ; i<arrayNum ; i++) printf("%16llx ",A[i]);
    printf("\n");
}

struct timeval gtod_start_time;
void timer_start(void) {
```

```
    gettimeofday(&gtod_start_time,0);
}

uint32_t timer_gettime(void) {
    uint32_t result;
    struct timeval gtod_now_time;

    gettimeofday(&gtod_now_time,0);
    result=(gtod_now_time.tv_usec-gtod_start_time.tv_usec);
    //result+=(gtod_now_time.tv_sec-gtod_start_time.tv_sec)*1000;

    return result;
}

void test(uint64_t l) {
    uint i,j,us;
    uint32_t *H32;
    uint64_t *H64;
    uint8_t *M32,*M64;

    printf("Generating %ix16 (32 & 64 bit) random numbers...",l);
    /* Create Nx16 message block full of random numbers, 32&64 bits */
    M32 = (uint8_t *)malloc(l*sizeof(uint8_t));
    M64 = (uint8_t *)malloc(l*sizeof(uint8_t));
    for(i=0 ; i<l ; i++) {
        srand(rdtsc());
        M32[i] = (uint8_t)(rand() % UINT32_MAX);
        M64[i] = (uint8_t)(rand() % UINT32_MAX);
    }
    printf("done\n");

    timer_start();
    H32 = TurboSHA224(M32,l);
    us = timer_gettime();
    printf("Turbo SHA-224 (%i us)\n",us); print32_hex(H32,8); free(H32);

    timer_start();
    H32 = TurboSHA256(M32,N);
    us = timer_gettime();
    printf("Turbo SHA-256 (%i us)\n",us); print32_hex(H32,8); free(H32);

    timer_start();
    H64 = TurboSHA384(M64,N);
    us = timer_gettime();
    printf("Turbo SHA-384 (%i us)\n",us); print64_hex(H64,8); free(H64);

    timer_start();
    H64 = TurboSHA512(M64,N);
    us = timer_gettime();
    printf("Turbo SHA-512 (%i us)\n",us); print64_hex(H64,8); free(H64);

    free(M32);
    free(M64);
}
```