NTNU – Trondheim
Norwegian University of
Science and Technology

Energy Efficient Task Pool Scheduler in OmpSs

Thomas Bølstad Martinsen

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Lasse Natvig, IDI

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

Problem Statement

This is a continuation of the autumn 2012 project "Towards an energy efficient task pool implementation for OmpSs" by Thomas B. Marthinsen. The project contributes to NTNUs participation in the PRACE project[1]. It involves detailed studies of the OmpSs programming environment and it's scheduling of tasks.

It is a goal to develop at least one alternative scheduling-plugin which is designed to improve the energy efficiency, and evaluate it against existing plugins. Evaluations can use synthetic benchmarks but should also use all OmpSs benchmarks the CARD group has access to, including the Mont Blanc applications. One possibility that should be investigated is online monitoring of the active threads serving the OmpSs task pool. The number of threads used in an OmpSs application are determined before execution, and will have the same configuration until the application is completed. Complex applications may consist of several phases, where the preferred number of threads may vary due to differences in resource contention and scalability. If one is able to identify the different phases in an application it may be possible to apply energy efficient techniques or even modify the thread configuration at runtime. The student is also free to investigate other possibilities within the overall goal.

As execution platforms we can continue to use Intel core i7 multicores; quad core Sandy Bridge and Ivy Bridge, the Sandy Bridge-EP server and maybe also the Vilje supercomputer.

Main supervisor: Professor Lasse Natvig (CARD-group, IDI)
Technical co-supervisor: PhD Jan Christian Meyer (High Perf. Computing Section, NTNU - IT Dept.)

Abstract

The European Mont-Blanc project aims to build future exascale systems using energy efficient low-power devices. Exascale systems built using low-power devices will require a large number of processors to achieve competitive performance against state-of-the-art supercomputers. The project relies on the OmpSs programming model and its runtime system, in order to handle the complexity of such a massively parallel system.

In this study, an alternative scheduling-plugin has been developed to improve the energy efficiency of the OmpSs runtime system. The proposed scheduling policy from the paper *'Process Cruise Control'* has been extended for multi-core systems and integrated into the developed scheduling-plugin. The scheduling-plugin improves the energy efficiency by continuously monitoring the workload, in order to identify situations where it would be beneficial to adjust the frequency through dynamic voltage and frequency scaling.

The solution has been evaluated on Sandy Bridge-EP with 17 OmpSs application kernels. Energy consumption is measured for the processor package through the Running Average Power Limit interface on Sandy Bridge. The results shows that energy savings can reach up to 30% in memory intensive applications, with limited impact on performance.

Sammendrag

Det europeiske Mont-Blanc prosjektet tar sikte på å bygge fremtidige exascale-systemer ved hjelp av energieffektive enheter. Exascale-systemer bygget ved hjelp av energieffektive enheter vil kreve et stort antall prosessorer for å oppnå konkurransedyktig ytelse mot state-of-the-art superdatamaskiner. Prosjektet er avhengig av programmeringsmodellen OmpSs og dets runtime system for å administrere kompleksiteten til et slikt massivt parallelt system.

I dette studiet har det blitt utviklet en alternativ scheduling-plugin for å forbedre energieffektiviteten i runtime systemet til OmpSs. Den foreslåtte planleggingspolitikken fra artikkelen *'Process Cruise Control'* har blitt utvidet for flerkjernesystemer og integrert i den utviklede scheduling-pluginen. Scheduling-pluginen forbedrer energieffektiviteten ved kontinuerlig overvåking av arbeidsmengden for å identifisere situasjoner hvor det vil være gunstig å justere frekvensen gjennom dynamisk spenning og frekvensskalering.

Løsningen har blitt evaluert på Sandy Bridge-EP med 17 OmpSs applikasjonskjerner. Energiforbruket er målt for prosessorpakken gjennom Running Average Power Limit grensesnittet på Sandy Bridge. Resultatene viser at løsningen kan spare opp til 30% energi for minne-intensive applikasjoner, med begrenset innvirkning på ytelsen.

Preface

Acknowledgements

This master thesis is the result of work at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The assignment was given by the Computer Architecture and Design Group (CARD) and carried out during the spring 2013. The author would like to thank supervisor, professor Lasse Natvig at IDI for his support and guidance during the project, PhD Jan Christian Meyer, for attending supervision meetings, reading through the report and giving useful feedback, and PhD Juan Manuel Cebrián for technical assistance when using Sandy Bridge-EP.

Contents

Abstract **iii**

Preface **vii**
Acknowledgements . . . vii

Contents **ix**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Research Questions . . . 2
1.3 Contributions . . . 2
1.4 Report Outline . . . 3

2 Background **5**
2.1 Task-based programming . . . 5
2.2 OpenMP SuperScalar (OmpSs) . . . 5
2.2.1 Nanos++ runtime library . . . 7
2.2.2 Overview of Nanos++ class hierarchy . . . 7
2.2.3 Plugins . . . 8
2.3 Contention for Shared Resources in Multi-core processors . . . 9
2.4 Energy efficiency . . . 10
2.4.1 Metrics . . . 10
2.4.2 Energy efficient techniques . . . 11
2.5 Hardware performance counter . . . 13
2.5.1 Performance Application Programming Interface (PAPI) . . . 13
2.6 Intelligent agent . . . 13
2.7 The Landscape of Parallel Computing: Dwarfs . . . 14
2.8 Governors . . . 16
2.9 Related Work . . . 16
2.9.1 Towards an energy efficient task pool implementation for OmpSs . 16
2.9.2 Evaluating Scalability of Multi-threaded Applications on a Many-core Platform . . . 16

2.9.3 Monitoring of Cache Miss Rates for Accurate Dynamic Voltage and Frequency Scaling . . . 17
2.9.4 Process Cruise Control . . . 18
2.9.5 Green Governors: A Framework for Continuously Adaptive DVFS 19

3 Design of scheduling-plugin 21
3.1 The Intelligent Agent . . . 23
3.1.1 Ideas and decisions behind the intelligent agent . . . 23
3.1.2 Design of the lookup table . . . 28
3.1.3 Implementation of the intelligent agent . . . 32

4 Methodology 33
4.1 Benchmarks . . . 33
4.1.1 Mont Blanc application kernels . . . 33
4.1.2 The Barcelona OpenMP Task Suite . . . 34
4.1.3 Others . . . 35
4.2 Experimental Setup . . . 37
4.2.1 Hardware . . . 37
4.2.2 Software and Libraries . . . 39
4.2.3 Compilation . . . 39
4.3 Experiment Methodology . . . 41
4.3.1 Training the lookup table . . . 41
4.3.2 Experiments . . . 41
4.3.3 Experiment configurations . . . 43
4.3.4 Lookup table . . . 44
4.3.5 Measurement tools . . . 48

5 Results 51
5.1 Results for the metric Operations per Joule . . . 51
5.1.1 Results for 16 threads . . . 52
5.1.2 Results for 12 threads . . . 53
5.2 Results for the metric Energy Delay Product . . . 54
5.2.1 Results for 16 threads . . . 54
5.2.2 Results for 12 threads . . . 55
5.3 Results for the metric Operations per Joule under the constraint that performance cannot suffer by more than 10% . . . 56
5.3.1 Results for 16 threads . . . 56
5.3.2 Results for 12 threads . . . 57
5.4 Results for the metric Energy Delay Product under the constraint that performance cannot suffer by more than 10% . . . 58
5.4.1 Results for 16 threads . . . 58
5.4.2 Results for 12 threads . . . 59

6 Discussion 61
6.1 Computational complexity and accuracy of the developed scheduling-plugin . . . 61

6.2 Analysis of the results from the Histogram application kernel 63
6.3 Comparison with results from related work 67

7 Conclusion and Further Work 71
7.1 Conclusion 71
7.2 Further Work 72
7.2.1 Improve the accuracy of the intelligent agent 72
7.2.2 Test the intelligent agent on a system where the frequency of the cores can be changed separately 72
7.2.3 Determine whether Intel® Turbo Boost Technology is energy efficient 72
7.2.4 Investigate the pros and cons between busy waiting and putting an idle thread to sleep 72
7.2.5 Implementation of an asymmetric aware intelligent agent 72
7.2.6 Standardize benchmark suite for task-based programming 73

References 75

Appendices 79

Source Files - Scheduling-plugin 81

Experiment configurations 95

Training Phase - Application kernels 99

Lookup table 105
Operations per Joule 105
Energy Delay Product 110
Operations per Joule under the constraint that performance cannot suffer by more than 10% 116
Energy Delay Product under the constraint that performance cannot suffer by more than 10% 122

Experiment results 129
Results for the metric Operations per Joule 130
Results for the metric Energy Delay Product 146
Results for the metric Operations per Joule under the constraint that performance cannot suffer by more than 10% 162
Results for the metric Energy Delay Product under the constraint that performance cannot suffer by more than 10% 178

List of Figures

2.1 Dependency Graph for the OmpSs example illustrated in Listing 2.1 6
2.2 Overview of the Nanos++ runtime 7
2.3 Conceptual overview of a multi-core 10
2.4 5 Ps of parallel processing 11
2.5 Overview of an Intelligent Agent 13
2.6 The color of a cell indicates the presence of the computational pattern in an application: red/high; orange/moderate; green/low; blue/rare 14
2.7 Lookup table for determining the approperiate clock frequency based on the current state of the system 18

3.1 Overview of the Intelligent Agent 23
3.2 Relative energy consumption between four different frequencies 25
3.3 Block Diagram of the Intel Xeon Processor E5-2600 Family 26
3.4 The temperature of the processor package depends upon the frequency and the number of active cores 27
3.5 The lookup table models the frequency domain with *instructions per cycle*, *cache misses per cycle* and *the number of active cores* 28
3.6 The agent retrieves the most energy efficient frequency based on the current state of the system 28
3.7 Overview of the procedure for training the lookup table 29
3.8 Matrices from the application kernels listed in Table 3.1. In this example, Operations per Joule is applied as metric for energy efficiency. a) = N-body, b) = Merge Sort and c) = Histogram. 30
3.9 The dominance property enforces that the frequencies of the white bins must be lower or equal to the frequencies of the blue bins 31

4.1 Diagram of caches, cores and hyper-threads on Sandy Bridge-EP 37
4.2 Lookup table for optimizing Operations per Joule for core configuration 16 . 44
4.3 Lookup table for optimizing Energy Delay Product for core configuration 16 . 44
4.4 Lookup table for optimizing Operations per Joule under the constraint that performance cannot suffer by more than 10% compared to the execution at the highest frequency for core configuration 16 45
4.5 Lookup table for Energy Delay Product under the constraint that performance cannot suffer by more than 10% compared to the execution at the highest frequency for core configuration 16 45

4.6 Optimal frequency domain 46
4.7 The agent predicts wrong due to variance in measurements and small bins . . 46
4.8 The agent predicts correct, however the bins are too large to replicate the optimal frequency domain 46
4.9 The agent predicts correctly since the training phase takes the variation in the measurements into account 47
4.10 Power Measurements for different combinations of processors, vectorization technologies and thread configurations 49

6.1 The accuracy of the agent decreases as the number of active cores increases . 62
6.2 The agent is able to identify the parallel and sequential phases in the Histogram application kernel 63
6.3 Resource-related stalls for each application kernel 64
6.4 Overview of performance and energy consumption for the Histogram kernel. The results are only for 3-16 threads since these configurations provide the clearest color map. a) = performance and b) = energy consumption 66
6.5 Results for the metric Energy Delay Product with 10% limit when 16 threads are running 67
6.6 Results from *'Process Cruise Control'* 68
6.7 Results from *'Green Governors: A Framework for Continuously Adaptive DVFS'* . 68

7.1 Overview of the Asymmetric Agent 73

List of Tables

2.1 Scheduling-plugin functions . . . 9
2.2 Data structures accessible from the scheduling-plugin . . . 9
2.3 Definitions of equations for reasoning about energy consumption . . . 10
2.4 Description of CPUfreq Governors . . . 16

3.1 Three application kernels used for training . . . 29

4.1 Similarities and differences between the Sandy Bridge-EP and a node in Vilje 38
4.2 Specification for Intel® Xeon® CPU E5-2670 . . . 38
4.3 Cache information for Intel® Xeon® CPU E5-2670 . . . 38
4.4 Software and libraries used in the research . . . 39
4.5 Compiler flags for software and libraries used in the study . . . 39
4.6 Configure flags for Mercurium . . . 39
4.7 Compiler flags for application kernels . . . 40
4.8 Classification of the computational patterns in the selected kernels . . . 42
4.9 Configuration for the lookup tables . . . 44
4.10 Performance counters used in the research . . . 48
4.11 MSRs used for energy measurement . . . 48

6.1 Computational complexity of the developed scheduling-plugin . . . 61
6.2 The accuracy of the agent for the various metrics . . . 62
6.3 Analysis of the behavior of Histogram and Unstructured 3d stencil . . . 64
6.4 Performance counters used to analyze resource-related stalls . . . 64
6.5 Hardware and energy measurements for the different solutions . . . 67

1 Experiment configurations . . . 95
2 Experiment configurations . . . 96
3 Experiment configurations . . . 97

Chapter 1

Introduction

1.1 Motivation

Energy efficiency is one of the major concerns for design of supercomputers, and it is unanimously recognized that if future exascale systems should be affordable, the power consumption of current petascale systems must be reduced[2]. The Mont-Blanc project[3] is a European project that aims to achieve breakthroughs towards energy efficient designs of new supercomputers. The project is coordinated by the Barcelona Supercomputing Center(BSC)[4], and receives financial support from EU and other European partners. The system architecture relies on energy efficient components found in embedded and mobile devices. The project depends on the OmpSs runtime layer to manage the architectural complexity, in order to provide a simple parallel programming interface.

The aim of this research has been to investigate how one can integrate energy efficient techniques into the OmpSs runtime layer. This study is part of a series of research projects carried out by master students at NTNU to examine and evaluate the OmpSs environment[5][6]. This is the first study in the series that aims to develop a component that can be integrated into the OmpSs runtime layer, to improve the energy efficiency of the system.

In current systems the processor can use a large portion of the total energy consumption. Measurements made in *'Analyzing the Energy Efficiency of a Database Server'*[7] demonstrates that the processor can consume over 50% of the energy in database servers. The energy consumption of the processor varies with the frequency. If the processor must stall for outstanding memory requests, it can be energy efficient to decrease the frequency, since the latency of the main memory will be reduced.

In this study, an intelligent agent has been integrated into a scheduling-plugin which can be used from the OmpSs runtime layer. The agent is responsible for adjusting the frequency based on the processor state, in order to maximize the energy efficiency. This type of scheduling-plugin can have applications in systems with energy constraints[8]. If the system is not capable of utilizing the maximum frequency of all cores at the same time, then the scheduling-plugin can prioritize how the available energy should be distributed. The benefit of implementing this mechanism in a scheduling-plugin is that the runtime layer will be able to take into account the priority of different tasks when adjusting the frequency.

1.2 Research Questions

The following questions guided the research:

1. How can the different phases in an application be identified?
2. Is it possible to adjust the thread configuration at runtime?
3. How can knowledge of the phases in an application be used to apply energy efficient techniques?
4. How should the measurements be carried out for evaluating the developed scheduling-plugin?

The first research question deals with the issue of identifying the different phases in an application. The aim is to make the runtime system aware of underlying phases in tasks, so that energy efficient techniques can be applied based on this knowledge.

The second question addresses whether it is possible to modify the thread configuration at runtime. The aim is to investigate whether the OmpSs environment supports functionality to adjust the current thread configuration.

The third question deals with the problem of how energy efficient techniques should be applied. It is a goal to develop at least one alternative scheduling-plugin, so it must be investigated how energy efficient techniques can be integrated.

The fourth question addresses how the energy efficiency of the scheduling-plugin should be evaluated. The evaluation process will use benchmarks from the Mont Blanc application kernels and The Barcelona OpenMP Task Suite. Measurements obtained from the specialization project [9] indicated that the Distributed Breadth First(DBF) scheduler provided the best overall energy efficiency of the scheduling-plugins that were already implemented in the OmpSs environment. The developed scheduling-plugin will be evaluated against DBF, because it will extend functionality from this plugin.

1.3 Contributions

The main contributions of this work are:

1. A prototype of an alternative scheduling-plugin for the OmpSs environment is developed to improve the energy efficiency of the runtime layer.
2. The proposed energy-aware scheduling policy from the paper *'Process Cruise Control'*[10] is extended for multi-core processors.
3. The scheduling-plugin is trained on four different metrics, in order to find out which metric that provides the best energy efficiency for solutions that use dynamic voltage and frequency scaling.

4. The report highlights potential problems with the current implementation of the OmpSs runtime layer that have been discovered through this study.

1.4 Report Outline

Below is a short summary of the content in the different chapters.

Chapter 2 Background presents the OmpSs environment, metrics and techniques for energy efficiency, the dwarfs of parallel computing, governors, the concept of an intelligent agent, and related work.

Chapter 3 Design of scheduling-plugin describes the ideas and decisions behind the scheduling-plugin and how it has been implemented.

Chapter 4 Methodology covers the application kernels used for benchmarking, the experimental setup, and methodology.

Chapter 5 Results presents how the developed scheduling-plugin affects performance and energy consumption for the different application kernels.

Chapter 6 Discussion interprets the results and compares them with findings from related work.

Chapter 7 Conclusion and Further Work concludes with a summary of the results and discusses opportunities for further work.

Chapter 2

Background

2.1 Task-based programming

Task-based programming is a parallel programming model based on Task-Level Parallelism. There are several reasons why it is advantageous[11][12] to express the parallelism in an application in terms of tasks:

- Correspondence between parallelism and the available resources
- Minimize the overhead of starting and terminating a thread
- Opportunities for runtime optimizations
- Increased programmability through abstraction

The fact that parallelism is expressed in terms of tasks creates an abstraction layer, and enables use of intelligent runtime systems that can perform optimizations. The runtime system can limit the number of active threads to the number of cores, reuse threads, improve load balancing, prefetch, and perform replication management.

2.2 OpenMP SuperScalar (OmpSs)

OpenMP SuperScalar(OmpSs) is a task based programming model under development by Barcelona Supercomputing Center(BSC). Explicit message passing has widely been used for communication and exchange of data between nodes in a cluster. Challenges such as variability in application types and resource availability have called for less structured and more asynchronous execution models. OmpSs addresses this problem by exposing the programmer to a virtual shared memory model, where it creates the illusion of a task based model on a single address space. Directionality clauses are used to specify how data is accessed by each task, so the runtime system can compute dependencies, automatically handle data movements, and create an asynchronous dataflow. Listing 2.1 provides an example of how tasks are expressed, and Figure 2.1 illustrates its associated dependency graph.

Listing 2.1 Example of task-based programming with OmpSs

```
/* header */
#pragma omp task output(a)
void A(int* a);

#pragma omp task output(b)
void B(int* b);

#pragma omp task input(a) inout(b)
void C(int* a, int* b);

#pragma omp task input(b)
void D(int* b);

/* source */
int main(int argc, char *argv[]) {
    int* a = new int[1000];
    int* b = new int[1000];

    A(a);
    B(b);
    C(a, b);
    D(b);
}
```

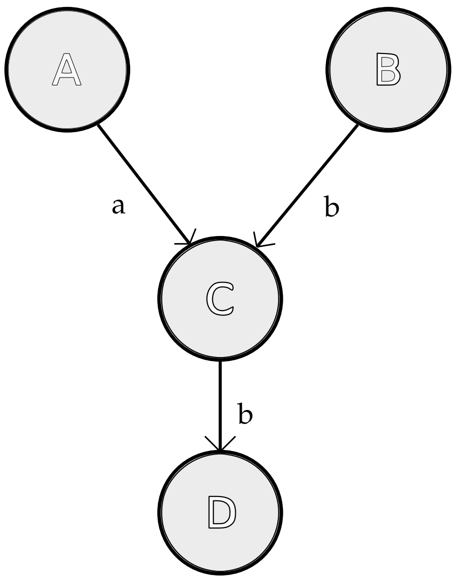


Figure 2.1: Dependency Graph for the OmpSs example illustrated in Listing 2.1

2.2.1 Nanos++ runtime library

Nanos++ is the runtime used by the OmpSs programming model. It is an extensible runtime library designed to support parallel environments. The main purpose of Nanos++ is to be used for research in parallel programming environments, so it is extensible by various forms of plugins. Mercurium is a source-to-source compiler, that transforms specified *#pragmas* to runtime calls.

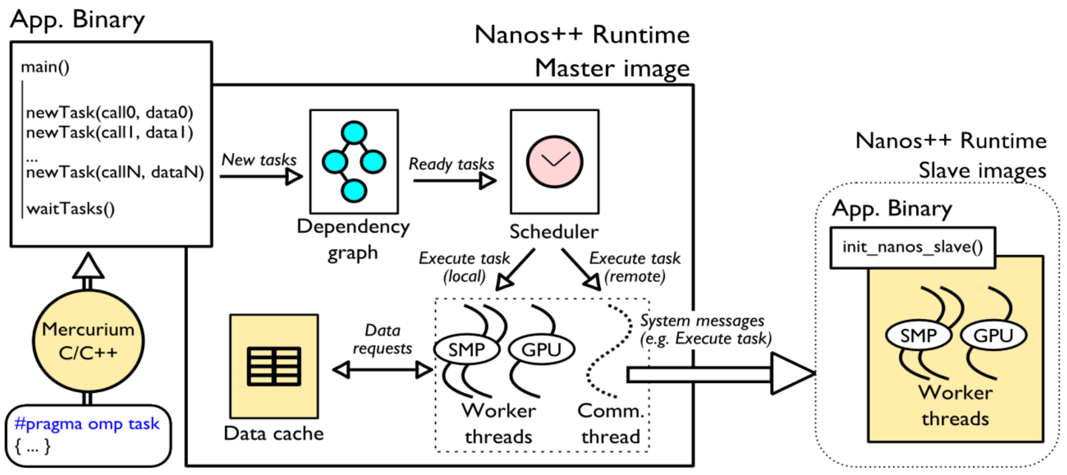


Figure 2.2: Overview of the Nanos++ runtime[13]

2.2.2 Overview of Nanos++ class hierarchy

System

System acts as the interface to the runtime library, and defines the functionality that can be used by parallel programming models. The class is responsible for initializing data structures, and ensures that the system shuts down in a controlled manner.

WorkDescriptor

WorkDescriptor is the class that is responsible for keeping the necessary information and data for a given task. It contains the executable code, dependencies and other properties that are necessary to meet the OpenMP requirements. According to OpenMP 3.0[14], tasks can be specified as either *tied* or *untied*. A tied task can not be stolen after it starts executing on a particular thread due to scheduling restrictions, whereas untied tasks can move freely between threads for load balancing. OpenMP supports tied tasks, since certain functions require that a task is restricted to a single thread in order to make the implementation thread-safe.

DependenciesDomain

DependenciesDomain is a graph node that keeps track of dependencies between tasks, and is part of the internal dependency graph in Nanos++. The DependenciesDomain contains a WorkDescriptor, and prevents it from being submitted to the runtime system until its dependencies are satisfied. When a task is completed, the dependency graph is updated. If all the dependencies for a task are satisfied, it will be submitted to the scheduler.

Schedule

Schedule encapsulates functionality from the scheduling-plugin, and provides basic infrastructure common to every scheduler. A WorkDescriptor which contains the scheduler code is assigned to each thread during the initialization. When a thread completes its current work, it will always return to the scheduler code for further assignments. The scheduler code consists of an infinite loop that calls various functions from the scheduling-plugin, to ensure progression in the dataflow. Nanos 0.6 does not support sleep mode for threads that have been idle for a longer period. This issue is handled in the development of version 0.7.

Processingelement and Accelerator

Processingelement and Accelerator are abstract classes that describe the devices available in the system. They contain functionality for transporting data and initializing the threads that will use the device.

BaseThread

BaseThread is an abstract class that defines functionality for initializing a thread and executing the code contained in a WorkDescriptor. The default thread implementation in Nanos++ 0.7a uses POSIX Threads, however it is possible to support other customizations by inheriting from BaseThread. Each thread contains a data structure which is defined in the scheduling-plugin.

WDDeque

WDDeque is a data structure that can store WorkDescriptors. It is thread-safe by enforcing synchronization when accessed. WorkDescriptors can be added and removed from both sides of the queue.

2.2.3 Plugins

Nanos++ has an extensible design by means of plugins. The plugin is a class that provides an interface to register configurations and code that should be added to the runtime library. Examples of functionality that can be extended through plugins are *Scheduling policies*, *Throttling policies* and *Barrier algorithm*. A plugin should be linked as a shared library after compilation, since Nanos++ makes use of libraries through runtime options.

Scheduling-plugin

The scheduling-plugin defines the policy for how tasks that have satisfied their dependencies should be executed. The policy must determine in which order tasks should be executed, and how they should be distributed between threads. The most important functions found in the scheduling-plugin are *atSubmit* and *atIdle*. The scheduling-plugin

Property	Description
atSubmit	The response of the scheduler when a new task has satisfied its dependencies
atIdle	What should the scheduler do when the current thread has no task

Table 2.1: Scheduling-plugin functions

can assign one of two data structures to each thread, either *ScheduleTeamData* or *ScheduleThreadData*. The data structure can be customized inside the plugin, so there is no limit to what information that can be stored per thread.

Class	Description
ScheduleTeamData	Shared data structure between all threads in the same team
ScheduleThreadData	Per-thread data structure

Table 2.2: Data structures accessible from the scheduling-plugin

Distributed Breadth First scheduler

The Distributed Breadth First scheduling-plugin implements a local WDDeque per thread. Each thread inserts and retrieves tasks from its local queue in a LIFO order (Last In First Out, Stack). If the local queue is empty, the thread will try to execute the parent of its current task. If the parent task can not be executed the thread will try to steal from other queues. The stealing is done in FIFO order (First In First Out).

2.3 Contention for Shared Resources in Multi-core processors

Multi-core processors have several shared devices integrated on the same die. Figure 2.3 shows a conceptual example of how a multi-core can be organized. Both the *last-level cache*, *prefetcher* and the *front-side bus controller* are shared between the two cores. It has become more important to be aware of the underlying hardware when carrying out parallel programming than what was necessary with the uniprocessor. For multi-cores, the programmer must not only make sure that each core performs well on its own, but must also consider how the cores uses the shared resources. If too many cores experience a large number of last-level cache misses, it may indicate that there are conflicts over the cache blocks, and the temporal locality may be reduced as a result. In addition, multiple active cores may increase the need for keeping the shared data in private caches updated, so the interconnection network will waste energy by performing work on behalf of the coherency protocol.

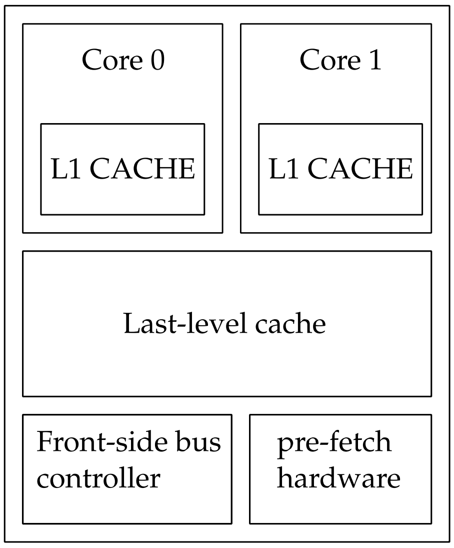


Figure 2.3: Conceptual overview of a multi-core

2.4 Energy efficiency

Power issues are the toughest challenges the computer industry faces in order to maintain a steady growth in performance [15]. Traditionally, time-consumption has been used as the metric to measure the effectiveness of an application. However, since energy consumption has become a major constraint, it is natural that new metrics will emerge. When developing techniques to make a system more energy efficient, the power consumption is often categorized into two groups: *Dynamic power dissipation* and *Static power dissipation*. Dynamic power dissipation is the power used when transistors are turned on and off, while static power dissipation is the leakage current in transistors.

Equation		
$\text{Power}_{\text{dynamic}}$	=	$\text{Frequency} * \text{Voltage}^2 * \text{Capacitance}$
$\text{Power}_{\text{static}}$	=	$\text{Current}_{\text{static}} * \text{Voltage}$
$\text{Power}_{\text{total}}$	=	$\text{Power}_{\text{dynamic}} + \text{Power}_{\text{static}}$

Table 2.3: Definitions of equations for reasoning about energy consumption

2.4.1 Metrics

In computer architecture, energy efficiency refers to the goal of maximizing the ratio $\frac{\text{Performance}^n}{\text{Watt}}$ [15], where n can be selected with regard to how much the metric should emphasize performance over energy consumption. In the article *'Models and Metrics to Enable Energy-Efficiency Optimizations'* [16], Suzanne Rivoire *et al.* present several metrics to measure the energy efficiency of a system. This section presents two essential metrics discussed in the article for measuring the energy efficiency: *Operations per Joule* and *Energy Delay Product*.

Operations per Joule

If performance is measured as $\frac{\text{Operations}}{\text{Second}}$ then $\frac{\text{Performance}}{\text{Watt}}$ can be rewritten as $\frac{\text{Operations}}{\text{Joule}}$, since 1 Watt = 1 $\frac{\text{Joule}}{\text{Second}}$. The metric is suitable for situations where energy consumption is the main concern.

Energy Delay Product

Horowitz *et al.*[17] argued that the use of Operations per Joule as the only metric for measuring energy efficiency may result in designs that favor low performance microprocessors. Reduction of the voltage required to operate the transistors leads to lower energy consumption, but the propagation delay will increase and reduce the frequency. The Energy Delay Product ($\frac{\text{Performance}^2}{\text{Watt}}$) was proposed as the appropriate metric for comparing two processor designs. Consequently, the metric favors architectures that reduce energy consumption and still maintain high performance.

2.4.2 Energy efficient techniques

In the article *'Green Computing: Saving Energy by Throttling, Simplicity and Parallelization'*[18], Lasse Natvig and Alexandru C. Iordan presented an overview of several techniques that are used to make computers more energy efficient. They introduced the concept of the "5 Ps of parallel processing": performance, predictability, power-efficiency, programmability and portability. Figure 2.4 illustrates how the properties of a parallel application may have conflicting goals, so it is important to be aware of the impacts caused by applying energy efficient techniques.

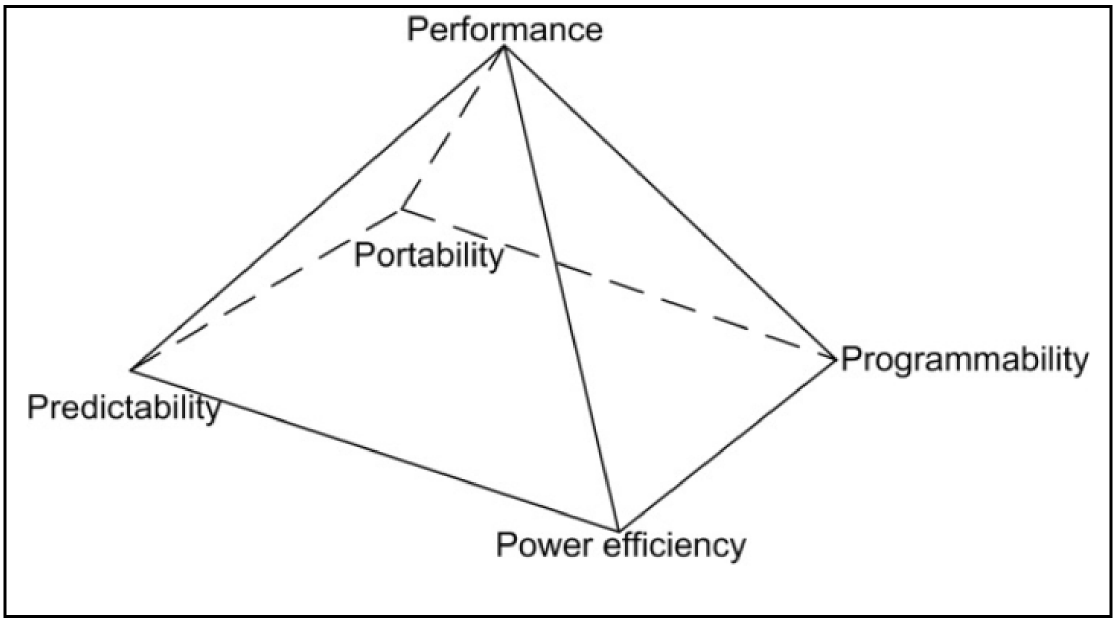


Figure 2.4: 5 Ps of parallel processing[18]

Dynamic power dissipation

Examples of techniques that aim to reduce dynamic power dissipation are: *Dynamic Voltage and Frequency Scaling*, *Sleep modes* and *Overclocking*. The techniques manage power consumption by adaptively adjusting the frequency and voltage in response to changing conditions in workload or environment.

- *Dynamic Voltage and Frequency Scaling (DVFS)*. Computers often experience periods with varying activity where there is no need to operate at the highest frequency. Microprocessors often provide a set of available clock frequencies, and DVFS is a technique that makes it possible to switch between frequencies at runtime.
- *Sleep modes*. In some cases, parts of the system may be disabled during periods to conserve energy. Sleep modes provide opportunities to control which parts of a system that should be enabled, and may have great impact in general computing when the number of devices that can be activated simultaneously is limited by the total energy consumption.
- *Overclocking*. To conserve energy, it may in several cases be an advantage to complete a task quickly so the system can shut down or enable sleep modes. If it is safe to run at a higher clock rate for a short period, it may conserve energy since the computation time decreases and overall static power dissipation may be reduced.

Static power dissipation

Static power dissipation is caused by leakage in transistors, important factors that may reduce the leakage are material properties, operational voltage and the initial design of components.

- *Power gating*. This technique involves turning off the power supply of inactive devices to control loss due to leakage.
- *Near Threshold Computing*. When transistors operate at low voltage the energy demand can be reduced significantly. However the frequency of the processor must be lowered due to increased propagation delay.
- *Simpler designs and Asymmetric Multi-core*. In the article, *'Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era'*[19] it was illustrated that Asymmetric Multi-core is the most energy efficient multi-core design. By focusing on maximizing $\frac{performance}{transistor}$ the overall energy spent on leakage may be reduced. When multiple simpler cores are combined, their static power dissipation will be lower than what state-of-the-art superscalar processors can achieve. However in situations where parallelism is limited, it may turn out that powerful processors are most energy efficient. Therefore, Asymmetric Multi-core systems which consists of many small cores combined with state-of-the-art superscalar processors may be the most energy efficient design.

2.5 Hardware performance counter

Hardware performance counters are a type of control registers that can be accessed in modern microprocessors. These registers can be used to monitor the condition of the processor, as they can be set to count how many times specific events occur during execution. Examples of some events that can be monitored are *cache misses*, *snoop requests*, *mispredicted branches* and *instructions completed*. Current processors support only that a limited number of performance counters can be monitored at the same time.

2.5.1 Performance Application Programming Interface (PAPI)

PAPI is a portable library which provides a consistent interface towards the hardware performance counters in most of the major microprocessors. Both predefined high-level events and more processor-specific events are supported by the library. Each thread in an application can register a private *EventSet* that keeps track of which performance counters that should be monitored. The ability to read, start and stop the performance counters is supported through high-level functions. At context switch, the status of the performance counters are stored just like other private states.

2.6 Intelligent agent

From the field of artificial intelligence in computer science, an intelligent agent refers to anything capable of observing the environment through sensors and acting upon it through actuators. Agents can be categorized into several types based on their tasks. The simplest agents act upon their environment based on current observations where the available actions are determined from static rules, others may incorporate some form of memory to capture previous observations, while more advanced agents may even be able to learn new actuators based on experience and performance feedback.

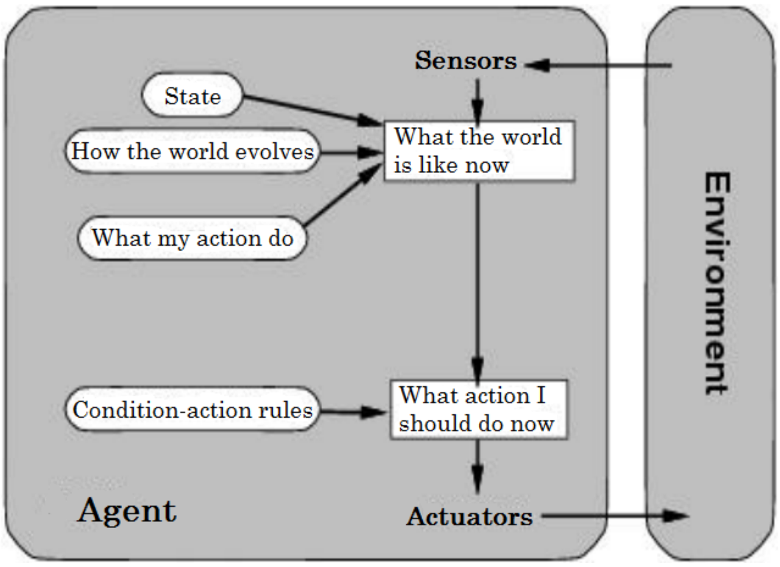
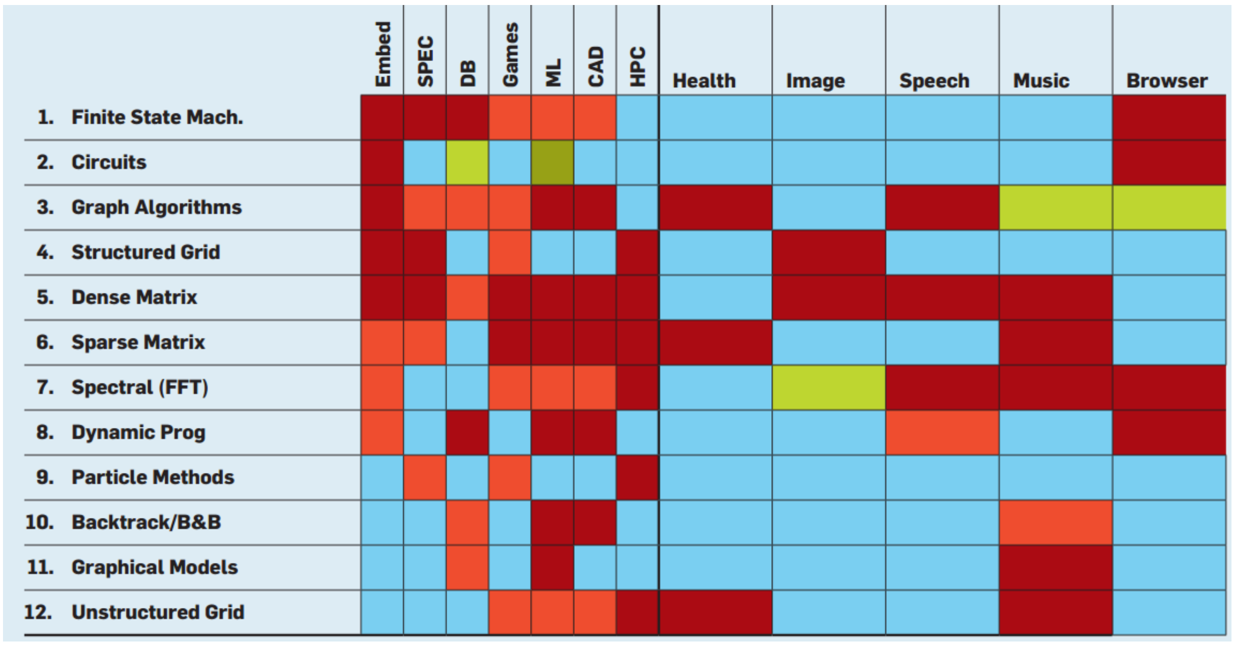


Figure 2.5: Overview of an Intelligent Agent[20]

2.7 The Landscape of Parallel Computing: Dwarfs

The article *'A View of the Parallel Computing Landscape'*[21], provides insight into the challenges that comes with parallel computing. Scientists from **Par Lab** at Berkeley believe that research prototypes should be built on the basis of needs in practical applications. Instead of traditional benchmarks, one should use *dwarfs* identified from real applications when designing and evaluating prototypes. A dwarf is an algorithmic method that captures a pattern of computation and communication. Figure 2.6 provides an overview of 12 dwarfs found in real applications areas.



	Embed	SPEC	DB	Games	ML	CAD	HPC	Health	Image	Speech	Music	Browser
1. Finite State Mach.	🟥	🟥	🟥	🟧	🟧	🟧	🟦	🟦	🟦	🟦	🟦	🟥
2. Circuits	🟥	🟦	🟩	🟦	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟥
3. Graph Algorithms	🟥	🟧	🟧	🟧	🟥	🟥	🟦	🟥	🟦	🟥	🟩	🟩
4. Structured Grid	🟥	🟥	🟦	🟧	🟦	🟦	🟥	🟦	🟥	🟦	🟦	🟦
5. Dense Matrix	🟥	🟥	🟧	🟥	🟥	🟥	🟥	🟦	🟥	🟥	🟥	🟦
6. Sparse Matrix	🟧	🟧	🟦	🟥	🟥	🟥	🟥	🟥	🟦	🟦	🟥	🟦
7. Spectral (FFT)	🟧	🟦	🟦	🟧	🟧	🟧	🟥	🟦	🟩	🟥	🟥	🟥
8. Dynamic Prog	🟧	🟦	🟥	🟦	🟥	🟥	🟦	🟦	🟦	🟧	🟦	🟥
9. Particle Methods	🟦	🟧	🟦	🟧	🟦	🟦	🟥	🟦	🟦	🟦	🟦	🟦
10. Backtrack/B&B	🟦	🟦	🟧	🟦	🟥	🟥	🟦	🟦	🟦	🟦	🟧	🟦
11. Graphical Models	🟦	🟦	🟧	🟦	🟥	🟦	🟦	🟦	🟦	🟦	🟥	🟦
12. Unstructured Grid	🟦	🟦	🟦	🟧	🟧	🟧	🟥	🟥	🟦	🟦	🟥	🟦

Figure 2.6: The color of a cell indicates the presence of the computational pattern in an application: red/high; orange/moderate; green/low; blue/rare. The figure originates from *'A View of the Parallel Computing Landscape'*[21]

Finite State Machines

The computation is represented as a finite state machine that has interconnected states with transitions between one another.

Combinational Logic

Functions which exploit bit-level parallelism to obtain high throughput. This computational pattern is often found in algorithms that performs simple operations on large datasets.

Graph Traversal

Graph algorithms traverse a number of objects and examine them as they are traversed. Characteristic of graph traversal applications is that they often require indirect lookups and little computation.

Structured Grids

The data is arranged in a regular multidimensional grid. Computational steps update all points using data from the neighborhood around each point.

Dense Linear Algebra

Dense Linear Algebra consists of classic vector and matrix operations. The data is typically laid out in continuous arrays.

Sparse Linear Algebra

Matrices have a large number of zero entries, so it becomes advantageous to compress the matrix representation.

Spectral Methods

Spectral Methods consists of spectral domain computations transformed from either temporal or spatial domains. The Fast Fourier Transform algorithm is typically used in this field.

Dynamic Programming

Dynamic Programming is a problem solving paradigm which consists of algorithmic techniques that computes solutions by solving simpler overlapping subproblems. Solutions to previous subproblems are stored to avoid repeating the calculations.

N-Body Methods

N-Body methods involve calculations that depend on ineractions between discrete points or particles. The calculations can often be simplified by using hierarchical methods.

Backtrack and Branch-and-Bound

Branch-and-bound algorithms are effective for solving search and global optimization problems. Dynamic load balance is one of the major challenges in order to gain efficient parallel algorithms.

Graphical Models

Graphical models are graphs that represent random variables as nodes and conditional probabilities as edges. Hidden Markov models and neural networks are examples of graphical models.

Unstructured Grids

The data is arranged in a irregular grid. Computational steps update all points using data from the neighborhood around each point.

MapReduce

The data can be processed independently in parallel, however the results need to be merged.

2.8 Governors

The Linux kernel CPUfreq subsystem provides the ability to control the processor frequency through the use of CPUfreq governors. The governor defines the rules for how the frequency should be adjusted. Table 2.4 gives an overview of the supported governors in CPUfreq 3.8.12-100.fc17. The cpu load is calculated on the basis of how many jiffies the cpu has used to execute processes. A jiffy is the duration between successive system timer interrupts. The value of a jiffy typically varies between 1 ms and 10 ms.

Module	Description
Performance	Run the cpu at the highest frequency
Powersave	Run the cpu at the lowest frequency
Ondemand	Adjusts the frequency between minimum and maximum based on the cpu load
Conservative	The frequency is gradually adjusted base on the cpu load
Userspace	Enables userspace applications to specify the cpu frequency

Table 2.4: Description of CPUfreq Governors

2.9 Related Work

2.9.1 Towards an energy efficient task pool implementation for OmpSs

In the specialization project *'Towards an energy efficient task pool implementation for OmpSs'*[9], Thomas B. Martinsen carried out initial experiments to examine how an energy efficient scheduling-plugin for Nanos++ could be developed. The study evaluated the performance and energy efficiency for the different scheduling policies already implemented in the Nanos++ runtime environment. Experiments revealed that the performance and energy efficiency of the scheduler was mainly influenced by five parameters: search strategy, use of local or global task pool, work-stealing, throttle-policy and the number of active threads. A case-study presented how to develop scheduling-plugins for Nanos++.

2.9.2 Evaluating Scalability of Multi-threaded Applications on a Many-core Platform

In the article *'Evaluating Scalability of Multi-threaded Applications on a Many-core Platform'*[22], Gupta, Kim and Schwan performed a scalability analysis of parallel applications on a 64-threaded Intel Nehalem-EX server. The authors measured how the hardware was used through performance counters, and used the acquired measurements to demonstrate that application performance can be limited due to contention for shared resources. While additional threads are active, the contention for shared resources can increase, and the application may experience stagnation or slowdown in performance. The authors discussed two possibilities to reduce the energy consumption if the application performance is limited due to contention for shared resources. By regulating the number of threads, contention for shared resources can be reduced, and fewer cores need to be enabled. The authors argued that this method could reduce the energy consumption of the executed benchmarks with 59%. If an application must stall due to outstanding memory requests, DVFS can be applied to reduce the power consumption with minimal impact on performance. By applying DVFS, the authors stated that they could reduce the energy consumption of the executed benchmarks with 17%.

2.9.3 Monitoring of Cache Miss Rates for Accurate Dynamic Voltage and Frequency Scaling

In their article, *'Monitoring of Cache Miss Rates for Accurate Dynamic Voltage and Frequency Scaling'*[23] Singleton, Poellabauer and Schwan described an energy-aware scheduler that predicts which frequency consumes the least energy, and still satisfies deadlines in real-time systems. The authors proposed a linear regression model that could predict the execution time of an application based on CPU frequency, memory frequency and cache misses. The linear regression model is presented in Equation 2.1.

$$t_{\text{execution}} = C_{\text{CPU}}(f_{\text{CPU}}) + (\frac{\text{data cache misses}}{\text{instructions executed}} * C_{\text{bus}}(f_{\text{bus}})) \tag{2.1}$$

The equation estimates the execution time where $C_{\text{CPU}}(f_{\text{CPU}})$ is a constant that depends on the CPU frequency, and $C_{\text{bus}}(f_{\text{bus}})$ is a constant that depends on the bus frequency. In order to train the regression model, the authors designed a test program that could be used to generate a specific miss rate by performing memory accesses on a large array. The test program was executed with different miss rates, and the execution time was measured for each run. This procedure was carried out for each available clock frequency. Although the model is trained with a specific test program, the ratio between different $t_{\text{execution}}$ can be used to estimate how DVFS will affect the performance by adjusting the frequency. Once the model is trained, the scheduler can use it at runtime in order to select the most energy-efficient frequency that still meets deadlines.

2.9.4 Process Cruise Control

In the paper *'Process Cruise Control'*[10], Weissel and Bellosa proposed an energy-aware scheduling policy for non-real-time operating systems. The scheduler reads information from performance counters and utilizes it to determine *the approperiate clock frequency* for each running process. The approperiate clock frequency was defined to be the lowest frequency where the performance only suffered with 10% compared to the execution time with the highest clock speed. The authors argued that *instructions per clock cycle*(IPC) and *memory requests per clock cycle*(MRPC) were appropriate events to identify how the hardware is used by a process. It would be desirable to have a function *f(IPC, MRPC)* that returned the appropriate frequency, however, finding an analytical expression for this function may be challenging. Therefore, the authors suggested to represent the parameter space of the function in form of a precomputed lookup table as illustrated in Figure 2.7.

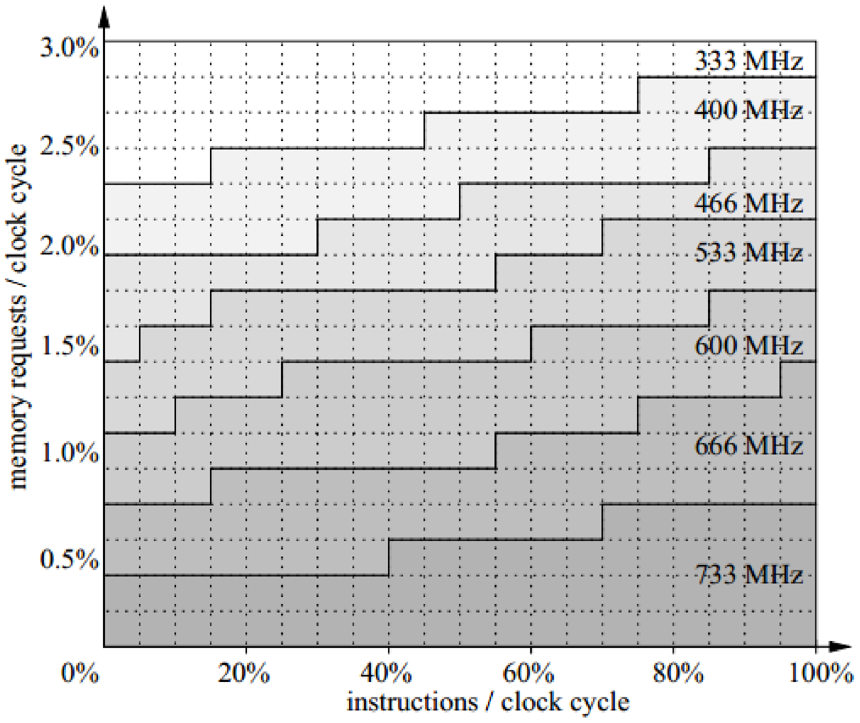


Figure 2.7: Lookup table for determining the approperiate clock frequency based on the current state of the system[10]

The lookup table was trained with six synthetic benchmarks with different characteristics in order to cover the parameter space. The lookup table was integrated into the Linux scheduler, where it was used to predict the optimal frequency for a process at each context switch.

2.9.5 Green Governors: A Framework for Continuously Adaptive DVFS

In the article *'Green Governors: A Framework for Continuously Adaptive DVFS'*[24], Spiliopoulos, Kaxiras and Keramidas developed energy efficient governors that adjusted the frequency based on information from performance counters. The governors are effective for memory intensive applications, since reducing the processor frequency will lower the main memory latency. The governors were evaluated on Intel i7 and AMD Phenom II. The authors used three equations to determine whether the frequency should be adjusted. The equations were evaluated for each possible frequency in order to estimate the minimum Energy Delay Product that could be achieved. Equation 2.2 provides the total energy consumption for an interval based on models trained for estimating the dynamic and static energy used by the system. The governors were configured to be called periodically every 50ms.

$$\text{Energy}_{\text{predicted}} = \text{Dynamic}_{\text{model}}(\text{ipc}) + \text{Static}_{\text{model}}(\text{frequency}, \text{temperature}) * 50ms \quad (2.2)$$

Equation 2.3 calculates the expected execution time that the processor will achieve based on frequency and stalls due to non-overlapping last level cache misses. The model relies on knowledge about the last level cache miss penalty in order to estimate the number of non-overlapping last level cache misses.

$$\text{Time}_{\text{predicted}} = \text{Stall}_{\text{model}}(\text{stalls}, \text{frequency}) \quad (2.3)$$

Equation 2.4 computes the predicted Energy Delay Product.

$$\text{EDP}_{\text{predicted}} = \text{Energy}_{\text{predicted}} * \text{Time}_{\text{predicted}} \quad (2.4)$$

Chapter 3

Design of scheduling-plugin

Several benchmarks were used in the specialization project[9] in order to identify situations where a scheduling policy is more energy efficient than others. The benchmarks were compiled with Mercurium 1.3.5.8, and Nanos++ 0.6a was used as runtime system. The benchmarks were executed on a Sandy Bridge-EP server, which consists of two Intel Xeon CPU E5-2670. Results from the study indicated that there was variation between the scheduling policies, however, it also appeared that the thread configuration had an impact on the energy efficiency. In some cases it was optimal to activate all the cores, while at other times the performance stagnated or even decreased when additional threads were added due to increased resource contention. If the thread configuration can be adjusted at runtime, it may be possible to search for an optimal configuration guided by information from hardware performance counters, as described in *'Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs'* [25] and *'Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring'* [26]. It was investigated whether Nanos++ supported functionality to suspend and resume threads during execution. The underlying thread implementation in Nanos++ 0.7a-2013-02-22 uses POSIX Threads for executing tasks on symmetric multiprocessors (SMP). The native POSIX thread library in Linux does not support functionality to suspend and resume threads, because it may be unsafe to block threads that currently hold locks. The POSIX thread implementation in RTLinux supports **pthread_suspend_np** and **pthread_unsuspend_np**. It may be possible to modify the thread implementation and compiler so Nanos++ can identify whether it is safe to suspend a running thread, however, it will require extensive knowledge about Nanos++, Mercurium and the Linux kernel, which is beyond the scope of this project.

Although it is not possible to adjust the thread configuration at runtime, the energy efficiency can be increased by changing the processor frequency according to the workload. If the speedup begins to stagnate or decrease as more threads are added, it may be efficient to lower the frequency in order to reduce the dynamic power dissipation. However, both the execution time and the static power dissipation may increase as a result. When the frequency is lowered, the processor will spend fewer cycles waiting for outstanding requests, since the memory latency will be reduced. If the performance is limited due to contention for shared resources, the energy saved by reducing the dynamic power dissipation may outweigh the increased static power dissipation.

In this study, the Distributed Breadth First scheduler has been extended with an intelligent agent that observes the state of the system, and applies *dynamic voltage and frequency scaling (DVFS)* in order to make Nanos++ more energy efficient. DVFS is a technique which enables the frequency of the microprocessor to automatically be adjusted at runtime. The *userspace* governor allows userspace applications to set the processor frequency explicitly, however Intel Xeon CPU E5-2670 enforces that each core must have the same frequency. This means that if the frequency of a core is adjusted it will affect all the cores on the same socket.

3.1 The Intelligent Agent

An intelligent agent has been designed to observe the system and apply DVFS if the current frequency is ineffective. The agent reads information from hardware performance counters at regular intervals through the PAPI interface, and uses it to predict the most energy efficient frequency based on the current state of the system. Figure 3.1 provides an overview of how the agent has been designed. The agent uses *Instructions per cycle (IPC)*, *Last level cache misses per cycle (LLCMPC)* and *the number of active cores* as indices into a lookup table that contains the frequency which is estimated to be the most energy efficient for this state.

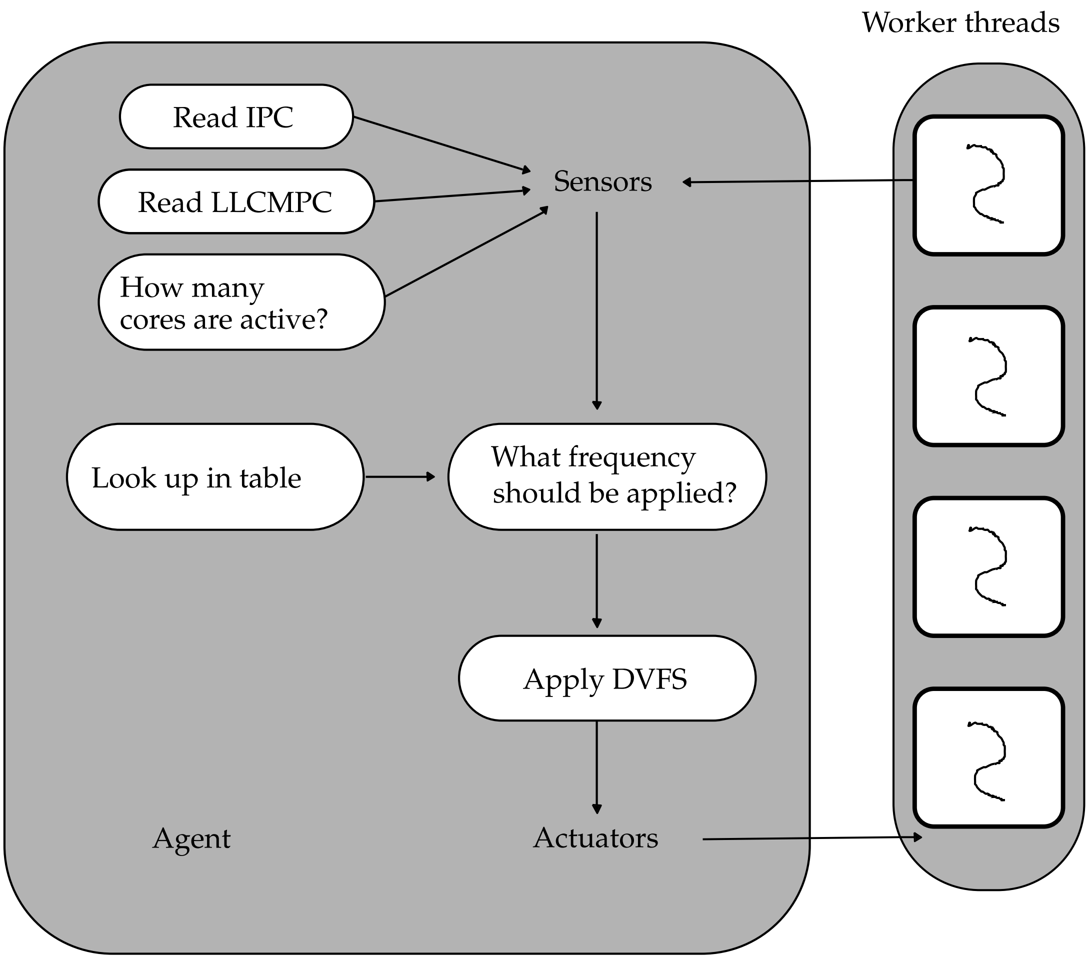


Figure 3.1: Overview of the Intelligent Agent

3.1.1 Ideas and decisions behind the intelligent agent

It was determined that the intelligent agent should be designed to leverage DVFS, since it was not possible to adjust the thread configuration at runtime. The organization of the task environment in Nanos++ limits which algorithms the agent can use to decide if DVFS should be applied or not. The agent is restricted to algorithms that do not rely on a fully observable task graph or knowledge about task durations. This study considers extensions of one of the energy-aware schedulers proposed in *'Monitoring of*

Cache Miss Rates for Accurate Dynamic Voltage and Frequency Scaling'[23], *'Green Governors: A Framework for Continuously Adaptive DVFS'*[24] and *'Process Cruise Control'*[10]. None of these schedulers rely on knowledge about task durations or the task graph.

In the article, *'Monitoring of Cache Miss Rates for Accurate Dynamic Voltage and Frequency Scaling'* it was proposed that the scheduler should determine which frequency consumes the least amount of energy, and still satisfies deadlines in real-time systems. In order to do this the scheduler uses a linear regression model to estimate the relative performance between frequencies. Nanos++ is not a real-time system, and the task durations are unknown. However, the relative performance between frequencies can still be useful to determine which frequency is most energy efficient. If one measures the energy efficiency in terms of the lowest energy consumption, then the original regression model can be extended by multiplying $t_{\text{execution}}$ with $\text{Power}_{\text{frequency}}$.

$$t_{\text{execution}} * \text{Power}_{\text{frequency}} = (C_{\text{CPU}}(f_{\text{CPU}}) + (\frac{\text{data cache misses}}{\text{instructions executed}} * C_{\text{bus}}(f_{\text{bus}}))) * \text{Power}_{\text{frequency}}$$

$\text{Power}_{\text{frequency}}$ can be estimated at runtime with a power model[27] based on performance counters. The agent can select the frequency with the lowest estimated energy consumption when DVFS should be applied. However, when benchmarks were executed in order to collect data, it appeared that the cache miss rate gave a poor estimate of how efficiently the processor could utilize the frequency to execute instructions. The study was originally carried out with an Intel XScale PXA255 evaluation board. From *'Intel® XScale™ Microarchitecture for the PXA255 Processor User's Manual'*[28], it was found that the XScale PXA255 pipeline is scalar and single issue. Although XScale PXA255 has three pipelines, Intel Xeon CPU E5-2670 is a more advanced processor which is able to hide memory latency by out of order execution. Therefore, cache misses on the Intel Xeon CPU E5-2670 may give an inaccurate estimate for which frequency is most energy efficient.

The problem with the algorithm proposed in *'Monitoring of Cache Miss Rates for Accurate Dynamic Voltage and Frequency Scaling'* was that it solely relied on the cache miss rate. As long as the processor does not need to stall for outstanding memory requests, it can continue to do useful work even though the cache miss rate is high. In the paper, *'Process Cruise Control'*, it was proposed that the scheduler should use information about both the IPC and memory requests in order to determine *the appropriate clock frequency*. The advantage of combining IPC and memory requests is that the scheduler gains more accurate insight about the processor state. A high IPC indicates that the processor still can do useful work, even if it experiences a large number of cache misses. The scheduler relies on a lookup table that has been trained to contain frequencies which are optimal in relation to a particular metric. In the original study, the metric was defined to be the lowest frequency where the performance only suffered with 10% compared to the execution time with the highest clock speed, however, the metric may as well be *Operations per Joule* or *Energy Delay Product*. In the article, *'Green Governors: A Framework for Continuously Adaptive DVFS'*, it was suggested to combine the IPC and the last level cache miss penalty in order to predict the most energy efficient frequency. Due to insufficient information about the last level cache miss penalty on Sandy Bridge-EP, it was decided to extend the solution proposed in *'Process Cruise Control'*.

In this study, the idea of using a lookup table to predict the most energy efficient frequency is extended to multi-core processors by adding a new dimension for the number of active cores. The new dimension is able to capture how the frequency is affected by factors as *the ratio between static power dissipation and dynamic power dissipation* and *increased leakage current* as additional cores are enabled. Instead of implementing the lookup table in the Linux scheduler, it is integrated in a scheduling-plugin that can be used in Nanos++. The Intel Sandy Bridge Microarchitecture does not support performance counters to monitor the number of memory requests, therefore, the number of last level cache misses is selected as a replacement, since requests that access main memory have the highest miss penalty.

Figure 3.2 illustrates how the number of enabled cores affects which frequency is most energy efficient. The results show the relative energy consumption between four different frequencies where Distributed Breadth First(DBF) is selected as scheduling-plugin. While additional threads are activated, lower frequencies become more energy efficient. The results are from a synthetic benchmark that only utilizes the private caches, and is configured to generate an identical task for each thread. This suggests that the optimal choice of frequency for energy efficiency is influenced by more factors than contention for shared resources. The measurements are made for the processor package.

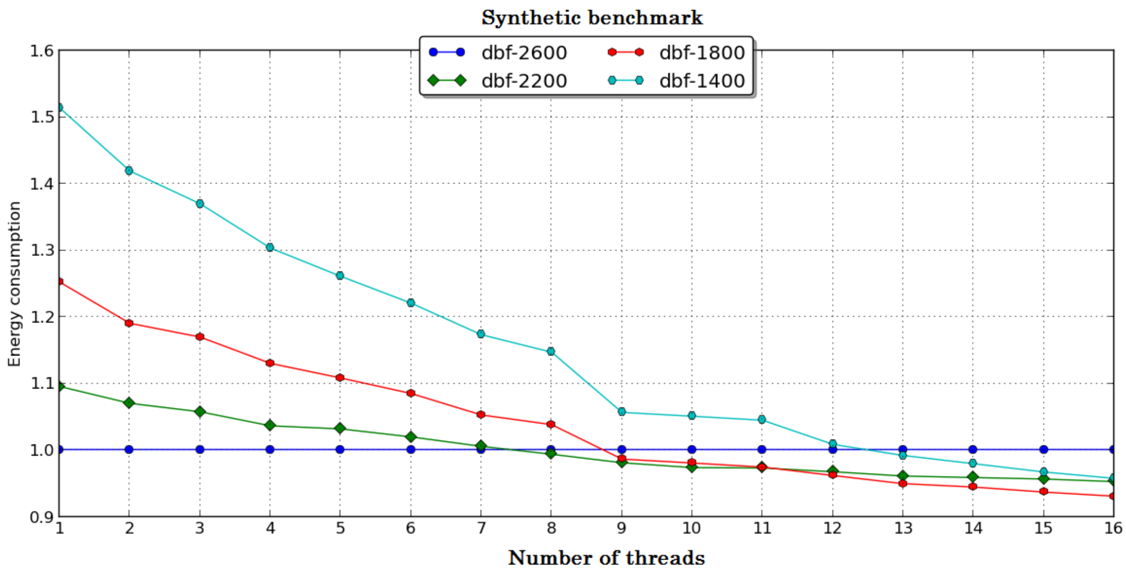


Figure 3.2: Relative energy consumption between four different frequencies

It is assumed that *the ratio between static power dissipation and dynamic power dissipation* and *increased leakage current* affects the frequency that is most energy efficient, since these factors are influenced by the number of enabled cores.

The ratio between static power dissipation and dynamic power dissipation

Intel Xeon CPU E5-2670 contains eight cores and various shared resources. As can be seen from Figure 3.3, the shared resources must be enabled if one of the cores are activated.

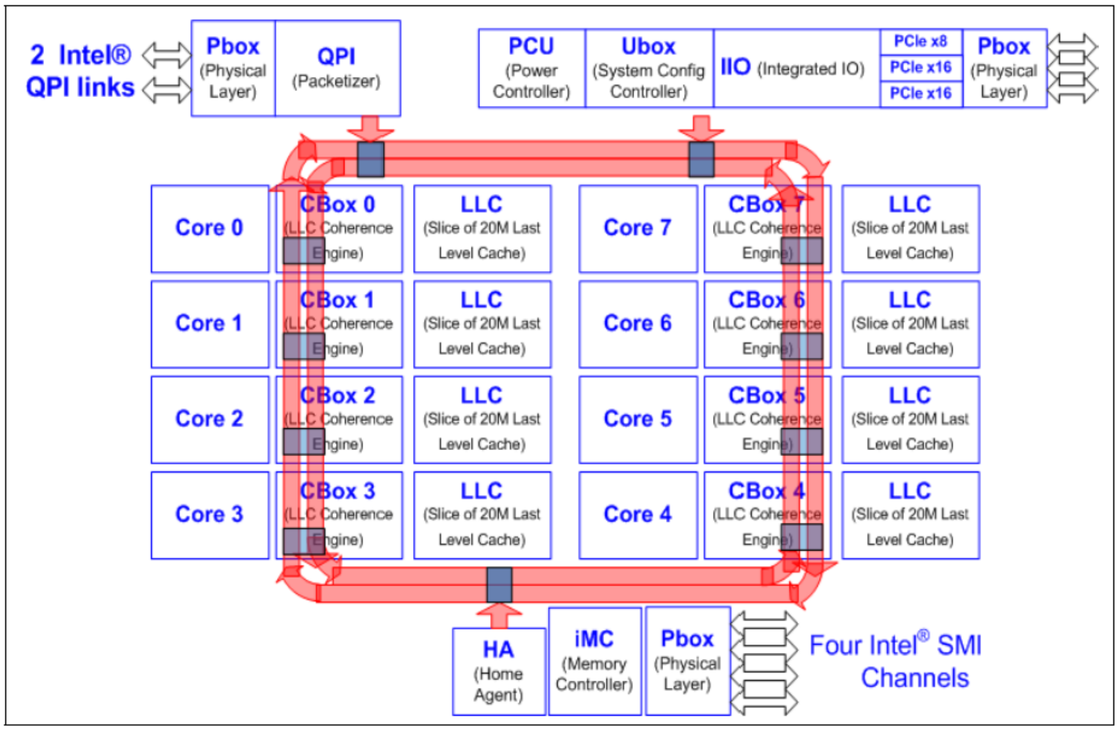

2 Intel® QPI links
Pbox (Physical Layer)
QPI (Packetizer)
PCU (Power Controller)
Ubox (System Config Controller)
IIO (Integrated IO)
PCIe x8
PCIe x16
PCIe x16
Pbox (Physical Layer)
Core 0
Core 1
Core 2
Core 3
CBox 0 (LLC Coherence Engine)
CBox 1 (LLC Coherence Engine)
CBox 2 (LLC Coherence Engine)
CBox 3 (LLC Coherence Engine)
LLC (Slice of 20M Last Level Cache)
Core 7
Core 6
Core 5
Core 4
CBox 7 (LLC Coherence Engine)
CBox 6 (LLC Coherence Engine)
CBox 5 (LLC Coherence Engine)
CBox 4 (LLC Coherence Engine)
HA (Home Agent)
iMC (Memory Controller)
Pbox (Physical Layer)
Four Intel® SMI Channels

Figure 3.3: Block Diagram of the Intel Xeon Processor E5-2600 Family[29]

Equation 3.1 describes the energy consumption for the processor package.

$$\text{Energy}_{\text{package}} = \text{Energy}_{\text{cores}} + \text{Energy}_{\text{shared resources}} \tag{3.1}$$

The goal is to minimize $\text{Energy}_{\text{package}}$, however, minimization of $\text{Energy}_{\text{cores}}$ and $\text{Energy}_{\text{shared resources}}$ represents a conflict. If $\text{Energy}_{\text{cores}}$ is reduced, it leads to increased execution time, which affects $\text{Energy}_{\text{shared resources}}$. When few cores are active, $\text{Energy}_{\text{shared resources}}$ dominates $\text{Energy}_{\text{cores}}$. In this case, a high frequency can be tolerated even if the processor must stall frequently due to outstanding memory requests. Similar ideas are used in *'Intel® Turbo Boost Technology'*[30], which is a technology that dynamically increases the frequency based on the number of active cores, estimated current consumption, estimated power consumption, and the processor temperature.

Increased leakage current

In the article, *'Leakage Current: Moore's Law Meets Static Power'*[31] an equation for calculating the subthreshold leakage current it is presented:

$$I_{sub} = K_1 * W * e^{\frac{-V_{th}}{n*V_{\Theta}}} * (1 - e^{\frac{-V}{V_{\Theta}}}) \tag{3.2}$$

In this equation, I_{sub} is the subthreshold leakage current. K_1 and n are empirically determined, W is the transistor width, while V_{Θ} in the exponents is the thermal voltage which increases linearly with the temperature. The authors argue that if the subthreshold leakage current builds up heat, V_{Θ} will start to rise, further increasing the leakage current and possibly causing thermal runaway.

The temperature of the system increases with the number of cores that are enabled, therefore, the subthreshold power leakage may influence which frequency that is most energy efficient. Lm-sensors (Linux monitoring sensors) provides tools and drivers for monitoring temperatures, voltage, and fans. Lm-sensors was used for measuring temperature on the processor package. Figure 3.4 illustrates how the temperature is affected by the frequency and the number of active cores. It is assumed that the system's temperature increases when two sockets are enabled, however, the exact temperature is not measured since the evaluated system has no off-chip temperature sensors that can be accessed through lm-sensors.

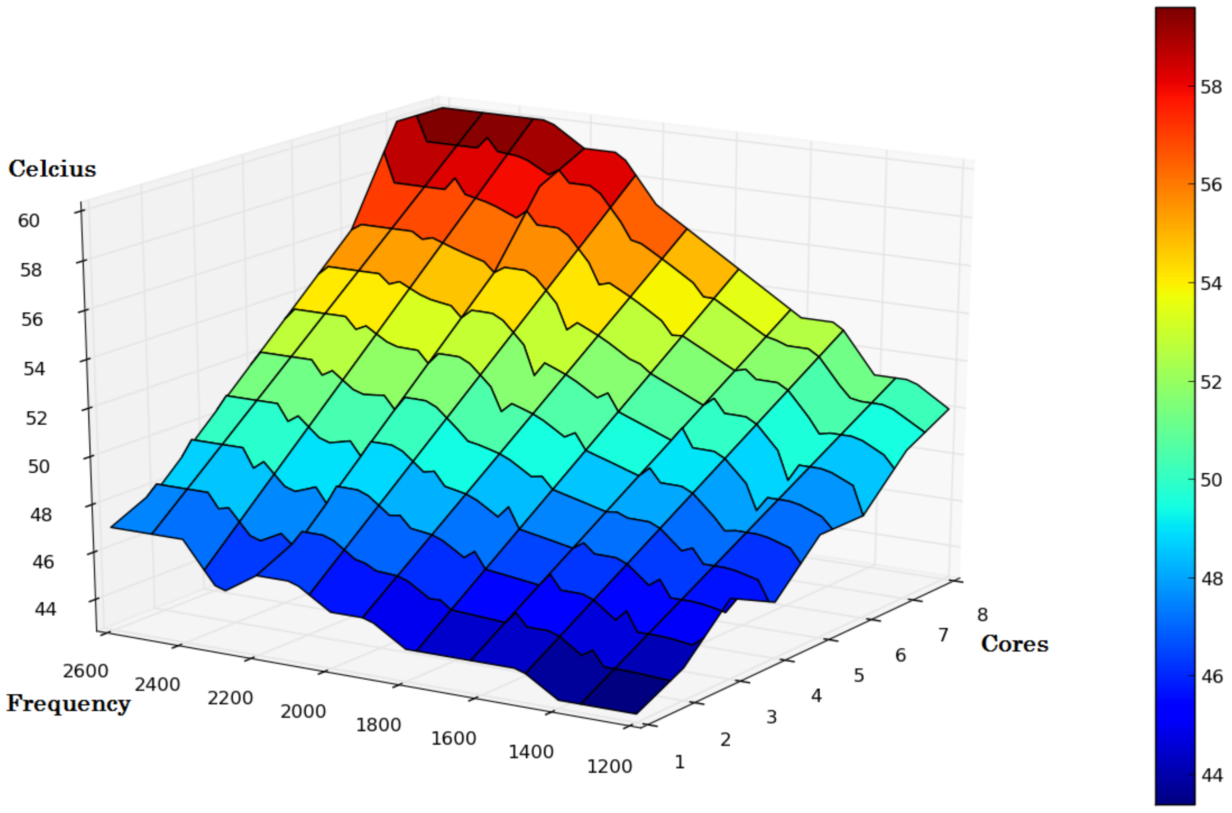


Figure 3.4: The temperature of the processor package depends upon the frequency and the number of active cores

3.1.2 Design of the lookup table

The lookup table from the article *'Process Cruise Control'* has been extended with a new dimension to account for the number of active cores as seen in Figure 3.5.

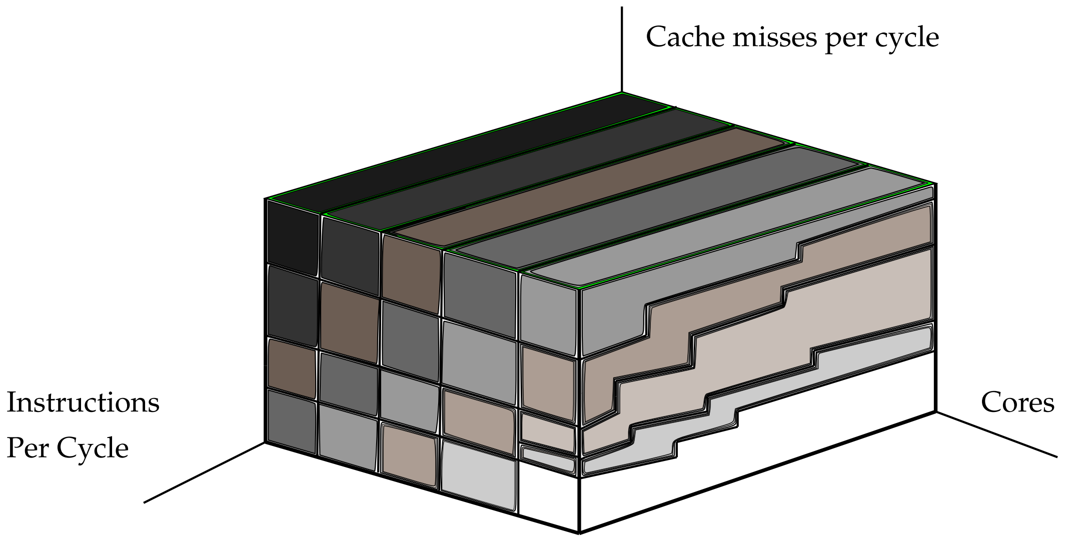


Figure 3.5: The lookup table models the frequency domain with *instructions per cycle*, *cache misses per cycle* and *the number of active cores*

The dimension for the number of active cores is discrete, while the dimensions for IPC and LLCMPC are continuous. Figure 3.6 illustrates how the parameter space is divided into discrete bins. Each bin contains a frequency and covers an area in the parameter space. When the intelligent agent should select a frequency from the lookup table, it will pick the bin that covers the current combination of IPC, LLCMPC and number of active cores.

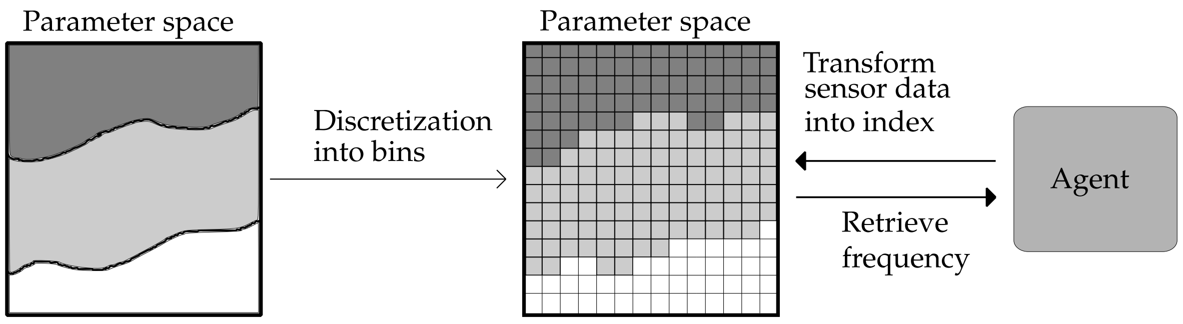


Figure 3.6: The agent retrieves the most energy efficient frequency based on the current state of the system

Training phase

Each bin in the lookup table is initialized with the highest available frequency. The lookup table needs to be trained before it can be used to predict the most energy efficient frequency based on IPC, LLCMPC and the number of active cores. In this study the lookup table has been trained with seven different application kernels described in Section 4.3. The training phase is an offline process, and is illustrated in Figure 3.7.

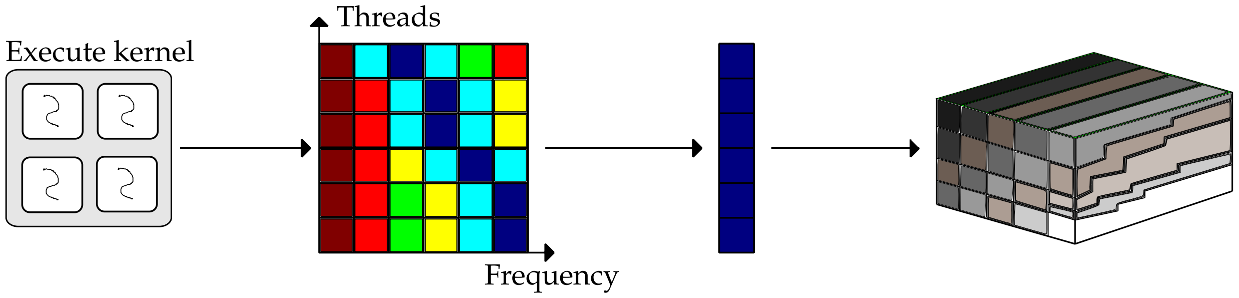


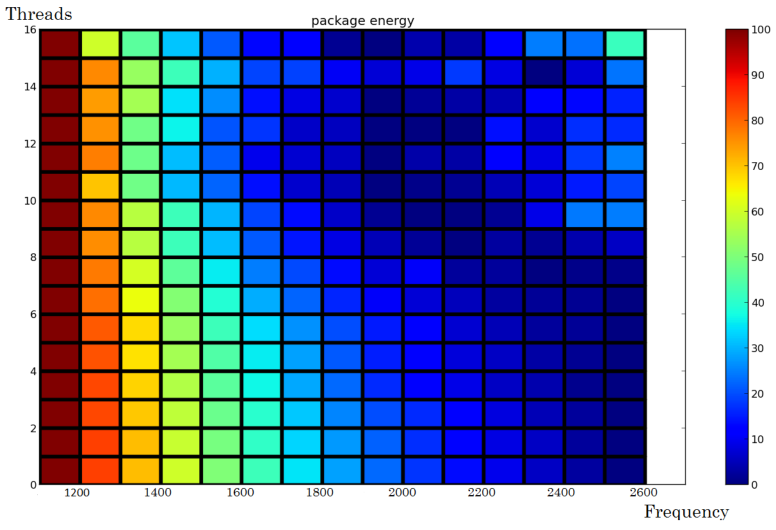
Figure 3.7: Overview of the procedure for training the lookup table

Each kernel is executed with every combination of frequencies and thread configurations available to the system. Energy consumption, IPC and LLCMPC are measured and stored in a temporary matrix. The matrix is indexed with the thread configuration and the frequency that were used during the execution of the kernel. The measured energy consumption can be used to indicate which frequency that is most energy efficient if $\frac{Operations}{Joule}$ is applied as metric, since the number of operations stays constant when only the frequency is adjusted. For each thread configuration in the matrix, the most energy efficient frequency will be selected and stored in a column. When the lookup table is updated with the selected frequencies, the table will be indexed with their corresponding IPC, LLCMPC and thread configuration. Figure 3.8 presents three of the matrices that have been produced in the training phase. The matrices illustrates how IPC and LLCMPC affect which frequency that is most energy efficient. Table 3.1 provides an overview of the selected application kernels. Note that the listed IPC and LLCMPC are calculated with the harmonic mean over all runs.

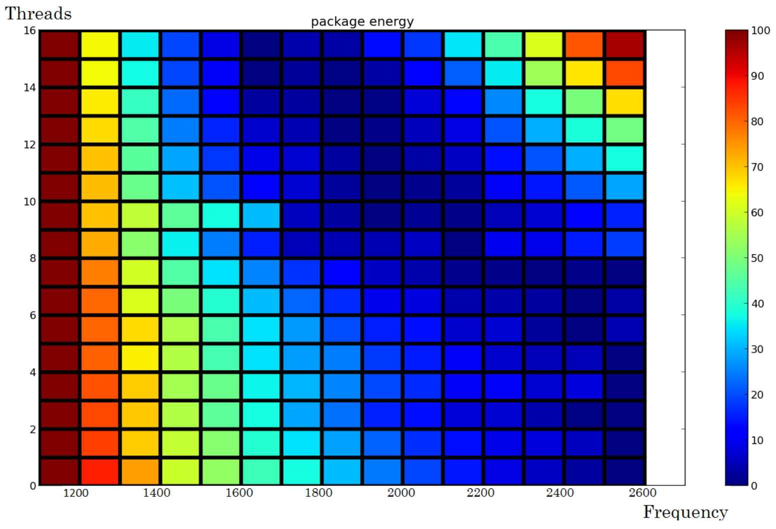
Application kernel	**IPC**	**LLCMPC**
N-body	1.2027	0.0000
Merge Sort	1.0501	0.0002
Histogram	0.5173	0.0064

Table 3.1: Three application kernels used for training

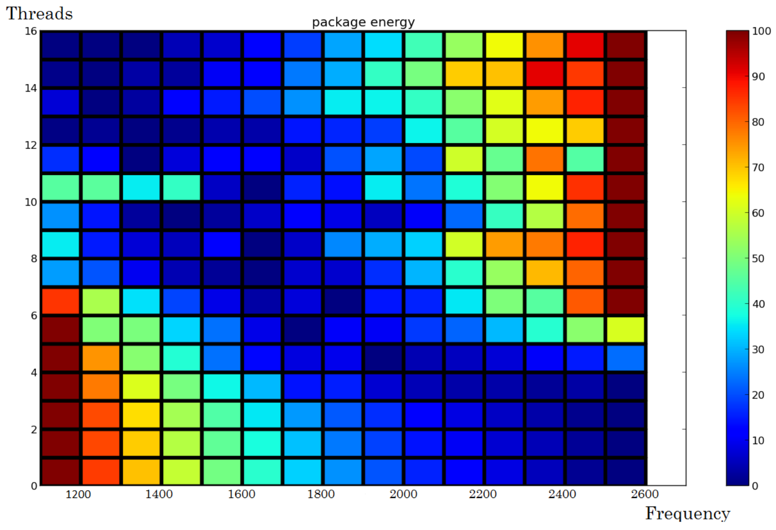
Each row in the matrix represents a thread configuration and each column a frequency. The energy consumption is normalized within each thread configuration to the range 0 - 100, where the frequency with the lowest value is the most energy efficient. The matrices show that the most energy efficient frequency depends on IPC, LLCMPC and the number of active cores. Appendix 7.2.6 provides an overview of all the matrices that have been used to train the lookup table.



(a)



(b)



(c)

Figure 3.8: Matrices from the application kernels listed in Table 3.1. In this example, Operations per Joule is applied as metric for energy efficiency. a) = N-body, b) = Merge Sort and c) = Histogram.

If the lookup table is large, the accuracy for selecting the most energy efficient frequency will increase. However, it may take a long time to fill each bin in the table since it would require execution of ($\mathrm{IPC}_{\mathrm{bins}} * \mathrm{LLCMPC}_{\mathrm{bins}} * \mathrm{Cores} * \mathrm{Frequencies}$) kernels. To simplify how the table is filled, it is assumed that certain bins will be dominated by others. A bin dominates another bin if it is as good or better in all dimensions. In this case, it is good to have high IPC, low LLCMPC and a low number of active cores.

Definition 1. *The definition of the dominance property.*

Bin_A *is dominated by* $Bin_B \leftrightarrow (Bin_A.IPC \leq Bin_B.IPC \wedge Bin_A.LLCMPC \geq Bin_B.LLCMPC \wedge Bin_A.Cores \geq Bin_B.Cores)$

Bin_A *is dominated by* $Bin_B \leftrightarrow (Bin_A.Frequency \leq Bin_B.Frequency)$

The principle of the dominance property is illustrated in Figure 3.9. The parameter space is divided into discrete bins, where the white bins are dominated by the blue bins. The dominance property enforces that the frequencies of the white bins must be lower or equal to the frequencies of the blue bins since their IPC are lower and their LLCMPC are higher.

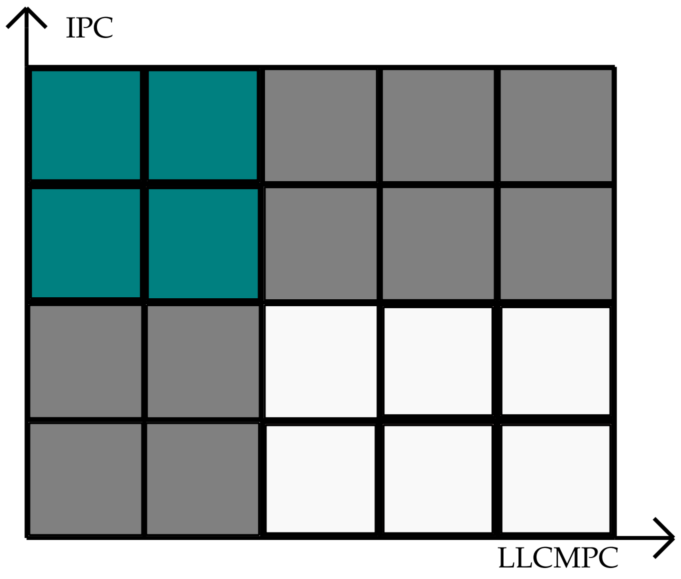


Figure 3.9: The dominance property enforces that the frequencies of the white bins must be lower or equal to the frequencies of the blue bins

The dominance property enforces that a bin will never have lower frequency than the bins it dominates. The advantage of this property is that bins will be updated although none of the kernels never covered their area explicitly. The algorithm for adding a new frequency to the lookup table is given in Algorithm 1.

Algorithm 1 Procedure for updating the lookup table

```
procedure UpdateTable(IPC, LLCMPC, Cores, Frequency)
    for Bin ∈ Bins do
        if $Bin.IPC \leq IPC \wedge Bin.LLCMPC \geq LLCMPC$ then
            if $Bin.Cores \geq Cores \wedge Bin.Frequency > Frequency$ then
                Bin.Frequency = Frequency
            end if
        end if
    end for
end procedure
```

3.1.3 Implementation of the intelligent agent

The intelligent agent is implemented as a pthread. The algorithm used by the agent is given in Algorithm 2. The agent sleeps while the performance counters collect data from the worker threads. When the agent wakes up, it will calculate the harmonic mean of IPC and LLCMPC from the running worker threads. In the article, *'Green Governors: A Framework for Continuously Adaptive DVFS'* [24] the authors also averaged the data from the performance counters, since the Intel i7 can only apply DVFS to the whole chip. The IPC, LLCMPC and number of running threads are used as indices into the lookup table, to pick the preferred frequency for the current state. The number of running threads indicates how many cores are active as long as Hyper-threading is disabled. It is assumed that Nanos++ is the only running application; if this is not the case, the implementation must be extended to take into account the total number of active cores. In the implementation, the lookup table contains indices which can be converted to frequencies in order to reduce the memory consumption. The agent will call cpufrequtils through a shell command if the frequency should be adjusted. The time it takes for the CPU to switch between two frequencies is 10μs. The idea of determining if DVFS should be applied on the basis of the system's current state is also used in [23], [10] and [24].

Algorithm 2 Intelligent Agent

```
procedure IntelligentAgent
    loop
        Sleep(Interval)
        ipc = readInstructionsPerCycle()
        llcmpc = readLastLevelCacheMissesPerCycle()
        frequency = table[ipc][llcmpc][runningThreads()]
        if frequency != currentFrequency then
            DVFS(frequency)
            currentFrequency = frequency
        end if
    end loop
end procedure
```

Chapter 4

Methodology

This chapter provides a detailed description of how the research has been carried out:

- Section 4.1 describes the application kernels that have been used during the research.
- Section 4.2 presents the setup of the experiments.
- Section 4.3 covers the experiment methodology which has been used in the study.

4.1 Benchmarks

This section provides an overview of the kernels that have been selected for benchmarking in this study.

4.1.1 Mont Blanc application kernels

The Mont-Blanc project aims to design "The Next Generation Supercomputer", and relies on the OmpSs programming model to handle hardware challenges. This section presents nine Mont Blanc application kernels that the CARD-group has access to. The kernels have been ported to OmpSs by the High Performance Computing Group from Universitat Politècnica de Catalunya[32].

Merge sort

The kernel sorts a random permutation of n integers with a parallel version of merge sort. The algorithm divides an array in two halves, sorting each recursively. For each recursive call a new task is generated. The merge is parallelized with a divide-and-conquer algorithm.

Histogram

The algorithm computes a histogram for the number distribution from an array of integers. Subsets of the array can be processed independently, however, the histograms must be merged at the end.

2D convolution

In image processing, the convolution operator is used as a filter to change the characteristics of an image. The kernel divides the image into blocks that can be processed in parallel. For each pixel, a filter that requires access to the neighboring pixels in order to compute the convolution is used.

3D stencil

3D stencil updates each element in a regular multidimensional grid according to the values from neighboring elements. The grid can be partitioned in order to be processed in parallel.

Dense matrix multiplication

Dense matrix multiplication computes the matrix product $C = A \bullet B$. The kernel partitions the matrices in blocks that can be processed in parallel. The matrix product of each block is computed with functionality from *cblas*, which is a C language interface to Basic Linear Algebra Subroutine(BLAS) libraries.

Sparse matrix vector multiplication

The kernel multiplies a compressed matrix with a vector. The problem size is divided in subsets that can be processed in parallel. The compressed matrix consists of pointers to non zero elements in the primary matrix.

Vector operation

Vector operation computes the sum of two vectors. The vectors are divided in subsets that can be processed independently.

N-body

N-body calculates the interaction between particles in a system. The algorithm computes the acceleration, velocity and future position of each particle in parallel.

Reduction

The kernel calculates the sum of all values in an array. Subsets of the array can be processed independently, however the results must be merged at the end.

4.1.2 The Barcelona OpenMP Task Suite

The Barcelona OpenMP Task Suite(BOTS)[33] provides a set of benchmarks targeting task level parallelism in OpenMP.

FFT

The Cooley-Tukey algorithm computes the one-dimensional fast fourier transform on a vector of n complex numbers. The algorithm recursively breaks down the Discrete

Fourier Transform into smaller problems, and for each division it generates additional tasks.

Fib

The algorithm calculates the nth fibonacci number by recursive parallelism. This is not an effective solution, however, the algorithm tests how well deep and balanced task trees can be parallelized.

SparseLU

The kernel computes the LU decomposition over sparse matrices. A primary matrix is composed of pointers to submatrices, however, a pointer is not allocated if the corresponding submatrix consists only of zeros. Each submatrix is a separate task that can be executed in parallel.

NQueens

The n queens puzzle is the problem of placing n chess queens on a nxn chessboard so that no queens attack each other. The algorithm relies on backtracking and pruning in order to find all possible solutions to the puzzle. For each recursive call a new task is generated, however, it can be specified that the algorithm should inline tasks after a certain depth.

Strassen

The Strassen algorithm is a divide-and-conquer algorithm which computes the matrix product $C = A \bullet B$, where it is required that n is an exact power of 2 in each of the n x n matrices. For each recursive step a new task is generated.

4.1.3 Others

Black Scholes

The Black Scholes model is a mathematical model of a financial market, which gives the price of an option over time. In finance, an option is a contract of selling and buying an underlying asset at a specified strike price prior to a given date. The Black-Scholes equation is a partial differential equation. The equation can be solved in parallel by partitioning the problem. The computational pattern of the algorithm is similar to what found in dense linear algebra.

Quick Sort

The algorithm sorts a random permutation of n integers with a parallel version of quick sort. For each of the recursive calls in the algorithm a new task will be generated.

Unstructured 3D stencil

The kernel updates each element in an irregular multidimensional grid according to the values from neighboring elements. Neighboring elements must be accessed through indices, since the grid is unstructured. The grid can be partitioned into independent tasks in order to be processed in parallel.

4.2 Experimental Setup

This section presents the hardware and software packages used in the research. Additionally, it provides an overview of how software packages, benchmarks and the scheduling-plugin have been compiled.

4.2.1 Hardware

Experiments were run on Sandy Bridge-EP, a research computer designed to resemble a node from the Vilje supercomputer[34]. Figure 4.1 provides a conceptual overview of how cores and caches are organized.

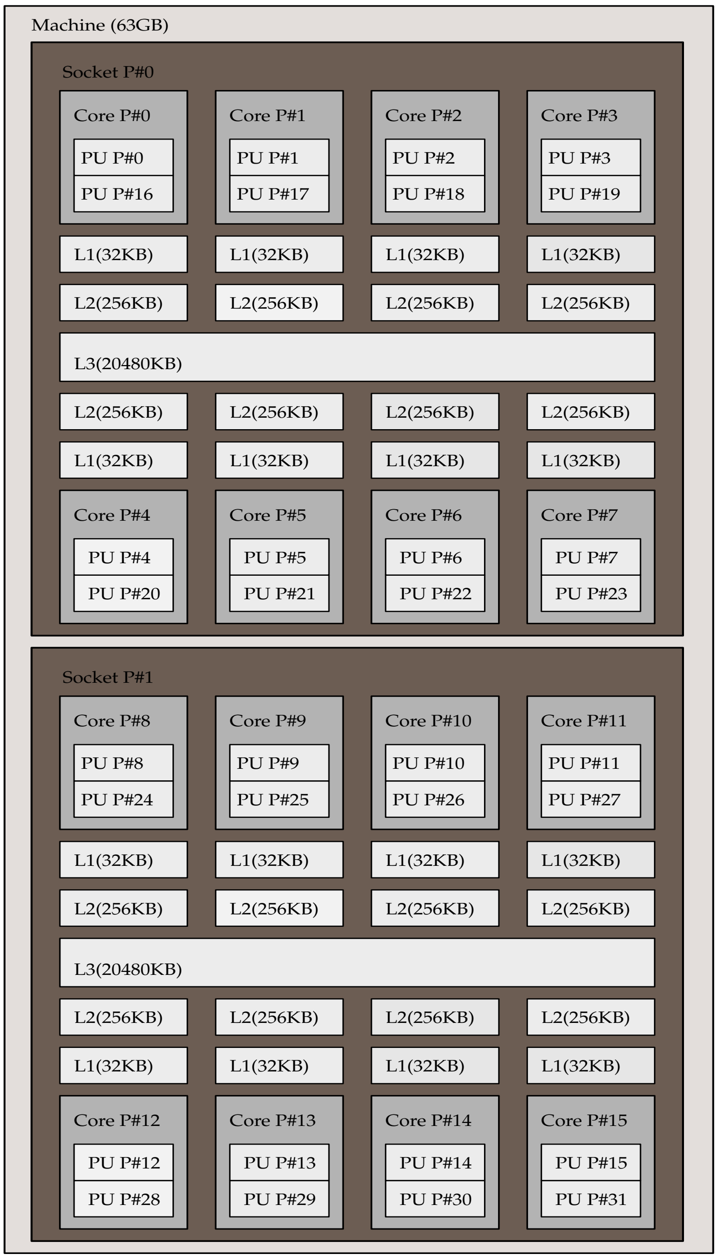


Figure 4.1: Diagram of caches, cores and hyper-threads on Sandy Bridge-EP

For an overview of the similarities and differences between Sandy Bridge-EP and a node from Vilje, see Table 4.1.

	Sandy Bridge-EP	**Vilje Node**
Processor	2 * Intel® Xeon® CPU E5-2670	2 * Intel® Xeon® CPU E5-2670
Motherboard	ASUS Z9PE-D8WS SSI-EEB	ASUS Z9PE-D8WS SSI-EEB
DRAM	2 * Corsair Vengeance DDR3 1600MHz	Corsair Vengeance DDR3 1600MHz
OS	Fedora 16(x86_64) with Linux kernel 3.5.3	SuSE SLES11

Table 4.1: Similarities and differences between the Sandy Bridge-EP and a node in Vilje

Hardware specifications for Intel® Xeon® CPU E5-2670 are listed in Table 4.2. Information about the processor is obtained from */proc/cpuinfo* and *Intel ARK*[35]. Information about the internal cache system is retrieved from */sys/devices/system/cpu/cpu0/cache/index**.

Property	**Value**
CPU model	Intel® Xeon® CPU E5-2670
Model #	45
Stepping	7
Manufacturing process	32nm
Clock frequency (min-max)	1.20GHz - 2.60GHz
Max Turbo Frequency	3.30GHz
Number of physical cores	16
Number of logical cores	32
Scalability	2 Sockets

Table 4.2: Specification for Intel® Xeon® CPU E5-2670

Cache	**Size**	**Ways of associativity**	**Line size**
Level 1	32KB(Data)/32KB(Instr)	8	64B
Level 2	256KB	8	64B
Level 3	20MB (shared)	20	64B

Table 4.3: Cache information for Intel® Xeon® CPU E5-2670

4.2.2 Software and Libraries

Table 4.4 lists the software and libraries used in the research.

Software	Version	Licence	Website
g++	4.6.3	GNU GPL	
gcc	4.6.3	GNU GPL	
Nanos++	0.7a-2013-02-22	GNU LGPL	http://pm.bsc.es/nanox-downloads
Mercurium	1.3.5.8	GNU LGPL	https://pm.bsc.es/projects/mcxx
ATLAS	3.10.1	BSD Licence	http://math-atlas.sourceforge.net
PAPI	4.4.0.0	BSD Licence	http://icl.cs.utk.edu/papi

Table 4.4: Software and libraries used in the research

4.2.3 Compilation

Nanos++ and Mercurium require gcc 4.6.3, M4, FLEX and GPERF in order to compile. The software packages listed in Table 4.4 includes Makefiles, so compiling and linking are performed automatically.

OmpSs Packages	Compiler suite	Compiler flags
Bison	`gcc`	`-O2`
Nanos++ runtime	`gcc`	`-O2`
Mercurium	`gcc`	`-O2`
Other Packages		
ATLAS	`gcc`	
PAPI	`gcc`	

Table 4.5: Compiler flags for software and libraries used in the study

OmpSs Packages	./configure flags
Mercurium	`--enable-tl-openmp-nanox` `--enable-ompss --enable-tl-superscalar` `--with-superscalar-runtime-api-version=5`

Table 4.6: Configure flags for Mercurium

The benchmarks have been compiled with Mercurium 1.3.5.8. If performance counters should be used to gather data for the lookup table presented in Section 3.1.2, it is necessary to link with PAPI and PFM.

Software	Compiler suite	Compiler flags
Mont Blanc applications		
Merge sort	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
Reduction	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
Histogram	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
2d convolution	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
3d stencil	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
Dense matrix multiplication	`sscc`	`--ompss -lpapi -lpfm -fopenmp -lcblas`
Sparse matrix vector multiplication	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
Vector operation	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
N-body	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
BOTS		
FFT	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
Fib	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
SparseLU	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
NQueens	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
Strassen	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
Others		
Black Scholes	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
Quick Sort	`sscc`	`--ompss -lpapi -lpfm -fopenmp`
Unstructured 3d stencil	`sscc`	`--ompss -lpapi -lpfm -fopenmp`

Table 4.7: Compiler flags for application kernels

Compilation of scheduling-plugin

The developed scheduling-plugin must be compiled to a shared library before it can be accessed from Nanos++. If the shared library is named **libnanox-sched-agent.so**, then the developed scheduling-plugin can be set with the command line flag: **--schedule agent**. Listing 4.1 provides instructions for how the developed scheduling-plugin can be integrated with Nanos++.

Listing 4.1 Compilation of scheduling-plugin

```
g++ -fpic -c agent_sched.cpp
g++ -shared -o libnanox-sched-agent.so agent.o -lpapi -lpfm
mv libnanox-sched-agent.so "PATH"/NANOS/lib/performance/
OMP_NUM_THREADS=threads NX_ARGS="--schedule agent" ./application
```

4.3 Experiment Methodology

This section describes how the experiments have been conducted. In addition, it explains how performance and energy efficiency have been measured.

4.3.1 Training the lookup table

The lookup table must be trained before it can be utilized by the intelligent agent to predict the most energy-efficient frequency. Two choices are important to consider when the lookup table should be trained:

- Which metric should the table optimize for?
- How should the table be trained?

In this study, four lookup tables have been trained. One of the lookup tables optimizes for the metric $\frac{\text{Operations}}{\text{Joule}}$, another tries to minimize the *Energy Delay Product*, whereas the last two optimizes for the metrics $\frac{\text{Operations}}{\text{Joule}}$ and *Energy Delay Product*, but under the constraint that performance cannot suffer by more than 10% compared to the execution at the highest frequency. This choice is made in order to test how the agent performs for various metrics.

The way the lookup table should be trained depends on how it will be used by the agent. The lookup table can either specifically be trained on *'entire libraries of kernels used in HPC applications'* or it can be trained to recognize *'the computational patterns found in the most common applications'*. Training the lookup table to recognize entire libraries may provide the highest accuracy if applications consist solely of known kernels, but the lookup table may be inflexible and make poor predictions for unknown kernels. The lookup table can become more robust against unknown kernels if it is trained to recognize the most common computational patterns found in real applications, however it may lack the accuracy one can obtain by tuning the table to recognize specific kernels.

Table 4.8 provides an overview of the computational patterns that are found in each kernel used in this study. In addition, it lists whether the kernel has been used for training the lookup table.

4.3.2 Experiments

The experiments have been carried out without the use of hyper-threading or Intel Turbo Boost. The temperature can affect the leakage current[36]. Therefore, before starting any experiments, the temperature of the processors were raised to 40 degrees Celsius in order to create equal conditions for each experiment. Distributed Breadth First(DBF) is used as scheduling-plugin for each experiment that obtained data for the lookup table. Execution time, energy, IPC and LLCMPC have been measured for all possible *frequency/thread* pairs for a given kernel. *'Process Cruise Control'* and *'Green Governors: A Framework for Continuously Adaptive DVFS'* apply a similar procedure to gather data for each kernel used in the research.

Kernel	**Dwarf**	**Selected for training**
Dense matrix multiplication	Dense Linear Algebra	Yes
Sparse matrix vector multiplication	Sparse Linear Algebra	Yes
3d stencil	Structured Grids	Yes
N-body	N-body Methods	Yes
FFT	Spectral Methods	Yes
NQueens	Backtrack and Branch-and-Bound	Yes
Histogram	Map Reduce / Unstructured Grids	Yes
Merge Sort	Graph Traversal	Yes
Quick Sort	Graph Traversal	No
Reduction	Map Reduce / Dense Linear Algebra	No
Black Scholes	Dense Linear Algebra	No
Vector operation	Dense Linear Algebra	No
Fibonacci	Graph Traversal	No
Strassen	Dense Linear Algebra	No
SparseLU	Sparse Linear Algebra	No
2d Convolution	Structured Grids	No
Unstructured 3d stencil	Unstructured Grids	No

Table 4.8: Classification of the computational patterns in the selected kernels

The developed scheduling-plugin has been run with each kernel. The benefit of such an approach is that it tests whether the agent is capable of accurately predicting the frequency for both known and unknown kernels. If the agent predicts the frequency of known kernels with a high accuracy, it demonstrates that the developed scheduling-plugin can be successful if the lookup table is trained to recognize *'entire libraries of kernels used in HPC applications'*. The lookup table has been trained with kernels that have different computational patterns. If the agent is capable of predicting the frequency of the unknown kernels with a high accuracy, it indicates that the lookup table can be trained to recognize computational patterns.

To measure the accuracy of the intelligent agent, the *predicted frequency* has been compared against the optimal frequency found for each kernel. Note that the *predicted frequency* is calculated as the average of all frequencies predicted by the agent while an experiment has been run. The intelligent agent has been configured to sleep for an interval of 250ms before it predicts what will be the most energy efficient frequency for the next period. By considering intervals ranging from 50ms to 1000ms, it was found by trial and error that an interval of 250ms provided the best balance between accuracy and overhead for invocation of the intelligent agent.

4.3.3 Experiment configurations

This section provides an overview of how the experiments have been configured. Three factors have influenced how the parameters have been set for each benchmark:

- When should tasks be throttled?
- How should idle threads be handled?
- Which problem size should be used?

The throttle policy determines if tasks should be created as entities that can be scheduled in the runtime system for asynchronous execution, or if the code should be executed immediately. All benchmarks in this study have been configured to use the throttle policy **taskdepth**. The throttle policy taskdepth stops creation of new entities after the depth of a task has passed a certain threshold. The depth of a task will always be one more than the parents depth.

If there are insufficient available tasks to keep all the threads occupied, one must consider whether idle threads should be busy waiting or enter a sleep-mode. The advantage of busy waiting is that threads will quickly be able to start on new tasks that become available in the system, but as a consequence busy waiting results in unnecessary waste of energy and contention for shared resources. An idle thread can relinquish its processing resources by entering a sleep-mode. If the system has more cores than available tasks, the redundant cores can be commanded to enter a low-power mode to reduce the energy consumption. However, if a thread has entered a sleep-mode it will not be able to start on a new task until its specified sleep time has expired. Nanos++ combines the use of busy waiting and putting idle threads to sleep. How long a thread should busy wait or sleep can be configured with the command *'--spins INTEGER --sleep-time INTEGER'* before starting an OmpSs application. In this study, experiments have been made with both busy waiting and letting idle threads sleep for longer periods. The choice of letting idle threads sleep was made on the basis of trial and error for each benchmark.

It is important that the problem size for each benchmark is selected in such a way that measurements becomes stable. Variation in the measurement tools will have greater impact on applications with short execution time. However if the execution time is too long, the energy measurements will give erroneous results due to overflow. How energy measurements are performed are discussed in Section 4.3.5. Several problem sizes have been used for each benchmark in order to ensure that the results are stable. For a detailed overview of the configuration to the experiments see Appendix 7.2.6.

4.3.4 Lookup table

This section provides an overview of how the lookup tables have been configured. Table 4.9 lists the configuration of the lookup tables that have been trained in this research.

Property	Value
IPC_{range}	0.000 - 2.625
$LLCMPC_{range}$	0.000 - 0.00375
Cores	16
IPC_{bins}	35
$LLCMPC_{bins}$	5

Table 4.9: Configuration for the lookup tables

Figure 4.2, 4.3, 4.4 and 4.5 illustrates how the frequency varies with IPC and LLCMPC when the number of cores are 16. Appendix 7.2.6 provides an overview of all core configurations.

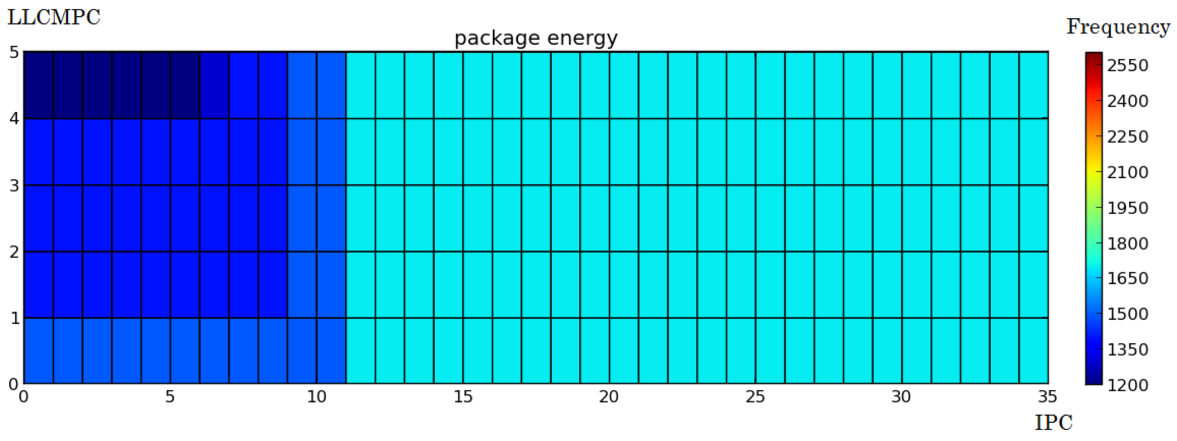


Figure 4.2: Lookup table for optimizing Operations per Joule for core configuration 16

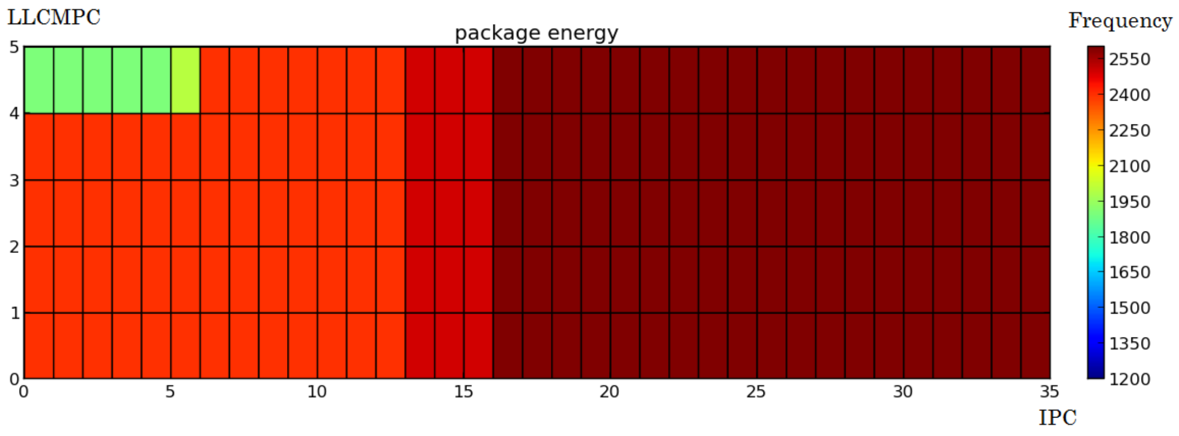


Figure 4.3: Lookup table for optimizing Energy Delay Product for core configuration 16

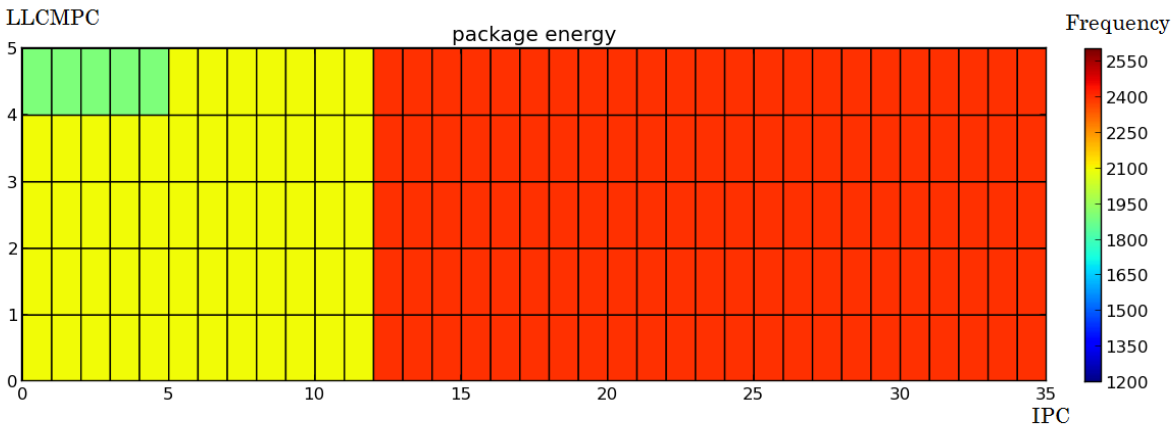


Figure 4.4: Lookup table for optimizing Operations per Joule under the constraint that performance cannot suffer by more than 10% compared to the execution at the highest frequency for core configuration 16

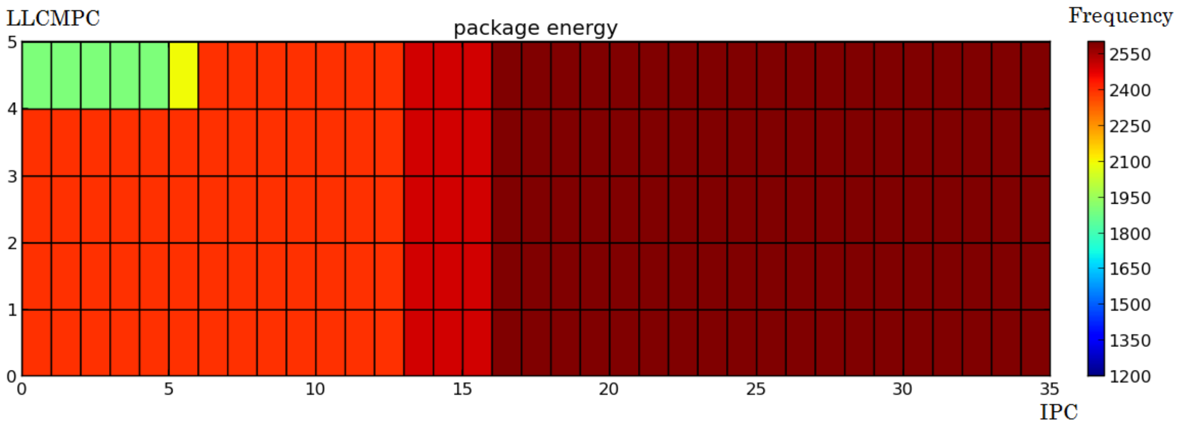


Figure 4.5: Lookup table for Energy Delay Product under the constraint that performance cannot suffer by more than 10% compared to the execution at the highest frequency for core configuration 16

Values that are outside the range of what the table can cover will be clamped to maximum IPC or LLCMPC. The size of the parameter space and the number of bins have been selected with trial and error in order to find a configuration that are sufficient to resist variance from the measurement tools. Figures 4.6, 4.7, 4.8 and 4.9 present the problems that can occur due to variance in the measurements. The frequency domain illustrated in Figure 4.6 represents the optimal lookup table. The optimal size of the parameter space and the number of bins are affected by how many application kernels that are available for training the lookup table. Small bins are preferable if the training set is large. However, small bins can perform poorly due to variance in the measurements if the training set contains few samples as illustrated in Figure 4.7. The application kernels used during the training phase will index the lookup table with IPC, LLCMPC and the number of active cores. It is unlikely that an application will generate the exactly same IPC and LLCMPC twice, therefore, one must ensure that the bins are large enough to allow for a certain variation in the measurements. Large bins makes it easier to cover the parameter space with few application kernels, however the number of collisions when updating the lookup table can increase.

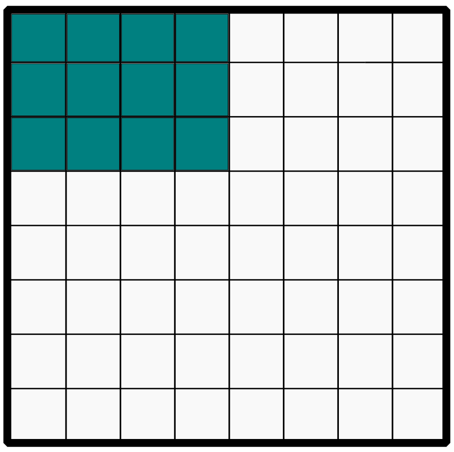


Figure 4.6: Optimal frequency domain

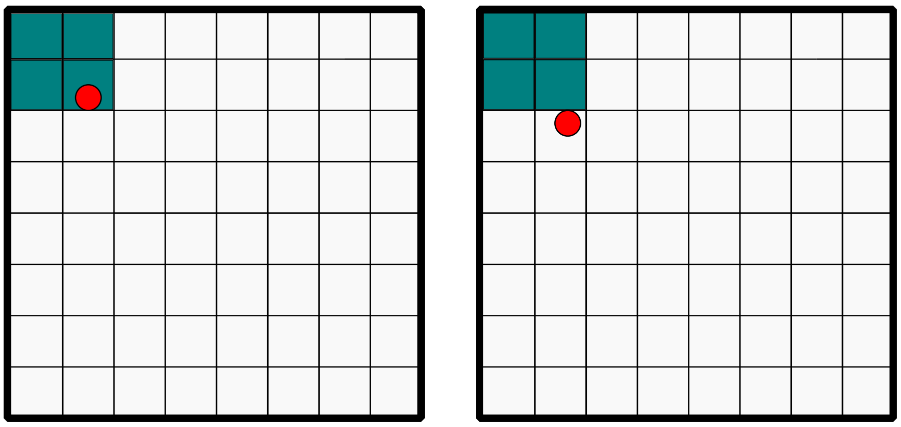


Figure 4.7: The agent predicts wrong due to variance in measurements and small bins

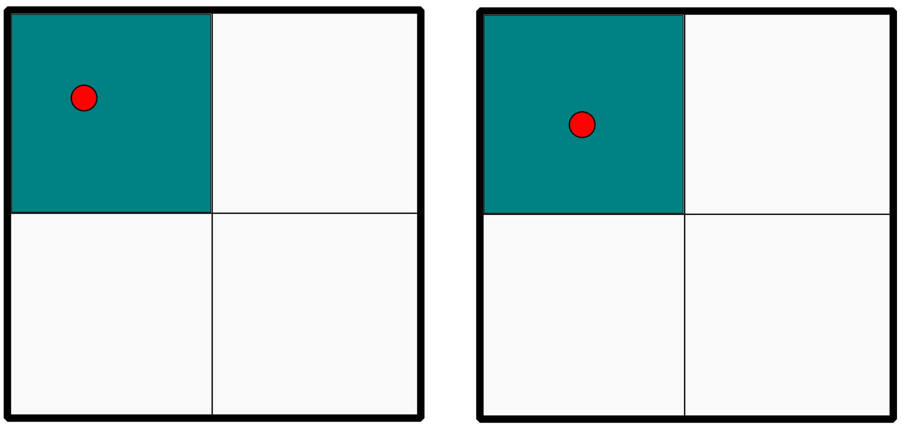


Figure 4.8: The agent predicts correct, however the bins are too large to replicate the optimal frequency domain

Figure 4.9 illustrates an alternative strategy to make the training phase more robust against variations in the measurements. One can ensure that small variations in the measurements will be less significant on the overall result if every bin within a certain radius are updated. Further work could examine how to develop software for auto-tuning of the lookup table configuration based on the number of application kernels available for training.

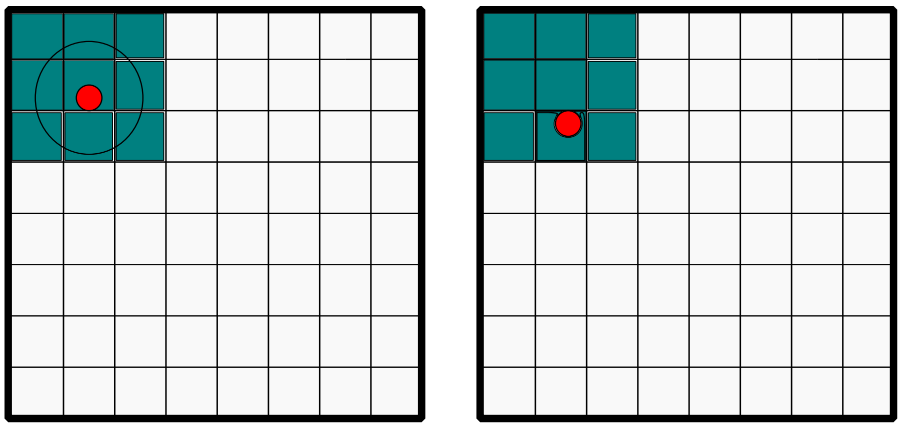


Figure 4.9: The agent predicts correctly since the training phase takes the variation in the measurements into account

4.3.5 Measurement tools

Execution time

Execution time is measured with **omp_get_wtime()**. The function is part of the OpenMP API, and calls the POSIX function **gettimeofday()** in order to return the elapsed wall clock time in seconds represented with double-precision floating point. The Linux man page specifies that the resolution of **gettimeofday()** is in microseconds.

Performance counters

The Intel Sandy Bridge Microarchitecture has the ability to measure four performance counters simultaneously per thread. In this study, three performance counters were measured through the PAPI interface. Table 4.10 lists the performance counters that have been measured, for more information about the performance counters in Sandy Bridge see *'Intel Architecture Developer's Manual Volume 3B, Appendix A'*[37].

PAPI event	Description
`PAPI_TOT_INS`	Instructions completed
`PAPI_L3_TCM`	Level 3 cache misses
`PAPI_TOT_CYC`	Total cycles

Table 4.10: Performance counters used in the research

Energy

A software library has been developed at NTNU which is capable of reading the energy consumption of the processor. The library is based on the utility program `rdmsr`, which reads the energy consumption through Running Average Power Limit(RAPL) Model-specific register(MSR). RAPL is available in the Sandy Bridge microarchitecture, and provides sensors to measure energy consumption of the processor components. Table 4.11 lists the MSRs read by the library. The MSRs are accessible from the device file located at */dev/cpu/*/msr* after loading the `msr` module.

MSR	Description
MSR_PKG_ENERGY_STATUS	Reports measured energy usage for the processor die
MSR_PP0_ENERGY_STATUS	Report actual energy usage to the processor cores

Table 4.11: MSRs used for energy measurement

According to *'Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B'*[38], the MSRs are updated every ~1 msec, and have a wraparound time of 60 secs when power consumption is high, otherwise it may be longer. The library takes one wraparound into account, therefore, the library can keep track of energy consumption for 120 seconds. The energy measurements from the MSRs are expressed in Joules, scaling must be applied in order to make the measurements meaningful in a finite number of bits. It is specified that the scaling factor is 15.3 micro-Joules. Equation 4.1 describes how the library converts values read from RAPL MSRs to millijoules.

$$\text{Millijoules} = \frac{\text{MSR\_ENERGY\_STATUS} * 15.3}{1000} \tag{4.1}$$

The motherboard, main memory and hard drive also consume power, however, `MSR_PKG_ENERGY_STATUS` only reports the energy consumption of the processor die. It is possible to measure the energy consumption of the entire system with devices such as Yokogawa WT210 that are connected between the power supply and the computer. In the paper *'Improving energy efficiency through parallelization and vectorization on Intel® Core i5 and i7 processors'*, Juan M. Cebrián, Lasse Natvig and Jan Christian Meyer measured the average total power consumption for different combinations of processors, vectorization technologies and thread configurations. It will only be energy efficient to adjust the frequency for I/O intensive tasks when the total power consumption of Sandy Bridge-EP is considered. The experiments performed in this study only measured the energy consumption of the processor die, in order to create situations where it is energy efficient to adjust the frequency due to outstanding memory requests. Measurements for Intel® Core i5 indicate that the processor accounts for larger parts of the total energy consumption in systems initially designed for energy efficiency. From the experiments that have been carried out in this study it has been measured that 75% of the energy consumption of Intel® Xeon® CPU E5-2670 is spent on the cores while the rest on shared resources when the processor operates at 2600 MHz.

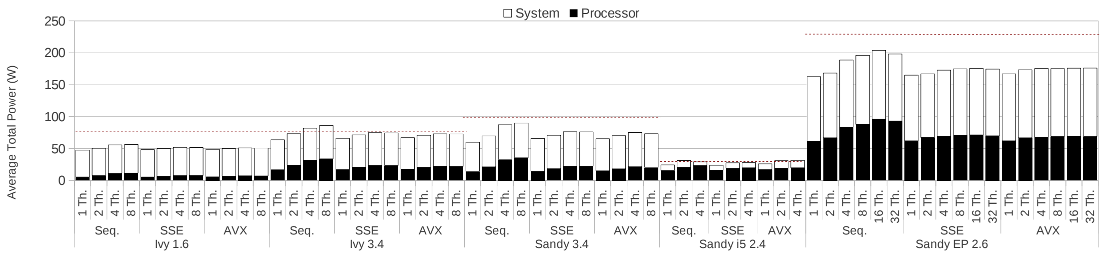


Figure 4.10: Power Measurements for different combinations of processors, vectorization technologies and thread configurations[39]

Chapter 5

Results

This chapter presents the results for how the developed scheduling-plugin performed for each of the 17 application kernels that have been used during the research. The accuracy of the developed scheduling-plugin has been measured by comparing the predicted frequency against the optimal frequency found for each application kernel. The results present how the developed scheduling-plugin performs compared to the Distributed Breadth First scheduler running at 2600 MHz in terms of performance and energy consumption. The chapter covers the results for two thread configurations in order to keep the content clear. A complete overview of the results is presented in Appendix 7.2.6. In this study, four lookup tables have been trained that optimize towards various metrics. Therefore, the chapter has been divided into four sections that present the results for each of the metrics:

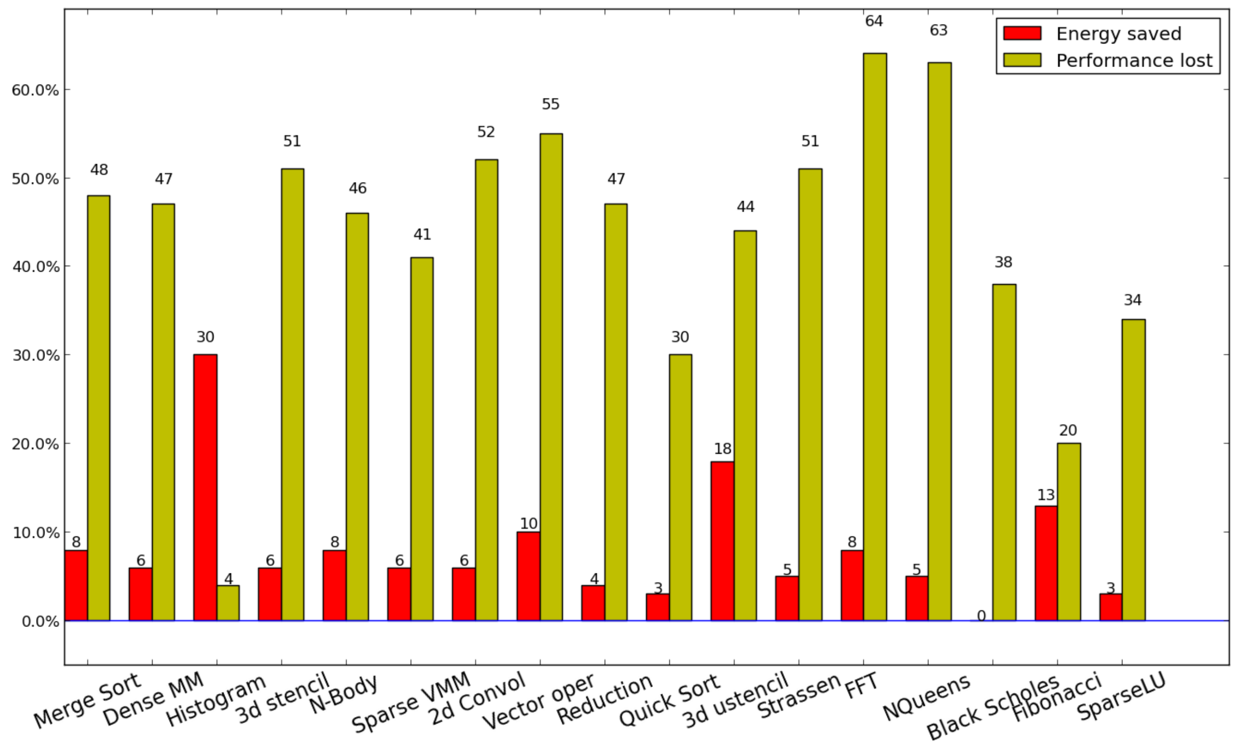
- Section 5.1 presents the results for the metric $\frac{\text{Operations}}{\text{Joule}}$.
- Section 5.2 shows the results for when the *Energy Delay Product* is applied as metric.
- Section 5.3 presents how the agent performs when $\frac{\text{Operations}}{\text{Joule}}$ is applied as metric with an additional constraint that performance cannot suffer by more than 10% compared to the execution at the highest frequency.
- Section 5.4 presents how the agent performs for the metric *Energy Delay Product* with the additional constraint that performance cannot suffer by more than 10% compared to the execution at the highest frequency.

5.1 Results for the metric Operations per Joule

Reducing the energy consumption will be prioritized when the agent optimizes for the metric $\frac{\text{Operations}}{\text{Joule}}$. It will be energy efficient to lower the frequency while the dynamic power dissipation accounts for most of the power consumption, even if performance declines. As can be seen from the results, there is greater potential to save energy when several cores are active, since the static power dissipation will be small compared to the dynamic power dissipation.

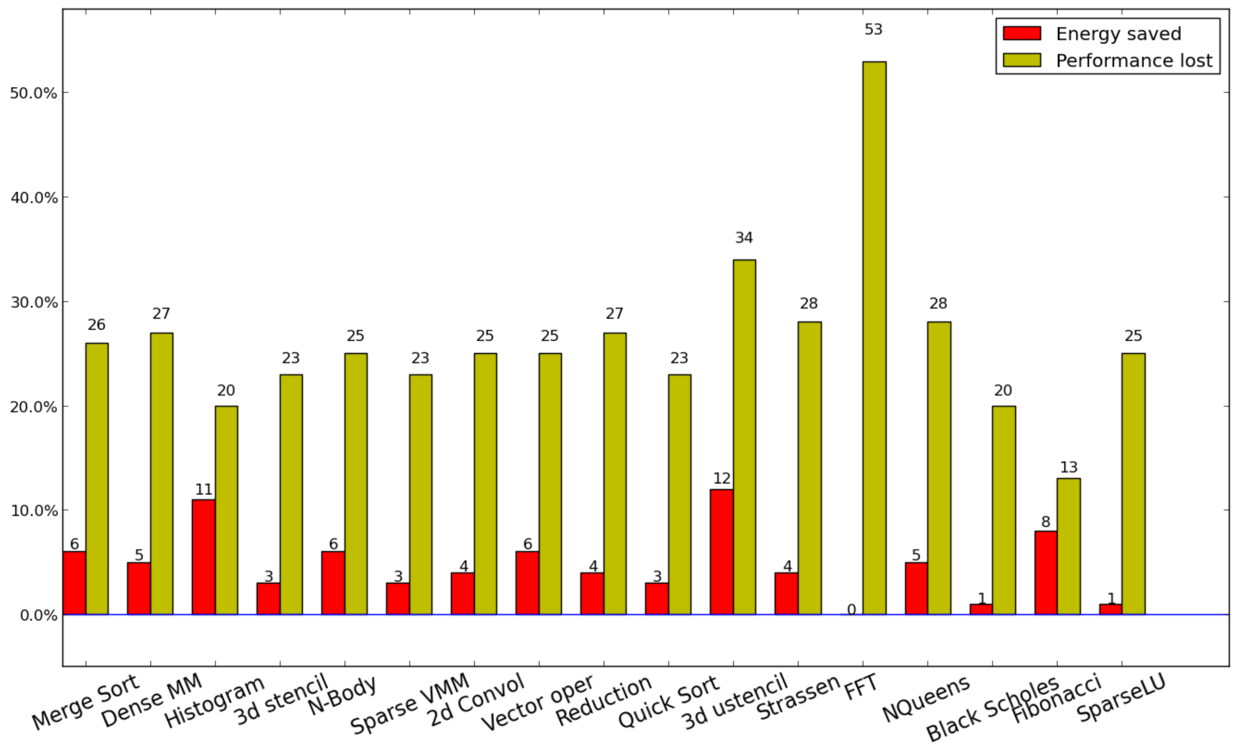
5.1.1 Results for 16 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	1700 MHz	1713 MHz	0.76 %
Reduction	1700 MHz	1758 MHz	3.41 %
Histogram	1200 MHz	1689 MHz	40.75 %
2d convolution	1900 MHz	1700 MHz	10.53 %
3d stencil	1700 MHz	1704 MHz	0.24 %
Dense matrix multiplication	1700 MHz	1722 MHz	1.29 %
Sparse matrix vector multiplication	2100 MHz	1849 MHz	11.95 %
Vector operation	1700 MHz	1600 MHz	5.88 %
N-body	2000 MHz	1700 MHz	15.00 %
FFT	1600 MHz	1400 MHz	12.50 %
SparseLU	1800 MHz	1933 MHz	7.39 %
NQueens	1800 MHz	1600 MHz	11.11 %
Strassen	1800 MHz	1600 MHz	11.11 %
Black Scholes	1800 MHz	1927 MHz	7.06 %
Fibonacci	1400 MHz	1738 MHz	24.14 %
Quick Sort	2100 MHz	1964 MHz	6.48 %
Unstructured 3d stencil	1200 MHz	1360 MHz	13.33 %



5.1.2 Results for 12 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2000 MHz	2000 MHz	0.00 %
Reduction	1900 MHz	2019 MHz	6.26 %
Histogram	1400 MHz	1892 MHz	35.14 %
2d convolution	2000 MHz	2061 MHz	3.05 %
3d stencil	1900 MHz	2083 MHz	9.63 %
Dense matrix multiplication	2000 MHz	2010 MHz	0.50 %
Sparse matrix vector multiplication	2000 MHz	2104 MHz	5.20 %
Vector operation	1700 MHz	2000 MHz	17.65 %
N-body	2000 MHz	2060 MHz	3.00 %
FFT	1700 MHz	1644 MHz	3.29 %
SparseLU	1900 MHz	2093 MHz	10.16 %
NQueens	2000 MHz	2000 MHz	0.00 %
Strassen	1800 MHz	2000 MHz	11.11 %
Black Scholes	1900 MHz	2132 MHz	12.21 %
Fibonacci	2300 MHz	2051 MHz	10.83 %
Quick Sort	2100 MHz	2103 MHz	0.14 %
Unstructured 3d stencil	1500 MHz	1612 MHz	7.47 %

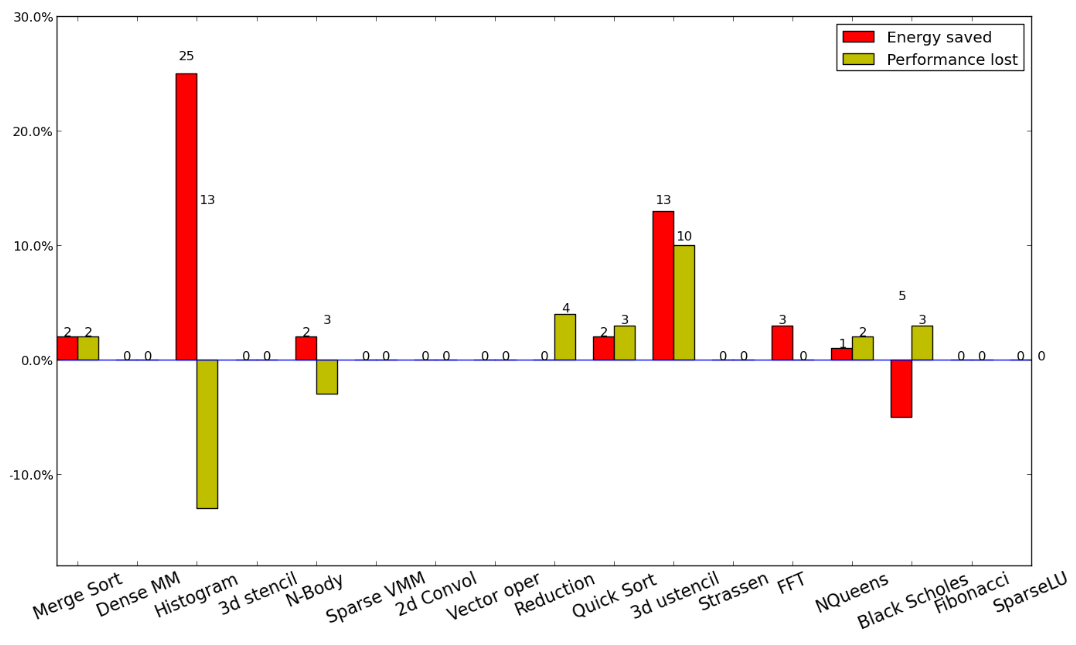


5.2 Results for the metric Energy Delay Product

The metric *Energy Delay Product* assigns equal weight to both energy and performance. This means that the agent can not lower the frequency as long as the reduction in performance is greater than the energy saved. The results indicate that only the applications with computational patterns similar to 'Unstructured Grids' will be able to save energy. The overhead of the intelligent agent in terms of energy consumption becomes more prominent when there are few opportunities to reduce the frequency.

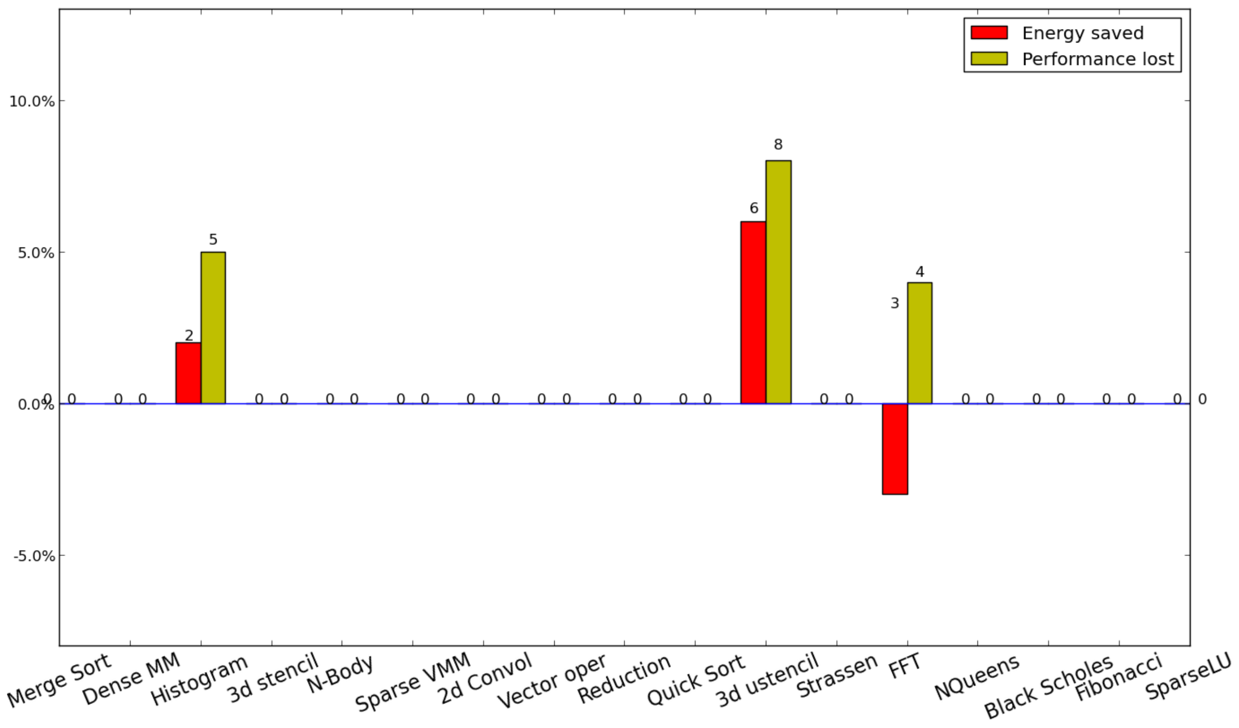
5.2.1 Results for 16 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2490 MHz	4.23 %
Reduction	2600 MHz	2505 MHz	3.65 %
Histogram	2000 MHz	2224 MHz	11.20 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2500 MHz	2598 MHz	3.92 %
FFT	2500 MHz	2363 MHz	5.48 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2549 MHz	1.96 %
Strassen	2400 MHz	2600 MHz	8.33 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2462 MHz	5.31 %
Unstructured 3d stencil	2100 MHz	2067 MHz	1.57 %



5.2.2 Results for 12 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2500 MHz	2434 MHz	2.64 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2400 MHz	7.69 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2300 MHz	2600 MHz	13.04 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2400 MHz	2248 MHz	6.33 %

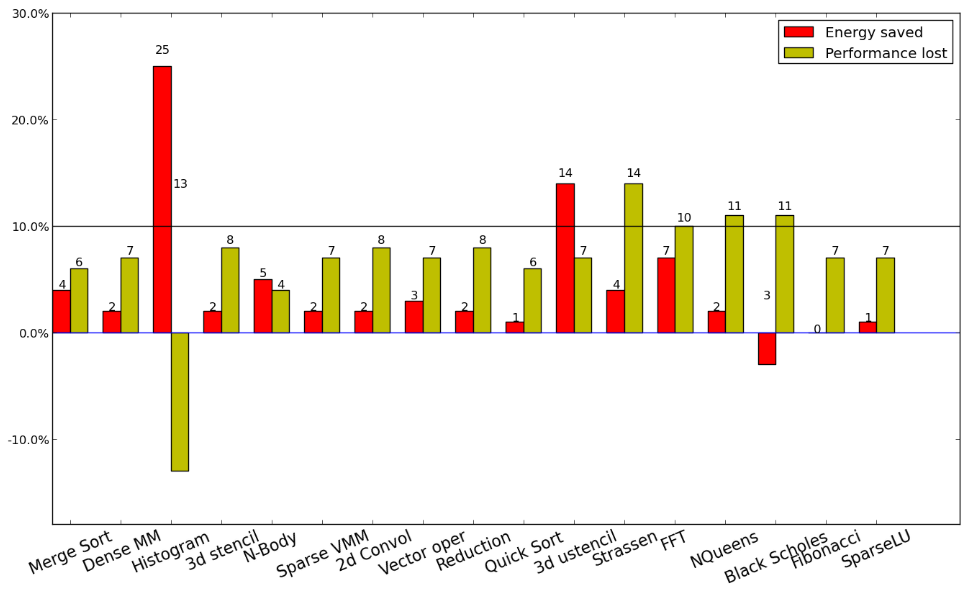


5.3 Results for the metric Operations per Joule under the constraint that performance cannot suffer by more than 10%

The agent tries to minimize the energy consumption under the constraint that performance cannot suffer by more than 10%. However, there exist situations where the agent is unable to satisfy the performance constraint. The agent will only be invoked every 250ms, which limits the reaction time for how quickly the frequency can be adjusted. In addition, if the agent predicts incorrect frequency for an application it can result in performance loss greater than 10%.

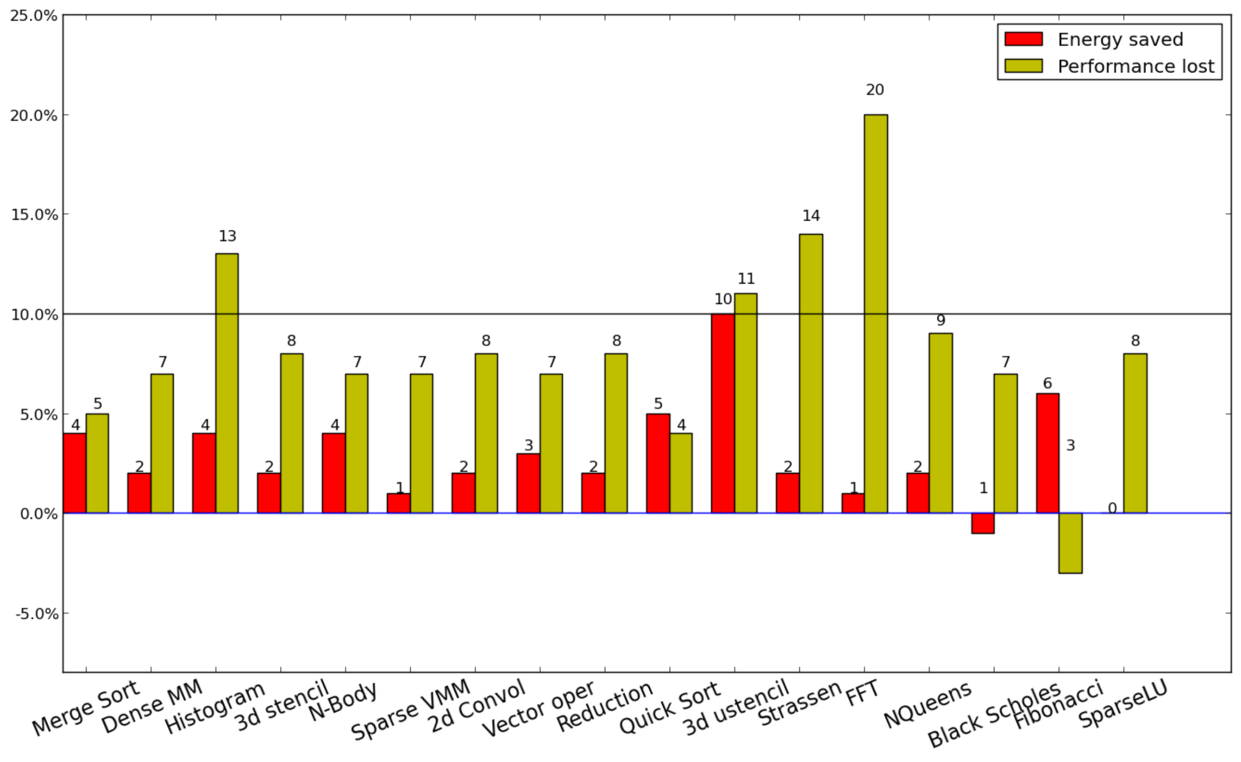
5.3.1 Results for 16 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2400 MHz	2393 MHz	0.29 %
Reduction	2400 MHz	2400 MHz	0.00 %
Histogram	2000 MHz	2197 MHz	9.85 %
2d convolution	2400 MHz	2400 MHz	0.00 %
3d stencil	2400 MHz	2400 MHz	0.00 %
Dense matrix multiplication	2400 MHz	2405 MHz	0.21 %
Sparse matrix vector multiplication	2500 MHz	2432 MHz	2.72 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2500 MHz	2403 MHz	3.88 %
FFT	2200 MHz	2100 MHz	4.55 %
SparseLU	2400 MHz	2425 MHz	1.04 %
NQueens	2400 MHz	2400 MHz	0.00 %
Strassen	2300 MHz	2242 MHz	2.52 %
Black Scholes	2400 MHz	2422 MHz	0.92 %
Fibonacci	2600 MHz	2418 MHz	7.00 %
Quick Sort	2600 MHz	2410 MHz	7.31 %
Unstructured 3d stencil	2200 MHz	2102 MHz	4.45 %



5.3.2 Results for 12 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2400 MHz	2400 MHz	0.00 %
Reduction	2400 MHz	2400 MHz	0.00 %
Histogram	2500 MHz	2220 MHz	11.20 %
2d convolution	2400 MHz	2400 MHz	0.00 %
3d stencil	2400 MHz	2401 MHz	0.04 %
Dense matrix multiplication	2400 MHz	2402 MHz	0.08 %
Sparse matrix vector multiplication	2400 MHz	2423 MHz	0.96 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2400 MHz	2400 MHz	0.00 %
FFT	2500 MHz	2135 MHz	14.60 %
SparseLU	2400 MHz	2424 MHz	1.00 %
NQueens	2400 MHz	2400 MHz	0.00 %
Strassen	2400 MHz	2238 MHz	6.75 %
Black Scholes	2600 MHz	2431 MHz	6.50 %
Fibonacci	2300 MHz	2400 MHz	4.35 %
Quick Sort	2500 MHz	2419 MHz	3.24 %
Unstructured 3d stencil	2200 MHz	2100 MHz	4.55 %

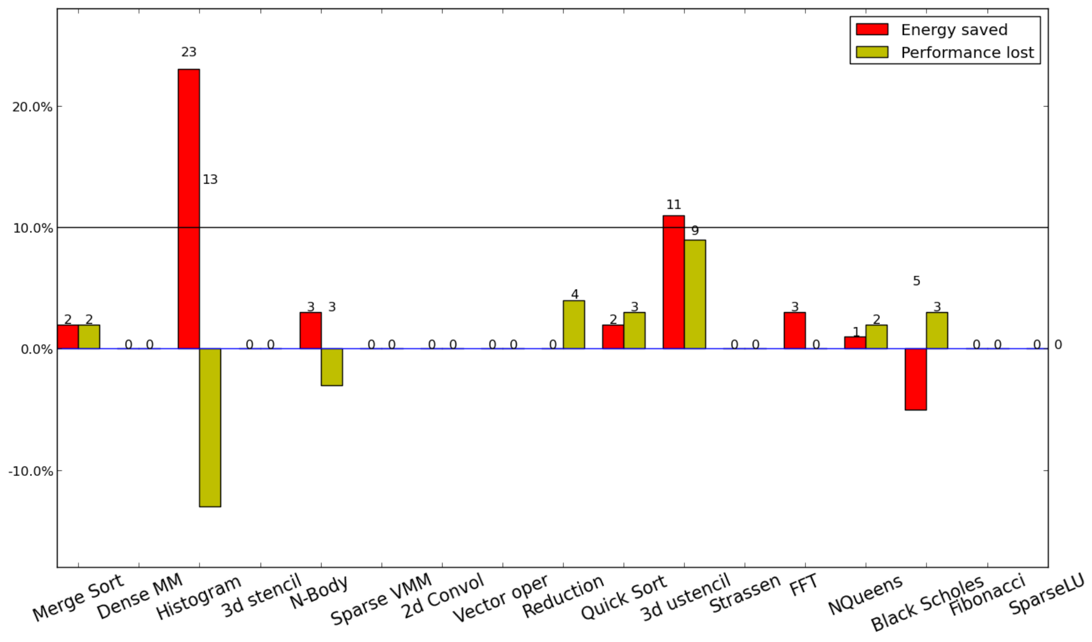


5.4 Results for the metric Energy Delay Product under the constraint that performance cannot suffer by more than 10%

The agent will minimize EDP under the constraint that performance cannot suffer by more than 10% compared to the execution at the highest frequency. The metric maintains quality of service, and ensures that there will be a balance between the energy saved and performance loss.

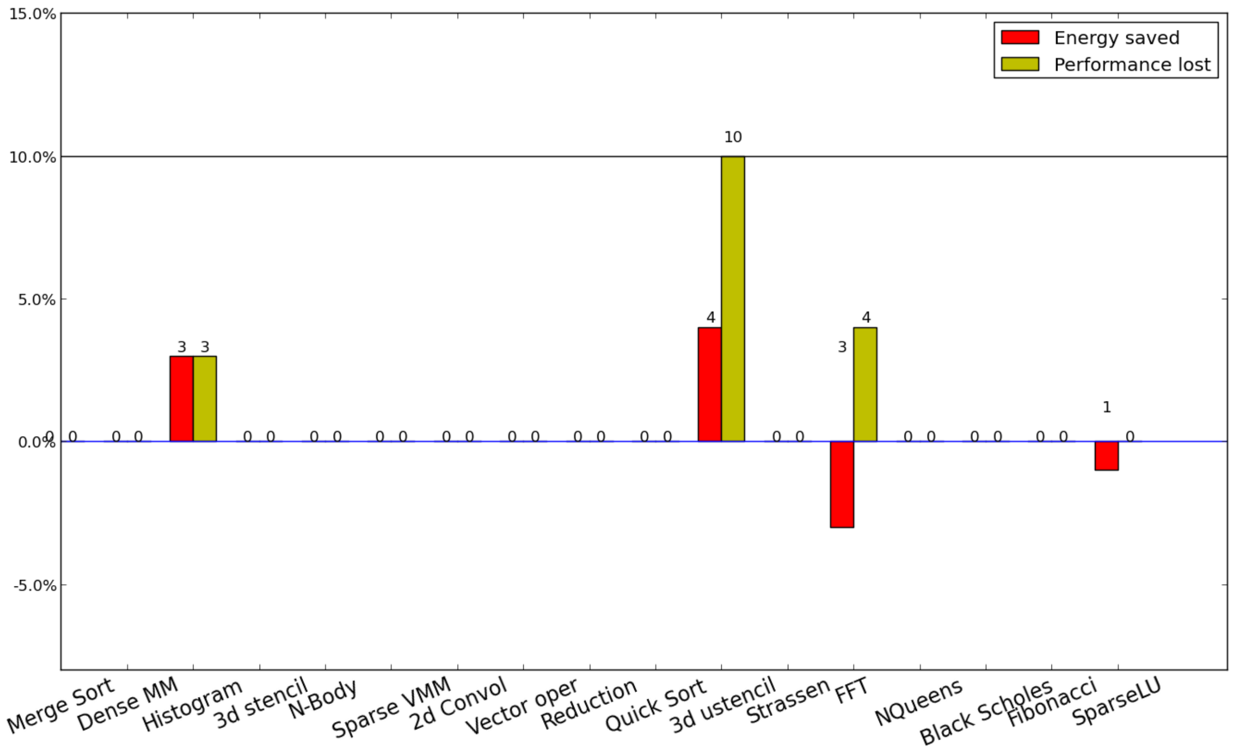
5.4.1 Results for 16 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2493 MHz	4.12 %
Reduction	2600 MHz	2505 MHz	3.65 %
Histogram	2000 MHz	2250 MHz	12.50 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2500 MHz	2598 MHz	3.92 %
FFT	2500 MHz	2363 MHz	5.48 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2549 MHz	1.96 %
Strassen	2400 MHz	2600 MHz	8.33 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2462 MHz	5.31 %
Unstructured 3d stencil	2200 MHz	2138 MHz	2.82 %



5.4.2 Results for 12 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2500 MHz	2435 MHz	2.60 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2400 MHz	7.69 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2300 MHz	2600 MHz	13.04 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2400 MHz	2253 MHz	6.12 %



Chapter 6

Discussion

This chapter interprets the results from the research and discusses the limitations of the static thread configuration in Nanos++ 0.7a-2013-02-22.

- Section 6.1 addresses the computational complexity and accuracy of the developed scheduling-plugin.
- Section 6.2 analyzes the results from the Histogram application kernel.
- Section 6.3 compares the performance and energy consumption of the developed scheduling-plugin with the results from related work.

6.1 Computational complexity and accuracy of the developed scheduling-plugin

This section discusses the computational complexity and accuracy of the developed scheduling-plugin. Table 6.1 lists the time and space complexity of the scheduling-plugin.

Resource	**Complexity**
Time	$O(\text{Cores})$
Space	$O(\text{IPC}_{\text{bins}} * \text{LLCMPC}_{\text{bins}} * \text{Cores})$

Table 6.1: Computational complexity of the developed scheduling-plugin

The intelligent agent considers how many cores are active before it applies DVFS. The time complexity of the solution scales linearly with the number of cores the lookup table must take into account. Linear scalability can be an issue if the solution should be extended for distributed environments. However, the space complexity can become the primary limitation as the number of cores on a multicore increases. One solution to this problem can be to only train each core configuration that is a multiple of n, and interpolate the values that are between two layers in the lookup table.

Table 6.2 lists the average error rate for how much the predicted frequencies differs from the frequencies which have been estimated to be the most energy efficient for the various metrics that have been tested in this study. The error rates are lower for the application kernels that have been used to train the lookup tables, than what they are for the unknown kernels. In addition, if the metric allows for a large variety of frequencies it often leads to higher error rate. The metric $\frac{\text{Operations}}{\text{Joule}}$ contains frequencies in the range 2600 MHz to 1200 MHz, while the Energy Delay Product includes frequencies in the range 2600 MHz to 2000 MHz.

Metric	Known kernel average error	Unknown kernel average error	Total average error
Operations per Joule	5.20%	6.29%	5.77%
Energy Delay Product	1.34%	1.53%	1.44%
Operations per Joule with 10% limit	2.29%	2.84%	2.58%
Energy Delay Product with 10% limit	1.31%	1.54%	1.43%

Table 6.2: The accuracy of the agent for the various metrics

Figure 6.1 provides an overview of how the average accuracy varies with the number of running threads. A large variety of frequencies will often lead to higher error rate. The dynamic power dissipation increases with the number of active cores. The range of energy efficient frequencies will increase as the dynamic power dissipation starts to dominate the static power dissipation.

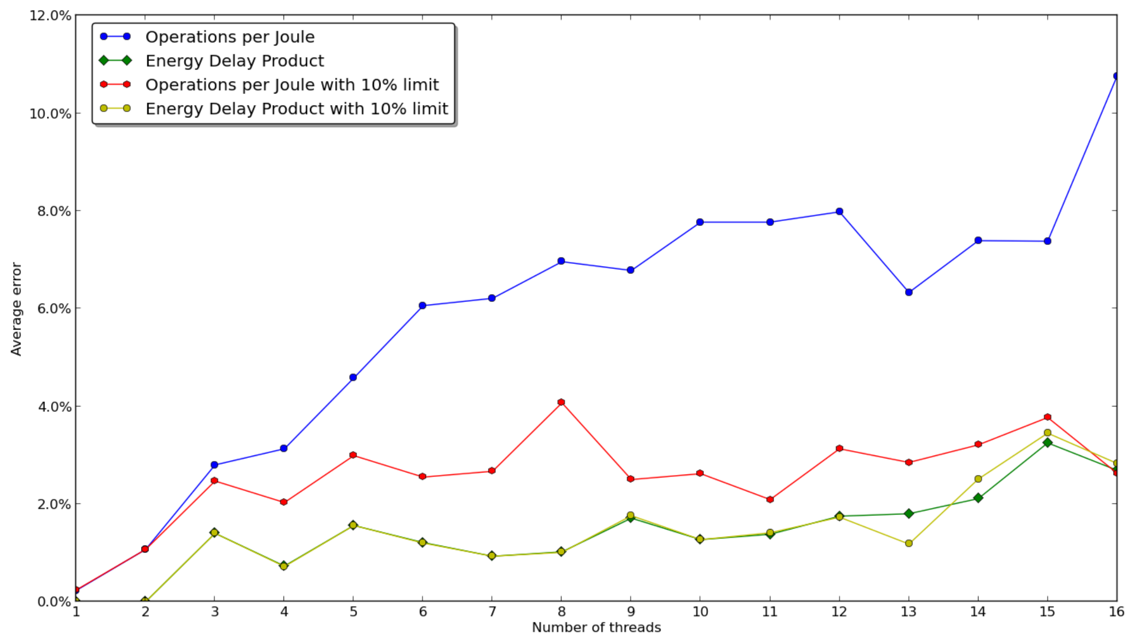


Figure 6.1: The accuracy of the agent decreases as the number of active cores increases

6.2 Analysis of the results from the Histogram application kernel

The results from the Histogram kernel stand out from the rest of the application kernels. Not only is the error rate high, but the developed scheduling-plugin is able to increase the performance over DBF that operates at the maximum frequency. This section interprets the results and provide explanations for the observations.

The results presented in Chapter 5 shows that the Histogram kernel has an error rate of 40% for the metric $\frac{\text{Operations}}{\text{Joule}}$ with 16 active threads. The reason why the error rate is so high is because the optimal frequency is found with an exhaustive search which only applies one frequency throughout the execution of an application. However, the Histogram kernel consists of both parallel and sequential phases. Figure 6.2 illustrates how the agent adjusts the frequency through the execution of the Histogram application kernel. As can be seen from the figure, the agent is capable of adjusting the frequency based on whether the current phase is parallel or sequential. This behavior increases the error rate although the the energy efficiency is improved.

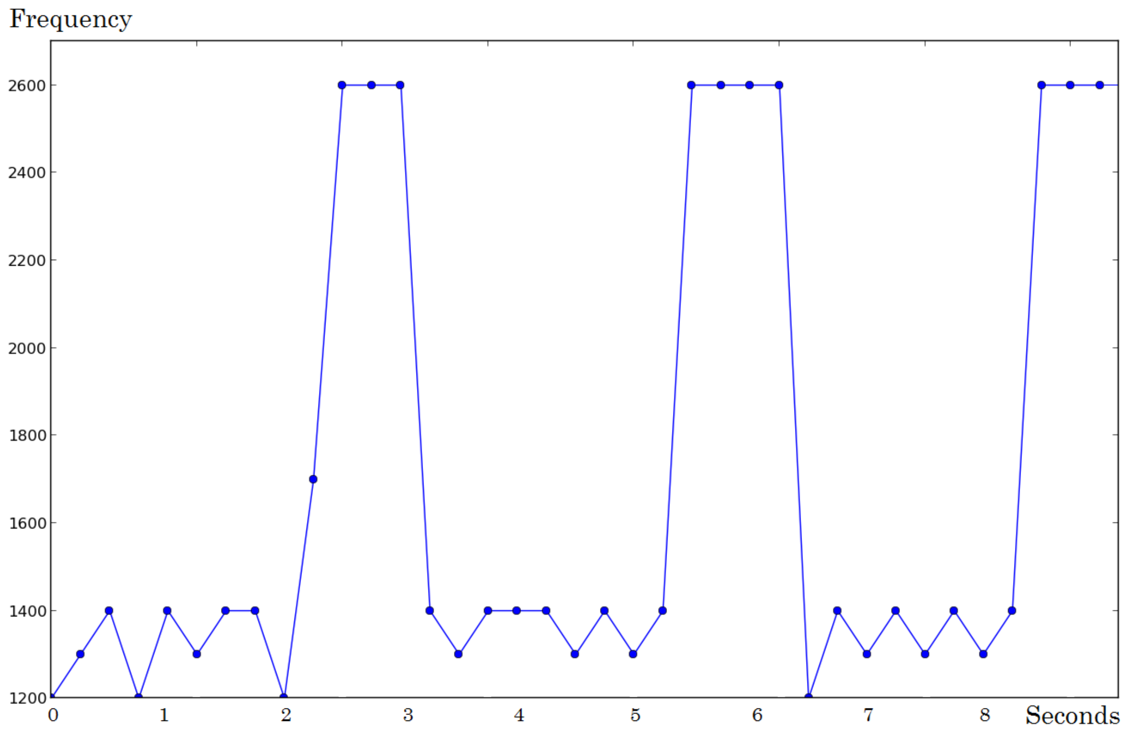


Figure 6.2: The agent is able to identify the parallel and sequential phases in the Histogram application kernel

The developed scheduling-plugin is able to increase the performance over DBF that operates at the maximum frequency for certain results. The performance does not increase for Unstructured 3d stencil, although it has the same computational pattern as Histogram. The behavior of these applications have been analyzed with performance counters in order to determine how the developed scheduling-plugin improves the performance. Table 6.3 lists the IPC and LLCMPC for Histogram and Unstructured 3d stencil for the metric $\frac{\text{Operations}}{\text{Joule}}$, under the constraint that performance cannot suffer by more than 10% with 16 active threads.

Scheduling-plugin	Application kernel	IPC	LLCMPC
Developed	Unstructured 3d stencil	0.3662	0.0035
DBF	Unstructured 3d stencil	0.3264	0.0032
Developed	Histogram	0.4126	0.0057
DBF	Histogram	0.2237	0.0055

Table 6.3: Analysis of the behavior of Histogram and Unstructured 3d stencil

The analysis indicates that Histogram has higher memory intensity than Unstructured 3d stencil. In addition, it shows that the IPC of DBF is approximately halved compared to the IPC of the developed scheduling-plugin for the Histogram application kernel. The resource-related stalls have been analyzed in order to identify why the IPC decreases at the maximum frequency for Histogram and not for Unstructured 3d stencil.

PAPI native event	Description
`RESOURCE_STALLS:ANY`	Cycles stalled due to Resource Related reason
`RESOURCE_STALLS:LB`	Cycles stalled due to lack of load buffers
`RESOURCE_STALLS:RS`	Cycles stalled due to no eligible reservation station entry available

Table 6.4: Performance counters used to analyze resource-related stalls

Figure 6.3 illustrates the distribution of resource-related stalls for Histogram and Unstructured 3d stencil. The measured stall cycles from the performance counters have been divided by the total number of cycles needed to complete the application kernel.

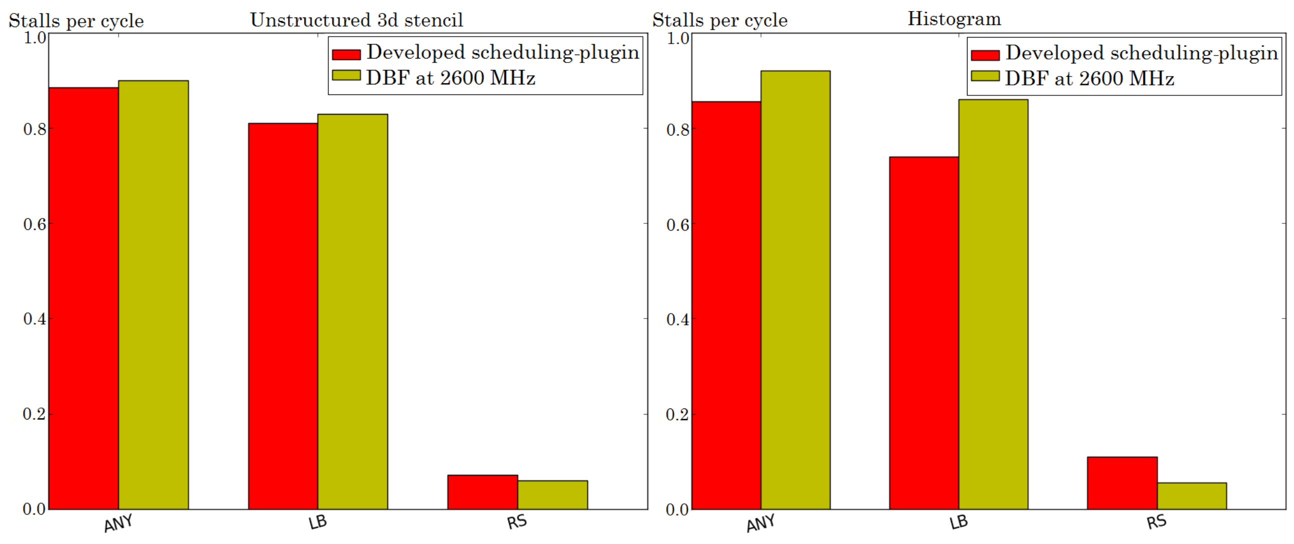
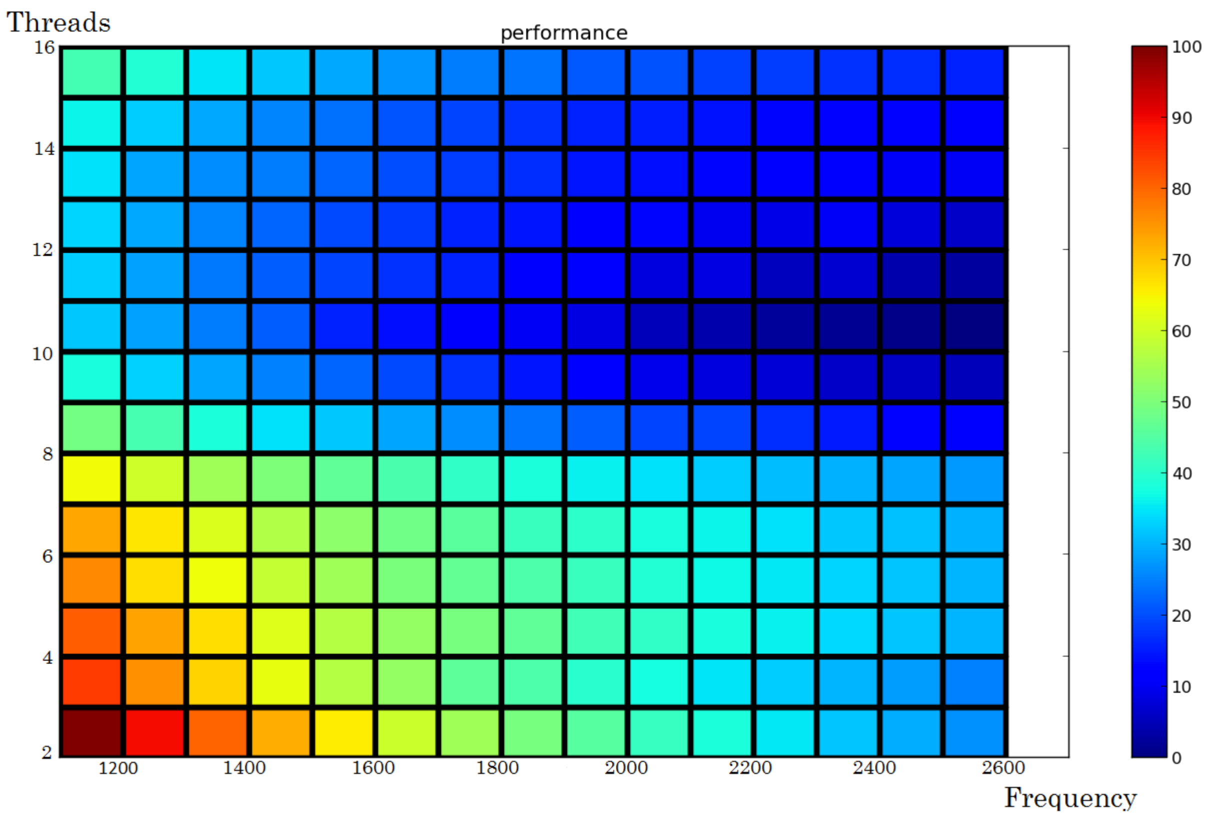


Figure 6.3: Resource-related stalls for each application kernel

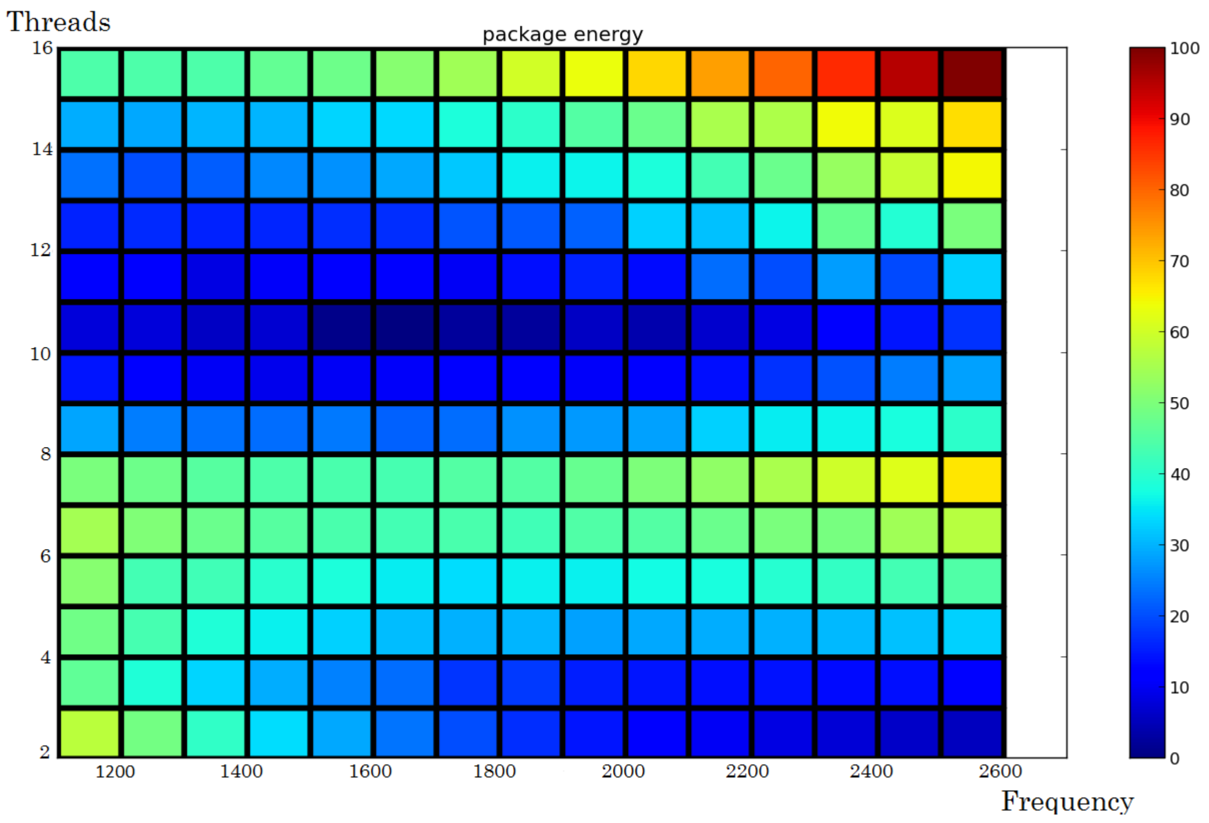
Intel® Xeon® CPU E5-2670 can achieve higher IPC than the number of stalls indicates because it is a superscalar processor which is capable of executing multiple instructions every clock cycle. However, the measurements can still provide valuable insights needed to interpret the observations. Measurements from the performance counters indicate that the processor must stall if there are insufficient available load buffers or eligible reservation station entries. High-performance out-of-order processors can decide to speculatively execute certain loads and stores out of order, and later determine if the loads and stores were correctly executed. Lowering the frequency can reduce the number of speculations because the processor will spend less time waiting for outstanding memory requests. The increase in performance can be explained if lowering the frequency reduces the amount of speculative memory requests occupying the load buffer.

The article *'Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs'* [25] illustrates how the performance of the Histogram application kernel can be improved by dynamic control of the number of running threads. The Histogram kernel divides the total work across *n* tasks, where each task is assigned to calculate a local histogram of size $\frac{\text{total work}}{\text{n}}$. The local histograms will ultimately be merged into a global histogram in a sequential task. Functions such as **omp_get_num_threads()** are often used when an application should be parallelized. For OmpSs applications **omp_get_num_threads()** will return the number of threads that are available in the runtime system. Additional threads beyond the optimal thread configuration can decrease the performance of an application if it is limited by contention for shared resources. Programmers can define the number of tasks explicitly instead of using functions like **omp_get_num_threads()**, however, the programmability and portability will be reduced. An alternative strategy for limiting contention for shared resources could be to disable cores from the Linux kernel, however, such an approach can lead to increased cache pollution since the threads would still share the available cores.

Figure 6.4 illustrates that the optimal thread configuration for the Histogram kernel consists of 11 worker threads. The developed scheduling-plugin reduces the contention for shared resources by lowering the frequency when the optimal thread configuration is exceeded. It would be desirable if the runtime system supported functionality to transfer every tasks from one thread to another. A thread can safely be suspended if it does not hold any locks. However, such functionality was not supported by Nanos++ as mentioned in Chapter 3. Even if it were possible to transfer every task from one thread to another, the runtime system will need to ensure that the OpenMP specification is not violated. The OpenMP specification requires that *tied* tasks are executed by one thread in order to ensure that certain functions will remain thread-safe.



(a)



(b)

Figure 6.4: Overview of performance and energy consumption for the Histogram kernel. The results are only for 3-16 threads since these configurations provide the clearest color map. a) = performance and b) = energy consumption

6.3 Comparison with results from related work

This section compares the results with the previous findings from *'Process Cruise Control'* and *'Green Governors: A Framework for Continuously Adaptive DVFS'*. Table 6.5 lists information about the hardware and how the energy has been measured for the various solutions.

Solution	Level	System	Energy measurement
Developed scheduling-plugin	Thread	Sandy Bridge-EP	Package
Process Cruise Control	Process	XScale 80200	Power Supply
Green Governors	Process	Core i7	Motherboard

Table 6.5: Hardware and energy measurements for the different solutions

The previous solutions have been restricted to only consider the process level, since they have been implemented in the Linux scheduler. The developed scheduling-plugin is able to read the performance counters from each thread since it is part of Nanos++. Decisions made at the thread level can further improve the energy efficiency for systems where the frequency of the cores can be changed separately.

The developed scheduling-plugin has been evaluated on a dual processor Sandy Bridge-EP system, and the energy measurements were made from the processor package. The fact that Sandy Bridge-EP is designed for high-performance limits the amount of energy that can be saved by lowering the frequency. The memory hierarchy of the system is optimized for performance and the static power dissipation is relatively high compared to the dynamic power dissipation. Due to these conditions only Histogram and Unstructured 3d stencil saves a significant amount of energy as illustrated in Figure 6.5.

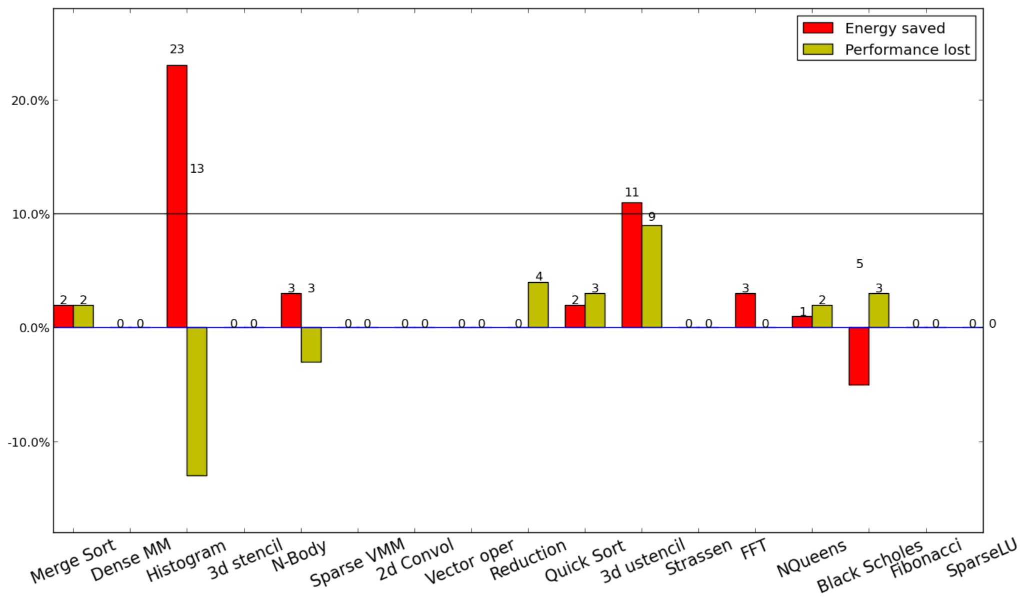


Figure 6.5: Results for the metric Energy Delay Product with 10% limit when 16 threads are running

The energy-aware scheduler from the paper *'Process Cruise Control'* was tested on an Intel XScale 80200 processor with the IQ80310 evaluation board. The energy was measured between the evaluation board and the power supply. The study found that one could save significant amounts of energy for memory intensive applications as illustrated in Figure 6.6. *Memcpy* swaps blocks of memory, *free db* release memory and *fill string* dumps a word database with strcat. The lookup table from *'Process Cruise Control'* was trained on microbenchmarks, while the lookup tables used in this study have been trained to recognize computational patterns found in real applications.

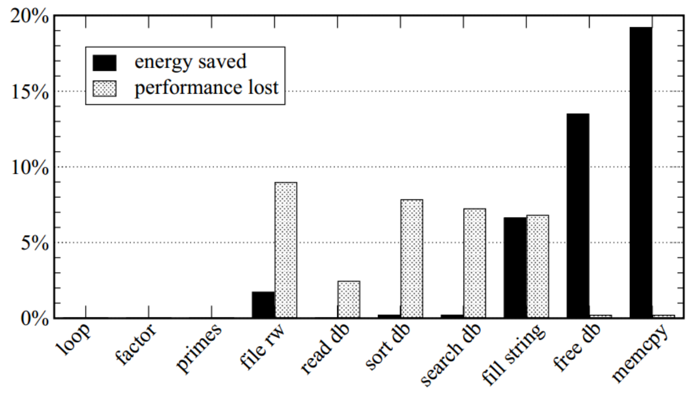


Figure 6.6: Results from *'Process Cruise Control'*[10]

The governors developed in *'Green Governors: A Framework for Continuously Adaptive DVFS'* were tested on a Core i7 and the energy measurements were done from the motherboard. The governors optimized towards the Energy Delay Product under the constraint that performance cannot suffer by more than 10%. The results presented in Figure 6.7 indicates that multiple applications can improve their energy efficiency if the frequency is dynamically adjusted according to the workload. The application kernels used in *'Green Governors: A Framework for Continuously Adaptive DVFS'* are from the SPEC2006 benchmark suite, however, they have not been classified into computational patterns.

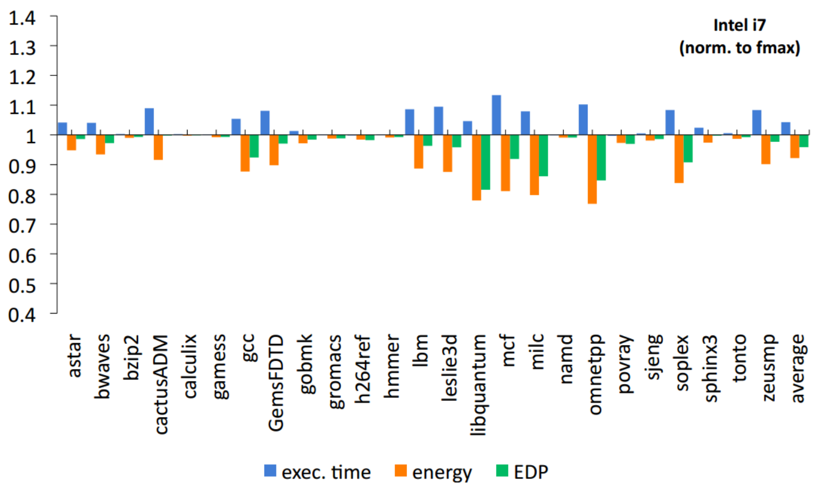


Figure 6.7: Results from *'Green Governors: A Framework for Continuously Adaptive DVFS'*[24]

The Mont-Blanc project will replace the typical Intel Xeon found in supercomputers with energy efficient components from embedded and mobile devices. Results from related work suggest that the developed scheduling-plugin will have greater potential to further improve the energy efficiency if low-power devices such as laptops, mobile components or embedded devices are used as computational platform. In addition, the results indicate that the proper metric to optimize for should be the Energy Delay Product under the constraint that the performance cannot suffer by more than a certain percentage. Although the metric limits the opportunities to save energy, the situations where the energy savings are minimal compared to the performance loss will be avoided.

Chapter 7

Conclusion and Further Work

7.1 Conclusion

This research has examined how energy efficient techniques can be integrated into the Nanos++ runtime system. An energy-aware scheduling-plugin has been proposed, which is able to adjust the frequency on the basis of the different phases in an application by utilizing information from performance counters. The solution proposed in the article *'Process Cruise Control'* has been extended for multi-core systems and implemented in a scheduling-plugin which builds on DBF. The developed scheduling-plugin makes use of a lookup table that models the parameter space for the most energy efficient frequencies according to a particular metric. The table has been trained to identify computational patterns found in common applications. The results show that the developed scheduling-plugin can improve the energy efficiency for applications with computational pattern similar to unstructured grids. Findings from related work suggests that dynamic adjustment of the frequency based on the workload will further improve the energy efficiency when low-power devices are used as computational platform. The fact that the Mont-Blanc project will be based on embedded power-efficient technology indicates that the energy-aware scheduling-plugin will have greater opportunities to improve the energy efficiency in future supercomputers. In the specialization project it was suggested that *'Online monitoring of active threads'* should be examined in order to reduce contention for shared resources. The fact that the thread configuration could not be modified at runtime will be a problem as more cores are added on a multi-core system, because the programmer must be aware of situations where contention for shared resources can occur. Whether an idle thread should busy wait or sleep was also found to be an issue that will escalate as the number of cores increases. The findings from this research suggest that further studies should continue to investigate how energy efficient techniques can be integrated into the Nanos++ runtime system, in order to enhance the energy efficiency while maintaining programmability and portability.

7.2 Further Work

7.2.1 Improve the accuracy of the intelligent agent

In this study, a model that uses IPC, LLCMPC and the number of active cores has been trained to predict the frequency which will be the most energy efficient. Further work could try to improve the model by discovering additional features that are highly correlated with the frequency. In addition, one could consider the advantages and disadvantages of replacing the lookup table with other machine learning algorithms.

7.2.2 Test the intelligent agent on a system where the frequency of the cores can be changed separately

If the frequency of the cores can be modified separately, it is possible to simulate systems with energy constraints. One can decide that the total frequency of the system should not exceed a certain threshold. The intelligent agent is then responsible for distributing the frequency in order to maximize energy efficiency. Priority of tasks can influence how the agent distributes the frequency. An example of a processor that supports separate adjustment of the frequency for each core is AMD Phenom II.

7.2.3 Determine whether Intel® Turbo Boost Technology is energy efficient

Intel® Turbo Boost Technology is a technology developed by Intel that dynamically increases the frequency based on the number of active cores, estimated current consumption, estimated power consumption and the processor temperature. Turbo boost can overclock the processor to higher frequencies than can be set from the userspace governor. If one can determine whether Intel® Turbo Boost Technology is energy efficient, it could be considered if it is possible to shift the governor when the number of cores are below a certain threshold in order to enable turbo boost.

7.2.4 Investigate the pros and cons between busy waiting and putting an idle thread to sleep

Further work could investigate the pros and cons between busy waiting and putting an idle thread to sleep. An alternative method for dealing with idle threads is to use conditional variables and signaling, however, it is unclear whether this is more energy efficient than combining busy waiting with sleep. There is a certain amount of overhead for a core to enter a low-power mode, therefore, it should be investigated how the energy efficiency is influenced by the length of the sleep interval. In addition, one could consider the possibility of adjusting these parameters dynamically at runtime based on the characteristics of the running application.

7.2.5 Implementation of an asymmetric aware intelligent agent

Nanos++ ensures that every worker thread has unique affinities in order to minimize thread migration. When the cores are homogeneous, this is an effective strategy since it helps to preserve the data in caches. However, for asymmetric multi-cores it is uncertain whether it is better to migrate threads to stronger cores than preserving the cache data.

If a task has been scheduled to a weak core, it may be energy efficient to migrate the thread to a stronger core in order to increase the performance. It is uncertain whether the operating system is able to detect this situation, if not, it can be implemented an asymmetry-aware intelligent agent in Nanos++ that can adjust the affinites at runtime. Figure 7.1 gives a conceptual overview of the asymmetry-aware intelligent agent. The adjustment can be as simple as swapping the affinity between two worker threads. The big.LITTLE system from ARM [40] is an example of an asymmetric multi-core. It is specified that one can influence which cores are active based on DVFS. The big core can operate at high frequencies, however the small core is more energy efficient if the workload is low. It would be interesting to examine how the use of thread affinites and userspace governor act in such a system.

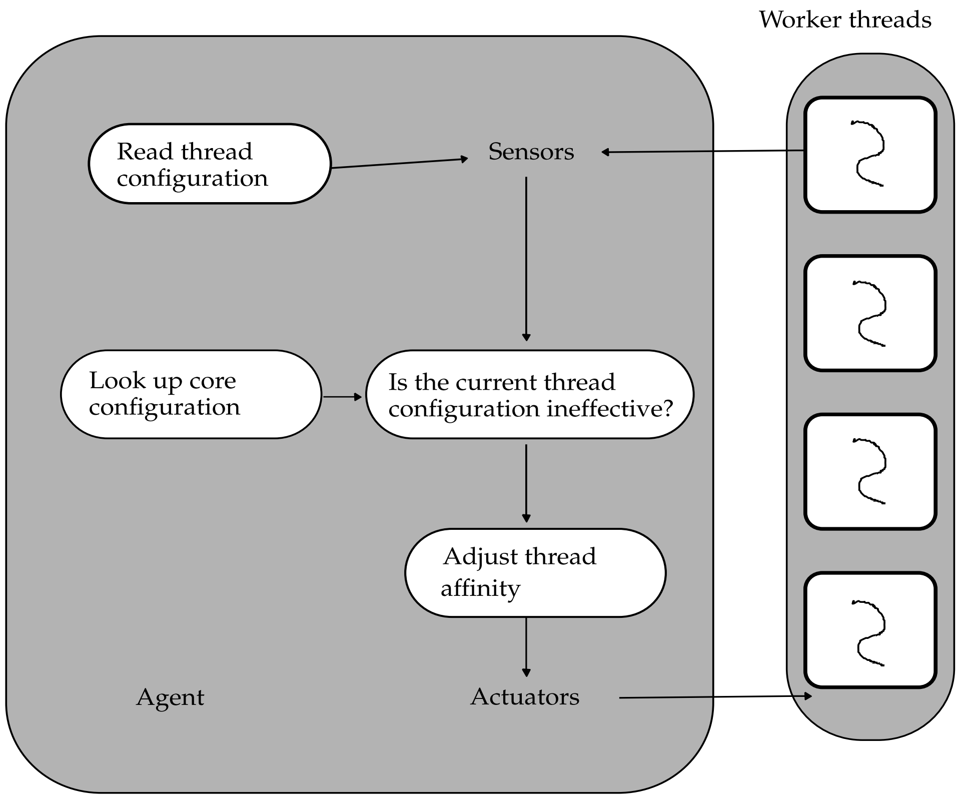

Worker threads
Read thread configuration
Sensors
Look up core configuration
Is the current thread configuration ineffective?
Adjust thread affinity
Agent
Actuators

Figure 7.1: Overview of the Asymmetric Agent

7.2.6 Standardize benchmark suite for task-based programming

A problem that often arises in research projects is shortage of benchmarks representing real-world applications. Further work could standardize an OmpSs benchmark suite based on the computational patterns found in real-world applications, this effort will help increasing the efficiency and quality of research oriented around task-based programming. In this study 17 application kernels have been used where 9 of 13 computational patterns have been covered. Ideally, every computational pattern should be covered, and one should have access to two or more application kernels which are representative for each pattern. The input to each application kernel should be standardized in relation to the degree of parallelism.

References

[1] Partnership for advanced computing in Europe, "PRACE Research Infrastructure." http://www.prace-project.eu/. [cited at p. i]

[2] S. Borkar, "The Exascale challenge," *VLSI Design Automation and Test, 2010 International*, pp. 2 – 3, April 2010. [cited at p. 1]

[3] Mont-Blanc project, "Mont-Blanc project." http://www.montblanc-project.eu/. [cited at p. 1]

[4] Barcelona Supercomputing Center, "BSC-CNS." http://www.bsc.es/. [cited at p. 1]

[5] Hallgeir Lien, "Case Studies in Multi-core Energy Efficiency of Task Based Programs." [cited at p. 1]

[6] Bjørn Fevang, "Initial experiments towards an energy efficient task pool implementation for OmpSs." [cited at p. 1]

[7] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah, "Analyzing the Energy Efficiency of a Database Server," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 231 –242, 2010. [cited at p. 1]

[8] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *Micro, IEEE*, pp. 122 –134, 2012. [cited at p. 1]

[9] T. B. Martinsen, "Towards an energy efficient task pool implementation for OmpSs," 2012. [cited at p. 2, 16, 21]

[10] A. Weissel and F. Bellosa, "Process cruise control," *International conference on Compilers, architecture, and synthesis for embedded systems,CASES*, pp. 238–246, 2002. [cited at p. 2, 18, 24, 32, 68]

[11] S. Shahrivari and M. Sharifi, "Task-oriented Programming: A Suitable Programming Model for Multicore and Distributed Systems," *Parallel and Distributed Computing, ISPDC*, pp. 139 –144, July 2011. [cited at p. 5]

[12] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, "A high-productivity task-based programming model for clusters," *Concurrency and Computation: Practice & Experience*, vol. 24, pp. 2421 – 2448, december 2012. [cited at p. 5]

[13] Rosa M. Badia, "Programming with StarSs." http://www.bsc.es/sites/default/files/public/mare_nostrum/hpc-events/5039.pdf. [cited at p. 7]

[14] OpenMP Architecture Review Board, "OpenMP Application Program Interface." http://www.openmp.org/mp-documents/spec30.pdf. [cited at p. 7]

[15] J. L. Hennessy and D. A. Petterson, *Computer Architecture, 5th Edition: A Quantitative*. Morgan Kaufmann, 2011. [cited at p. 10]

[16] S. Rivoire, M. A. Shah, P. Ranganathan, C. Kozyrakis, and J. Meza, "Models and Metrics to Enable Energy-Efficiency Optimizations," *Computer, IEEE*, vol. 40, pp. 39 –48, december 2007. [cited at p. 10]

[17] R. Gonzalez and M. Horowitz, "Energy Dissipation in General-Purpose Microprocessors," *Solid-State Circuits, IEEE*, pp. 1277 – 1248, september 1996. [cited at p. 11]

[18] L. Natvig and A. C. Iordan, "Green Computing: Saving Energy by Throttling, Simplicity and Parallelization," *UPGRADE, CEPIS*, vol. 12, pp. 49 –58, october 2011. [cited at p. 11]

[19] D. H. Woo and H.-H. S. Lee, "Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era," *Computer, IEEE*, vol. 41, pp. 24 –31, december 2008. [cited at p. 12]

[20] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 2009. [cited at p. 13]

[21] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, J. Kubiatowicz, N. Morgan, K. Keutzer, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Communications of the ACM - A View of Parallel Computing*, vol. 52, pp. 56–67, 2009. [cited at p. 14]

[22] V. Gupta, H. Kim, and K. Schwan, "Evaluating Scalability of Multi-threaded Applications on a Many-core Platform," *CiteSeerX*, vol. 21, 2012. [cited at p. 16]

[23] L. C. Singleton, C. Poellabauer, and K. Schwan, "Monitoring of Cache Miss Rates for Accurate Dynamic Voltage and Frequency Scaling," *Multimedia Computing and Networking, SPIE*, vol. 5680, 2005-01-17 2005. [cited at p. 17, 24, 32]

[24] V. Spiliopoulos, S. Kaxiras, and G. Keramidas, "Green Governors: A Framework for Continuously Adaptive DVFS," *Green Computing Conference and Workshops, IGCC*, 2011. [cited at p. 19, 24, 32, 68]

[25] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-threaded Workloads on CMPs," in *Proceedings of the 13th int'l conf. on architectural support for programming languages and operating systems*, ASPLOS, 2008. [cited at p. 21, 65]

[26] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via OS level monitoring," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, 2011. [cited at p. 21]

[27] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade, "Decomposable and Responsive Power Models for Multicore Processors using Performance Counters," *International Conference on Supercomputing, ICS*, pp. 147–158, 2010. [cited at p. 24]

[28] Intel, "Intel® XScale™ Microarchitecture for the PXA255 Processor User's Manual." http://int.xscale-freak.com/XSDoc/PXA255/27879601.pdf. [cited at p. 24]

[29] Intel, "Intel® Xeon® Processor E5-2600 Product Family Uncore Performance Monitoring Guide." http://www.intel.com/content/dam/www/public/us/en/documents/design-guides/xeon-e5-2600-uncore-guide.pdf. [cited at p. 26]

[30] Intel, "Intel® Turbo Boost Technology — On-Demand Processor Performance." http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html. [cited at p. 26]

[31] N. S. Kim, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage Current: Moore's Law Meets Static Power," *Computer, IEEE*, vol. 36, pp. 68–75, 2003. [cited at p. 27]

[32] UPC, "The High Performance Computing Group from Universitat Politècnica de Catalunya." http://hpc.ac.upc.edu/doku/doku.php. [cited at p. 33]

[33] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadè, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP, 2009. [cited at p. 34]

[34] Notur, "VILJE - the new supercomputer at NTNU." http://www.notur.no/publications/magazine/pdf/meta_2011_4.pdf. [cited at p. 37]

[35] Intel, "Intel® Xeon® Processor E5-2670." http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI. [cited at p. 38]

[36] J. M. Cebrián, "Temperature effects on power measurements for the Blackscholes, FFTW and Matrix Multiplication benchmarks," 2012. [cited at p. 41]

[37] Intel, "Intel Architecture Developer's Manual Volume 3B, Appendix A." [cited at p. 48]

[38] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B." http://download.intel.com/products/processor/manual/253669.pdf. [cited at p. 48]

[39] J. M. Cebrián, L. Natvig, and J. C. Meyer, "Improving energy efficiency through parallelization and vectorization on Intel® Core I5 and I7 processors," *High Performance, Networking, Storage and Analysis(SCC), 2012 SC Companion*, pp. 675–684, 2012. [cited at p. 49]

[40] ARM, "Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7." http://www.arm.com/files/downloads/big.LITTLE_Final.pdf. [cited at p. 73]

Appendices

Source Files - Scheduling-plugin

This appendix contains the source code of the developed scheduling-plugin. The way Nanos++ cleans up its resources will have to be modified if the scheduling-plugin should be used in a production environment. One can experience a segmentation fault if the runtime system starts to delete its resources before the agent has been stopped. This problem can easily be resolved by modifying the code for how Nanos++ deletes the resources. However, the segmentation fault does not affect the results of the experiments, since it only occurs when the job is finished. The source code of the scheduling-plugin is the only part that has been modified in order to make the implementation independent of the runtime system. In this way, the solution can easily be tested by others by simply compiling the scheduling-plugin instead of the entire runtime system.

```
#include <pthread.h>
#include <papi.h>
#include <iostream>
#include <fstream>

#include "schedule.hpp"
#include "wddeque.hpp"
#include "plugin.hpp"
#include "system.hpp"

/** The number of available frequencies **/
#define FREQUENCIES 15

/**
* Number of performance counters used by
* the scheduling-plugin
*/
#define PERFORMANCE_COUNTERS 3

/**
* The length of the interval the agents awaits before it tests
* if Nanos++ is initialized.
* The time is specified in microseconds
*/
```

```
#define INIT_TIME 100000

/**
* The length of the interval the agents sleeps before it gather
* data from the performance counters.
* The time is specified in microseconds
*/
#define IDLE_TIME 250000

/** Macro for indexing the lookup table **/
#define INDEX(i, j, k, binLLCMPC, binIPC)
      {i*binLLCMPC*binIPC)+(j*binIPC)+k}

/** Macro for reporting error **/
#define ERROR_RETURN(retval)
      { fprintf(stderr, "Error %d %s:line %d: \n",
        retval,__FILE__,__LINE__); }

/**
* global variable that determines whether
* the agent should run or not
*/
bool running;

namespace nanos {
  namespace ext {

    class DistributedBFPolicy : public SchedulePolicy
    {
      private:
        /** Data associated to each thread **/
        struct ThreadData : public ScheduleThreadData
        {
          /** local queue of ready tasks to be executed **/
          WDDeque _readyQueue;
          /**
              * variable to determine if papi
              * has been initialized
              */
          bool papi_initialized;
          /** papi event set **/
          int EventSet;
          /** pointer to idle task **/
          WD* idleWD;


          ThreadData () : _readyQueue()
          {
             papi_initialized = false;
             EventSet = PAPI_NULL; idleWD = NULL;
          }

          virtual ~ThreadData () {
             long long values[8] = {0};
             PAPI_stop(EventSet, values);
             ensure(_readyQueue.empty(),
                   "Destroying non-empty queue");
          }
       };
       /** pthread for the intelligent agent **/
       pthread_t thread_handle;

       /** disable copy and assigment **/
       explicit DistributedBFPolicy
          ( const DistributedBFPolicy & );
       const DistributedBFPolicy & operator=
          ( const DistributedBFPolicy & );

    public:
       /** constructor **/
       DistributedBFPolicy() :
          SchedulePolicy ( "Cilk" ) { initialize(); }

       /** destructor **/
       virtual ~DistributedBFPolicy() {
              running = false; void *status;
              pthread_join(thread_handle, &status); }

       virtual size_t getTeamDataSize () const { return 0; }
       virtual size_t getThreadDataSize () const {
                         return sizeof(ThreadData); }

       virtual ScheduleTeamData * createTeamData ()
       {
          return 0;
       }




```

```
      virtual ScheduleThreadData * createThreadData()
      {
         return NEW ThreadData();
      }

      /**
         * Queue task
         *
         * @param thread Pointer to the thread to where the
         *         task should be enqueued
         * @param wd Reference to the work
         *         descriptor to be enqueued
         */
      virtual void queue ( BaseThread *thread, WD &wd )
      {
      ThreadData *data;
         if ( wd.isTied() ) {
            data = ( ThreadData * ) wd.isTiedTo()->
                  getTeamData()->getScheduleData();
         } else {
            data = ( ThreadData * ) thread->
                  getTeamData()->getScheduleData();
         }
         data->_readyQueue.push_front ( &wd );
      }
      /**
         * Function called when a new task must be created
         *
         * @param thread Pointer to the thread to where the
         *         task should be enqueued
         * @param wd Reference to the work
         *         descriptor to be enqueued
         */
      virtual WD * atSubmit ( BaseThread *thread, WD &newWD )
      {
         ThreadData &data = ( ThreadData & ) *thread->
                           getTeamData()->getScheduleData();
         queue(thread, newWD);
         return 0;
      }

      virtual WD *atIdle ( BaseThread *thread );
```

```
        /**
         * This function is called when the thread
         * data structure should be initialized
         *
         * @param thread Pointer to the thread that
         *        should be initialized
         * @param data Reference to
         *        data associated with thread
         */
        virtual void init_thread( BaseThread *thread,
                                  ThreadData &data )
        {
            init_papi_thread(data);
            data.papi_initialized = true;
            data.idleWD = thread->getCurrentWD();
        }

        /**
         * This function initializes papi
         */
        virtual void init_papi()
        {
            int retval;

            retval = PAPI_library_init(PAPI_VER_CURRENT);

            if (retval != PAPI_VER_CURRENT) {
               ERROR_RETURN(retval);
            }

            retval = PAPI_thread_init(pthread_self);
            if (retval != PAPI_OK) {
               ERROR_RETURN(retval);
            }

            retval = PAPI_thread_id();
            if ((retval == -1) || (retval == PAPI_EMISC)) {
               ERROR_RETURN(retval);
            }
        }
        /**
         * This function registers which events that should
         * be tracked by performance counters
         *
         * @param EventSet Reference to papi event set
         */
        virtual void add_events_to_eventSet(int& EventSet)
        {
            int events[] = {PAPI_TOT_INS,PAPI_L3_TCM,PAPI_TOT_CYC};
            int nevents = PERFORMANCE_COUNTERS;

            for (int i = 0; i < nevents; i++) {
                int retval;
                /** query whether the event exists **/
                if ((retval = PAPI_query_event(events[i])) != PAPI_OK) {
                    ERROR_RETURN(retval);
                }
                /** add events to the event set **/
                if ((retval = PAPI_add_event(EventSet,events[i]))
                                                       != PAPI_OK) {
                    ERROR_RETURN(retval);
                }
            }
        }

        /**
         * This function initializes papi per thread
         *
         * @param data Reference to
         *         data associated with thread
         */
        virtual void init_papi_thread(ThreadData &data)
        {
            int retval;
            /** create the event set **/
            if ((retval = PAPI_create_eventset(&(data.EventSet)))
                                                       != PAPI_OK) {
                ERROR_RETURN(retval);
            }

            /** add events to the set **/
            add_events_to_eventSet(data.EventSet);



        /** start counting **/
        if((retval = PAPI_start(data.EventSet)) != PAPI_OK) {
            ERROR_RETURN(retval);
        }
    }

    /* This function initializes the intelligent agent */
    virtual void init_IntelligentAgent()
    {
        running = true;
        pthread_create(&thread_handle, NULL,
                        intelligentAgent, (void*)NULL);
        set_affinity(thread_handle, 7);
    }

    /* This function initializes the scheduling-plugin */
    virtual void initialize()
    {
        init_papi();
        init_IntelligentAgent();
        set_cpufreq(convertFrequency(FREQUENCIES-1));
    }





    /**
        * This function sets the processor
        * affinity of a thread
        *
        * @param thread_handle Handler for thread that
        *        should adjust affinity
        * @param cpu Id to the processor where the
        *        thread should be tied
        */
    static void set_affinity(pthread_t thread_handle, int cpu)
    {
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);
        CPU_SET(cpu, &cpuset);
        pthread_setaffinity_np(thread_handle,
                              sizeof(cpu_set_t), &cpuset);
    }
```

```

        /**
           * This function sets the processor frequency
           *
           * @param mHz New processor frequency
           */
        static void set_cpufreq(int mHz)
        {
            if (mHz >= 1200 && mHz <= 2600) {
                std::stringstream ss;
                ss << "sudo cpupower frequency-set -f "
                   << mHz << "MHz > /dev/null";
                system(ss.str().c_str());
            }
        }

        /**
           * This function initializes the lookup table
           *
           * @param binIPC Reference to the number of IPC bins
           * @param binLLCMPC Reference to the number
           *         of LLCMPC bins
           * @param threads Reference to number
           *         of active cores
           * @param binSizeIPC Reference to IPC range
           * @param binSizeLLCMPC Reference to LLCMPC range
           *
           * @return Pointer to the lookup table
           */
        static char* initializeLookupTable(int &binIPC, int &
binLLCMPC,
                int &threads, float &binSizeIPC,
                float &binSizeLLCMPC)
        {
            /** open lookup table **/
            std::ifstream file("table.data");
            if (file.is_open() && file.good()) {
                /** read configuration **/
                file >> binIPC >> binLLCMPC >> threads
                    >> binSizeIPC >> binSizeLLCMPC;
                int* table = new int[threads*binLLCMPC*binIPC];
                /** read data **/
                for (int i = 0; i < threads; i++) {
                    for (int j = 0; j < binLLCMPC; j++) {
                        for (int k = 0; k < binIPC; k++) {
```

```
                        int data;
                        file >> data;
                table[INDEX(i,j,k,binLLCMPC,binIPC)] = (char)data;
                    }
                }
            }
            file.close();
            return table;
        } else {
            return NULL;
        }
    }

    /**
        * This function reads the performance counters and
        * registers how many threads that processes tasks.
        * Data is averaged with the harmonic mean
        *
        * @param threads Reference to number
        * of active cores
        * @param ipc Reference to the processor ipc
        * @param llcmpc Reference to processor llcmpc
        */
    static void readSensors(int &threads, float &ipc, float &
llcmpc)
    {
        threads = 0;
        ipc = 0.0;
        llcmpc = 0.0;

        for (int i = 0; i < sys.getNumWorkers(); i++) {

            long long deltas[PERFORMANCE_COUNTERS];

            BaseThread* thread = sys.getWorker(i);
            ThreadData &data = ( ThreadData & ) *thread->
                            getTeamData()->getScheduleData();

            /** Check if papi has been initialized **/
            if (data.papi_initialized == true) {
                int retval = PAPI_read(data.EventSet, deltas);
                PAPI_reset(data.EventSet);

                /** Check if thread is not idle **/
                if (data.idleWD != thread->getCurrentWD()) {
                        threads++;
                        ipc += 1.0/(((double)deltas[0])
                                    /((double)deltas[2]));
                        llcmpc += 1.0/(((double)deltas[1])
                                    /((double)deltas[2]));
                    }
                }
            }
            if (ipc != 0.0  && llcmpc != 0.0) {
                ipc = threads/ipc;
                llcmpc = threads/llcmpc;
            }
        }

        /** The agent will wait for papi to be initialized **/
        static void waitForPapiToInitialize()
        {
            bool papi_initialized = false;
            while (!papi_initialized) {
                usleep(INIT_TIME);
                BaseThread* thread = sys.getWorker(0);
                ThreadData &data = ( ThreadData & ) *thread->
                                getTeamData()->getScheduleData();
                papi_initialized = data.papi_initialized;
            }
        }

        /* The agent will wait until Nanos++ is initialized */
        static void waitForSystemToInitialize()
        {
            while (sys.getNumWorkers() == 0) {
                usleep(INIT_TIME);
            }
        }

        /**
            * This function converts index to frequnecy
            *
            * @param index Internal representation of frequency
            *
            * @return Converted frequency
            */
        static int convertFrequency(int index) {
            return (index * 100) + 1200;
        }
```

```




    /**
       * This function retrieves the frequency
       * from the lookup table
       *
       * @param binSizeLLCMPC LLCMPC range
       * @param binSizeIPC IPC range
       * @param LLCMPC Observed llcmpc
       * @param IPC Observed IPC
       * @param binIPC Number of IPC bins
       * @param binLLCMPC Number of LLCMPC bins
       * @param thread Number of active cores
       * @param table Pointer to the lookup table
       *
       * @return Frequency
       */
    static int lookupFrequency(float binSizeLLCMPC,
       float binSizeIPC, float LLCMPC, float IPC,
       int binIPC, int binLLCMPC, int thread, int* table) {

       int indexIPC = std::min((int)(IPC/binSizeIPC),
                            binIPC-1);
       int indexLLCMPC = std::min((int)(LLCMPC/binSizeLLCMPC),
                            binLLCMPC-1);

       return (int)table[INDEX(thread, indexLLCMPC,
                   indexIPC, binLLCMPC, binIPC)];
    }

    static void * intelligentAgent(void * arg)
    {
       int binIPC = 0, binLLCMPC = 0, threads = 0;
       float binSizeIPC = 0.0, binSizeLLCMPC = 0.0;
       char* table = initializeLookupTable(binIPC, binLLCMPC,
                  threads, binSizeIPC, binSizeLLCMPC);

       if (table != NULL) {
          int currentFrequency = 2600;
          waitForSystemToInitialize();
          waitForPapiToInitialize();

          while (true == running) {
```

```

        usleep(IDLE_TIME);

                    int threads = 0;
                    float ipc = 0.0;
                    float llcmpc = 0.0;

                    /**
                          * read performance counters
                          * and active cores
                          */
                    readSensors(threads, ipc, llcmpc);

                    int frequency = currentFrequency;
                    if (threads != 0) {
                       /** lookup new frequency **/
                       frequency = convertFrequency(
                                lookupFrequency(binSizeLLCMPC,
                                binSizeIPC, llcmpc, ipc, binIPC,
                                binLLCMPC, threads−1, table));
                    }

                    /** should the frequency be adjusted? **/
                    if (currentFrequency != frequency) {
                       set_cpufreq(frequency);
                       currentFrequency = frequency;
                    }
                 }

                 delete [] table;
              }
        }
    };

    /**
      * This function will be called when
      * a new task should be picked
      *
      * @param thread Pointer to the thread that
      *        should pick new task
      *
      * @return Pointer to new task
      */
    WD * DistributedBFPolicy::atIdle ( BaseThread *thread )
    {
        WorkDescriptor * wd;
        WorkDescriptor * next = NULL;

        ThreadData &data = ( ThreadData & ) *thread->
                        getTeamData()->getScheduleData();

        if (data.papi_initialized == false) {
          init_thread(thread, data);
        }

        /**
           * First try to schedule the thread
           * with a task from its queue
           */
        if ( ( wd = data._readyQueue.pop_front ( thread ) ) != NULL ) {
            return wd;
        } else {
            /**
                * If the local queue is empty,
                * try to steal the parent
                * (possibly enqueued in the
                *  queue of another thread)
                */
            if ( ( wd = thread->getCurrentWD()->getParent() ) != NULL ) {
                /**
                     * Try to remove from one queue:
                     * if someone move it, I stop looking
                     * for it to avoid ping-pongs
                     */
                if ( wd->isEnqueued() ) {
                    if ( wd->getMyQueue()->removeWD( thread, wd, &next ) )
    {
                        return next;
                    }
                }
            }

            /**
                * If also the parent is NULL or if someone moved
                * it to another queue while was trying to steal
                * it, try to steal tasks from other queues
                */
            int thid = thread->getTeamId();
            int size = thread->getTeam()->size();
            WorkDescriptor * wd = NULL;

            do {
                thid = ( thid + 1 ) % size;

                BaseThread &victim = thread->
                                    getTeam()->getThread(thid);

                if ( victim.getTeam() != NULL ) {
                    ThreadData &tdata = ( ThreadData & )
                       *victim.getTeamData()->getScheduleData();
                    wd = tdata._readyQueue.pop_back ( thread );
                }

            } while ( wd == NULL && thid != thread->getTeamId() );

            return wd;
        }
    }

    class DistributedBFSchedPlugin : public Plugin
    {
       public:
          DistributedBFSchedPlugin()
          : Plugin( "Distributed Breadth-First scheduling Plugin",1
  ) {}

          virtual void config( Config& cfg ) {}

          virtual void init() {
             sys.setDefaultSchedulePolicy(NEW DistributedBFPolicy());
          }
    };

  }
}

nanos::ext::DistributedBFSchedPlugin NanosXPlugin;
```

Experiment configurations

This appendix provides an overview of the experiment configurations. Listing .1 presents the command that has been used to run an experiment.

Listing .1 Command to run an experiment

```
OMP_NUM_THREADS=threads
NX_ARGS="--schedule agent --throttle taskdepth
                  --throttle-limit 4" ./application
```

FFT, NQueens, Vector operation and Strassen have been configured with the additional command *'--spins 1 --sleep-time 90000000'* added in NX_ARGS. This command ensures that an idle thread will sleep for 9ms before it tries to steal a new task. Tables 1, 2 and 3 list the experiment configurations for each of the application kernels that have been used during the research.

Application kernel	Threads	Iterations	Parameters
Merge Sort	1 - 7	1	N = 1 GB, CUT_OFF = 64 KB
Merge Sort	8 - 16	1	N = 2 GB, CUT_OFF = 64 KB
Reduction	1 - 8	20	N = 1 GB, BSIZE = N / 1 KB
Reduction	9 - 16	40	N = 1 GB, BSIZE = N / 1 KB
Histogram	1 - 16	3	N = 1.5 GB
			NUM_LOCAL_HISTOGRAM = 64
			BSIZE = N / NUM_LOCAL_HISTOGRAM
			HISTOGRAM_MAX = 1 MB

Table 1: Experiment configurations

Application kernel	Threads	Iterations	Parameters

2d convolution	1 - 6	1	IMAGE_WIDTH = 2 KB, NUM_TASKS = 32
			IMAGE_HEIGHT = IMAGE_WIDTH
			FILTER_WIDTH = 32
			FILTER_HEIGHT = 32
			INPUT_IMAGE_WIDTH =
			IMAGE_WIDTH + FILTER_WIDTH
			INPUT_IMAGE_HEIGHT =
			IMAGE_HEIGHT + FILTER_HEIGHT

2d convolution	7 - 16	1	IMAGE_WIDTH = 4 KB, NUM_TASKS = 32
			IMAGE_HEIGHT = IMAGE_WIDTH
			FILTER_WIDTH = 32
			FILTER_HEIGHT = 32
			INPUT_IMAGE_WIDTH =
			IMAGE_WIDTH + FILTER_WIDTH
			INPUT_IMAGE_HEIGHT =
			IMAGE_HEIGHT + FILTER_HEIGHT

3d stencil	1 - 7	5	Nx = 1 KB, Ny = 1 KB, Nz = 1KB
			NUM_TASKS = 32

3d stencil	8 - 16	10	Nx = 1 KB, Ny = 1 KB, Nz = 1KB
			NUM_TASKS = 32

Dense matrix multiplication	1 - 8	1	DIM = 16, BSIZE = 512
Dense matrix multiplication	9 - 16	1	DIM = 20, BSIZE = 512
Sparse matrix vector multiplication	1 - 8	4000	input = bcsstk32.mtx
Sparse matrix vector multiplication	9 - 16	8000	input = bcsstk32.mtx

Vector operation	1 - 8	25	N = 1 GB, BSIZE = N / 128
			epsilon = 1.e-8f, T = float

Vector operation	9 - 16	50	N = 1 GB, BSIZE = N / 128
			epsilon = 1.e-8f, T = float

N-body	1 - 8	1	N = 49152, NUM_TASKS = 32
			FLOAT_TYPE = float
			dT = 0.001f, damping = 0.995f
			softeningSquared = 0.00125f

N-body	9 - 16	1	N = 65536, NUM_TASKS = 32
			FLOAT_TYPE = float
			dT = 0.001f, damping = 0.995f
			softeningSquared = 0.00125f

Table 2: Experiment configurations

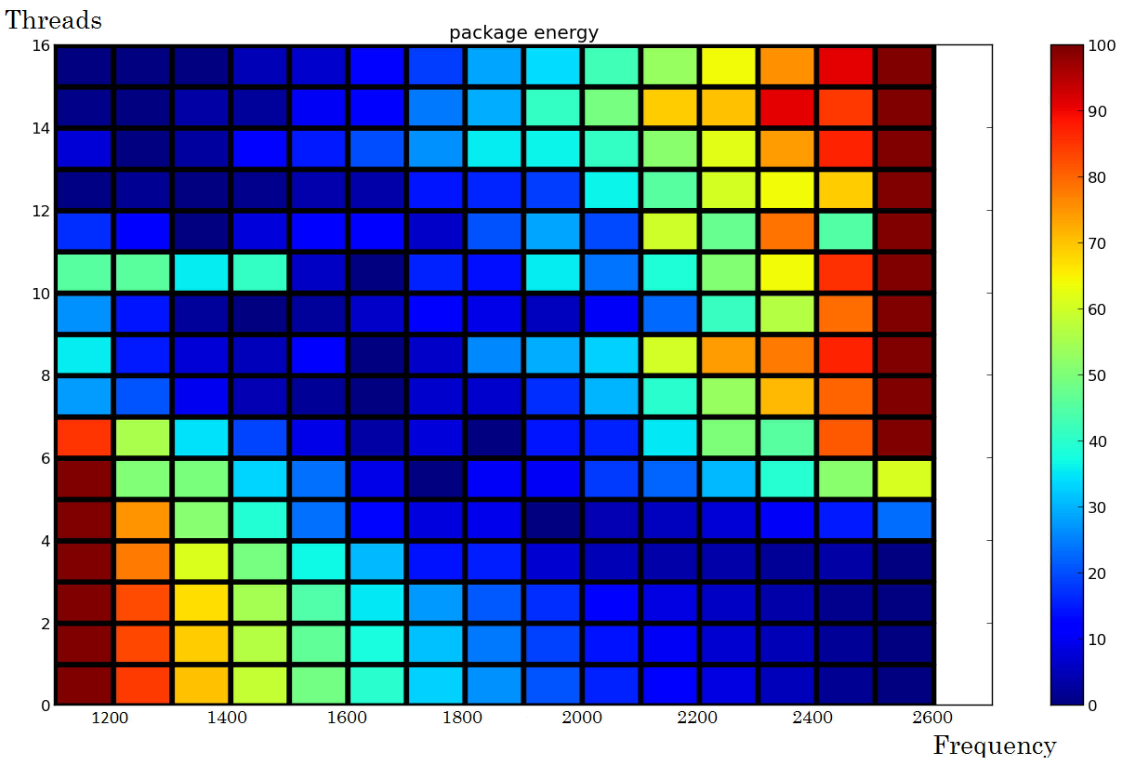
Application kernel	**Threads**	**Iterations**	**Parameters**
FFT	1 - 5	1	N = 64 MB
FFT	6 - 16	1	N = 256 MB
SparseLU	1 - 3	1	size = 35, size_1 = 100
SparseLU	4 - 6	2	size = 35, size_1 = 100
SparseLU	7 - 9	4	size = 35, size_1 = 100
SparseLU	10 - 12	8	size = 35, size_1 = 100
SparseLU	13 - 16	10	size = 35, size_1 = 100
NQueens	1 - 3	1	N = 13
NQueens	4 - 8	1	N = 14
NQueens	9 - 16	1	N = 15
Strassen	1 - 16	1	N = 4096
Black Scholes	1 - 5	5000	N = 64 KB
Black Scholes	6 - 16	20000	N = 64 KB
Fibonacci	1 - 8	1	N = 48
Fibonacci	9 - 16	1	N = 49
Quick Sort	1 - 7	1	N = 1 GB, CUT_OFF = 64 KB
Quick Sort	8 - 16	1	N = 2 GB, CUT_OFF = 64 KB
Unstructured 3d stencil	1 - 8	4	Nx = 512, Ny = 512, Nz = 512 NUM_TASKS = 32
Unstructured 3d stencil	9 - 16	8	Nx = 512, Ny = 512, Nz = 512 NUM_TASKS = 32

Table 3: Experiment configurations

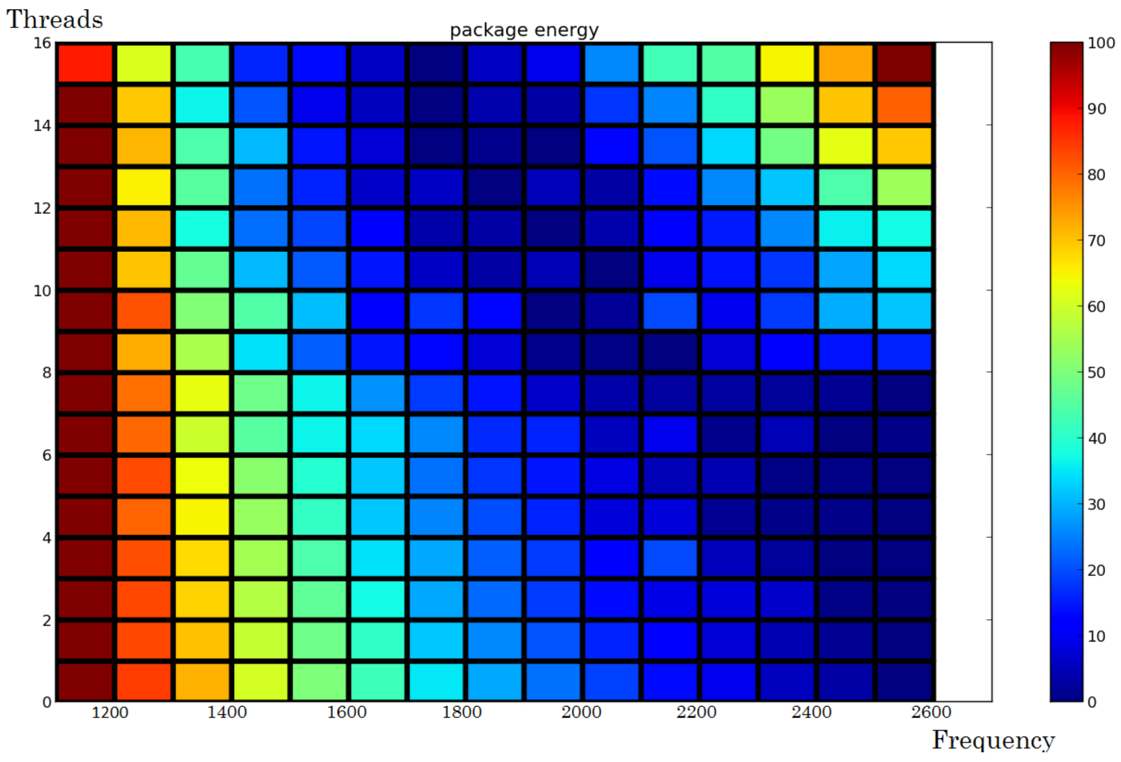
Training Phase - Application kernels

This appendix presents the matrices that have been generated during the training of the lookup table. Each row represents a thread configuration, and each column a frequency. The energy consumption is normalized, so the frequency with the lowest value is the most energy efficient.

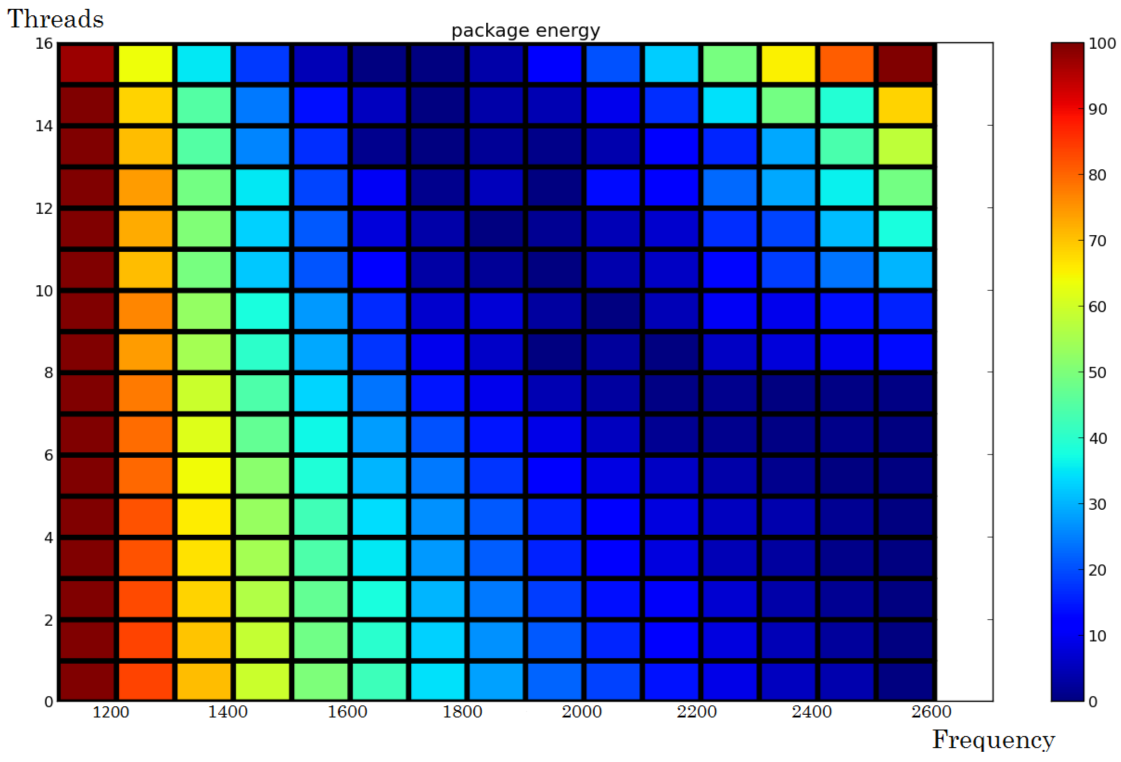
Histogram



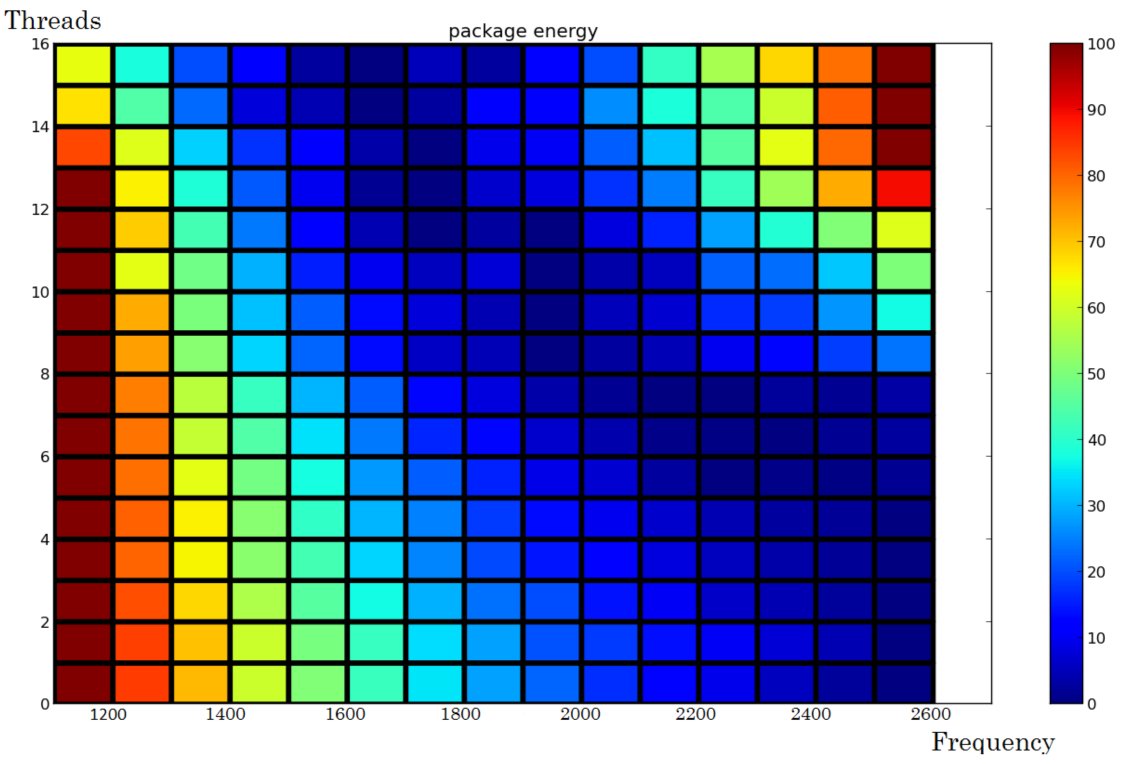
NQueens



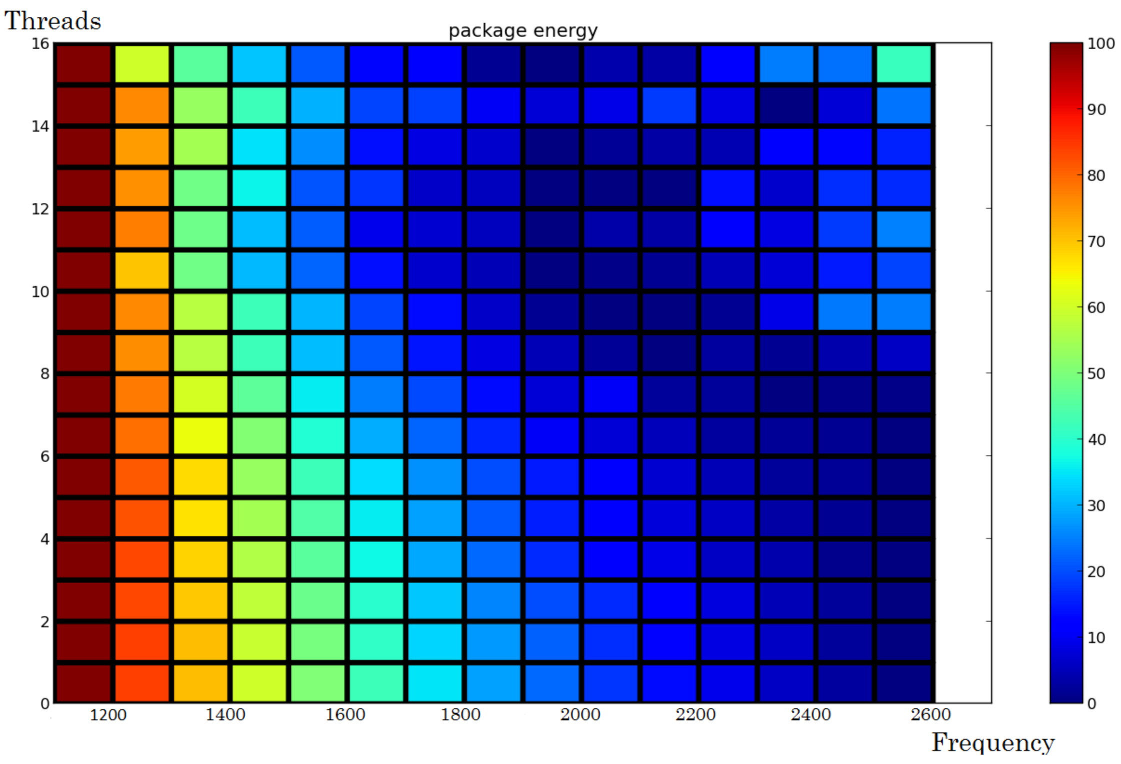
3d stencil



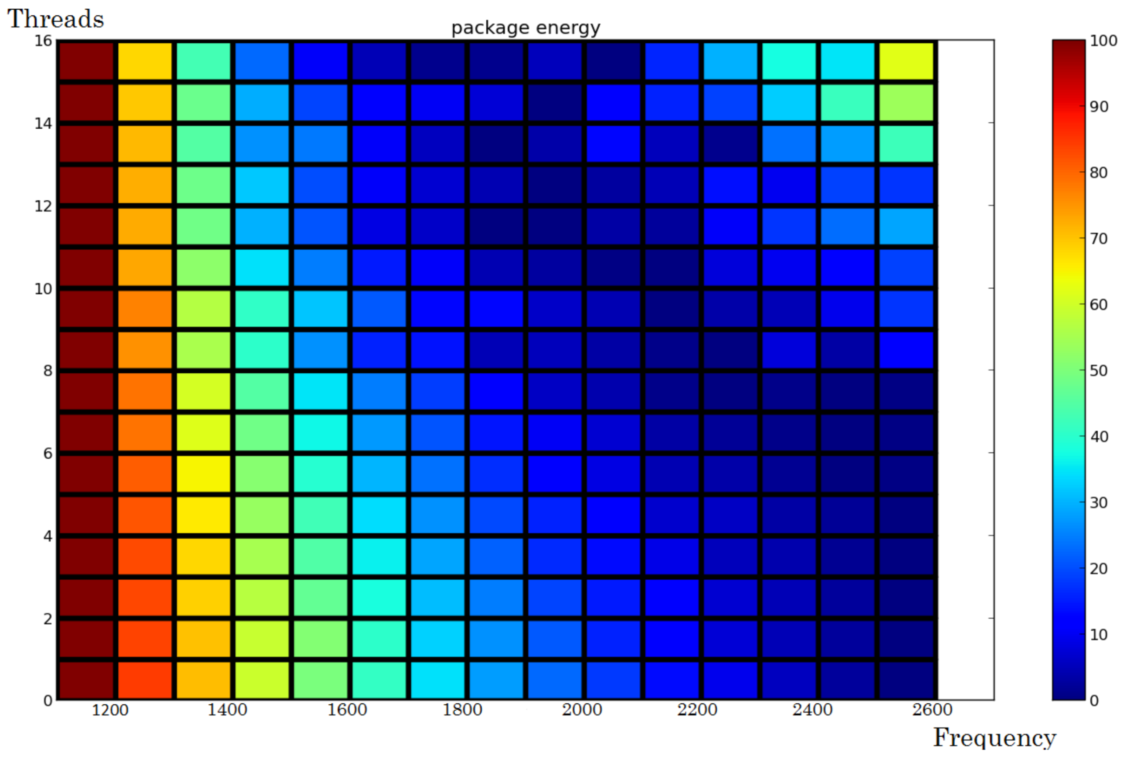
Dense matrix multiplication



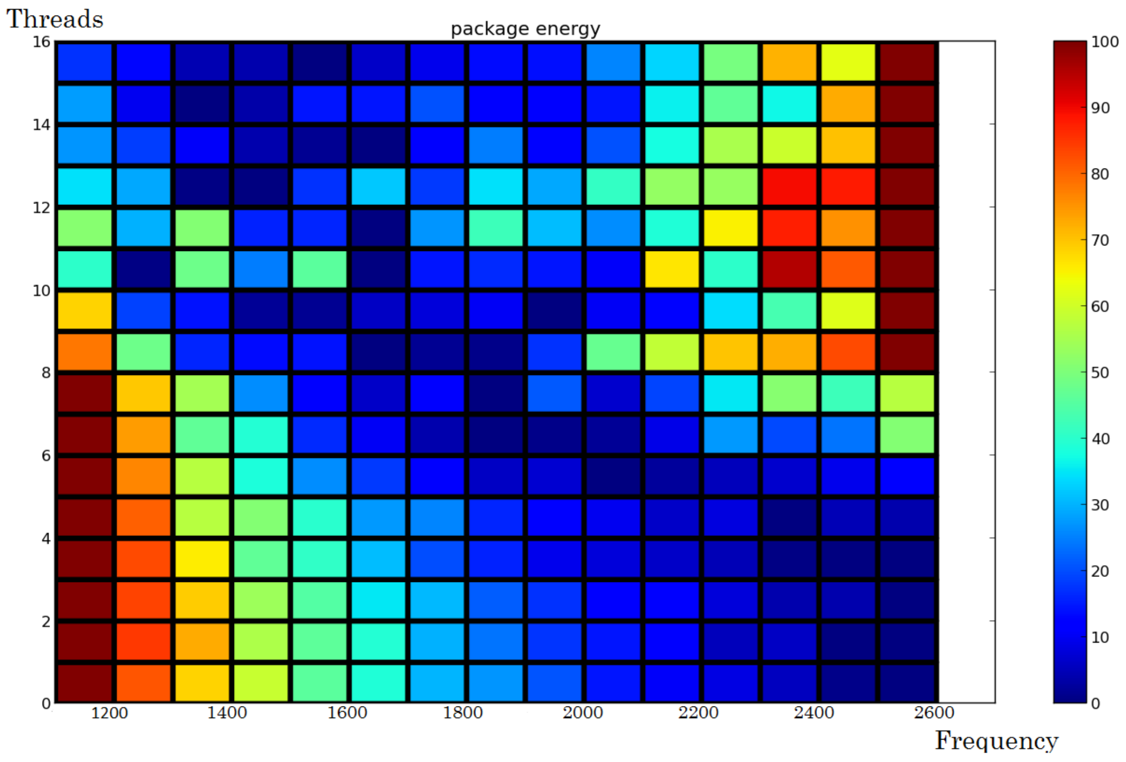
N-body



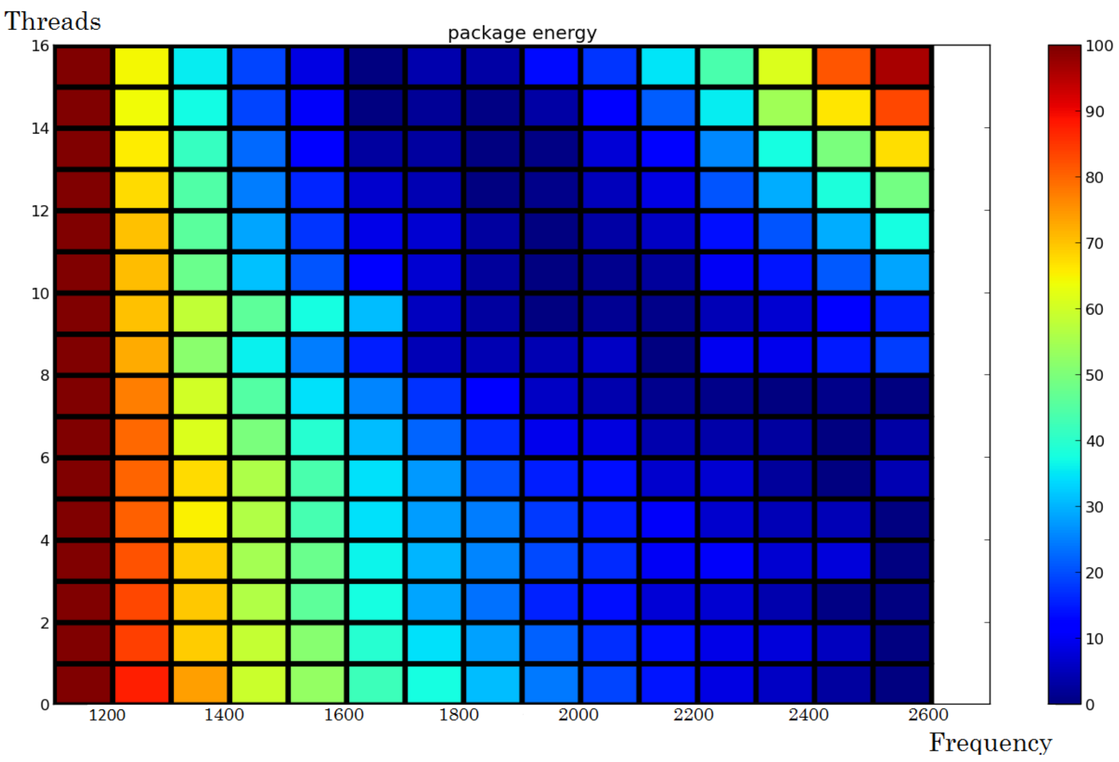
Sparse matrix vector multiplication



FFT



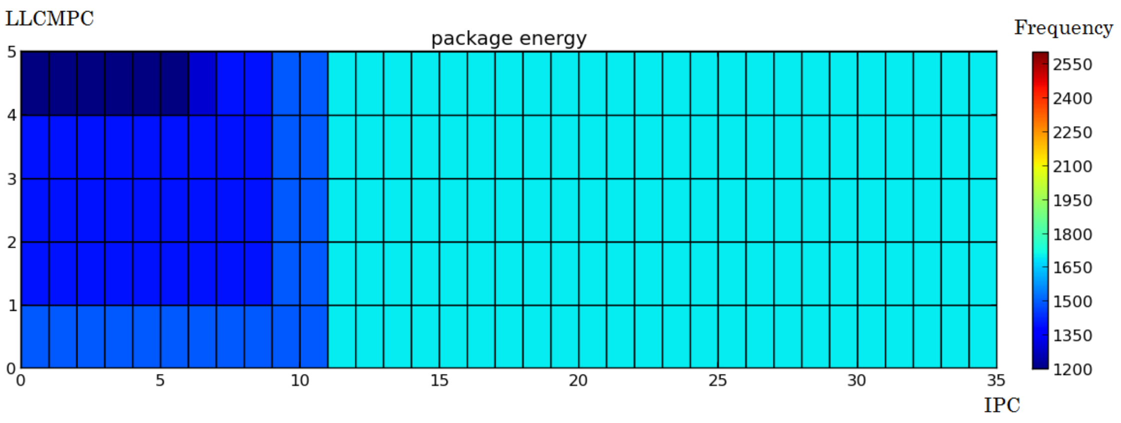
Merge Sort


Threads
package energy
16
14
12
10
8
6
4
2
0
1200
1400
1600
1800
2000
2200
2400
2600
Frequency
100
90
80
70
60
50
40
30
20
10
0

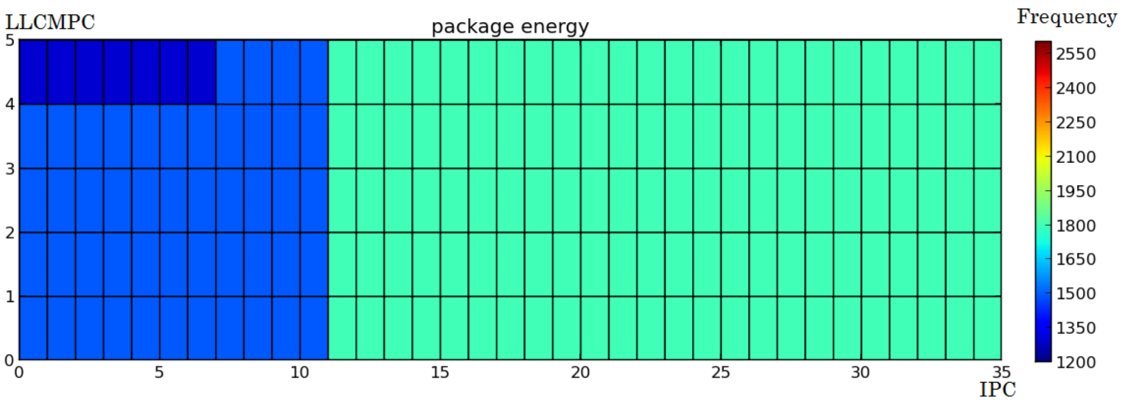
Lookup table

Operations per Joule

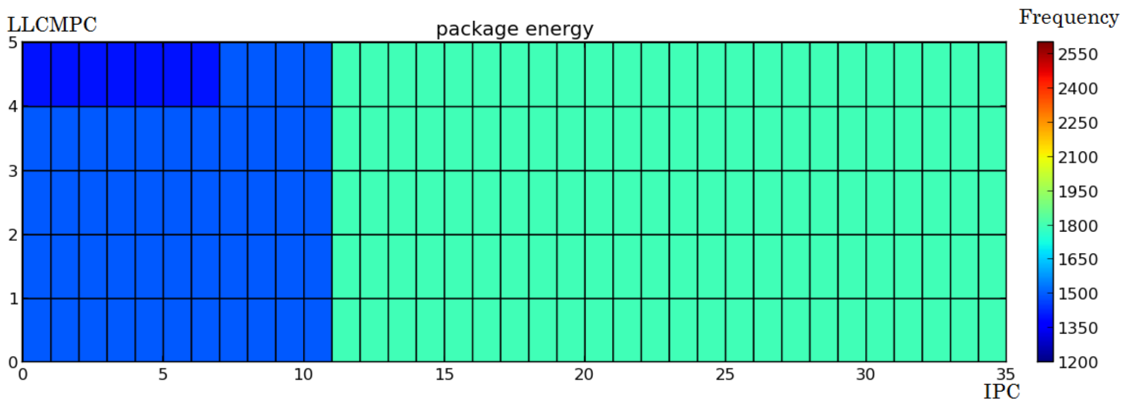
16 active cores



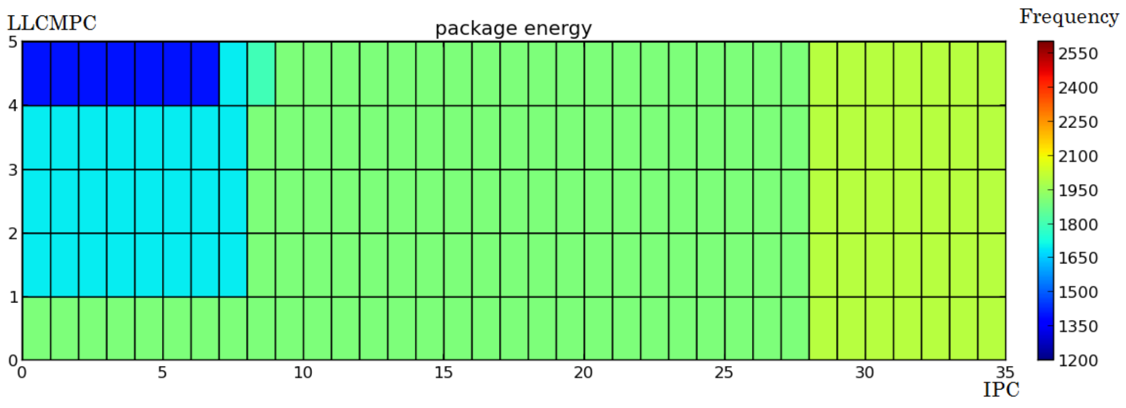
15 active cores



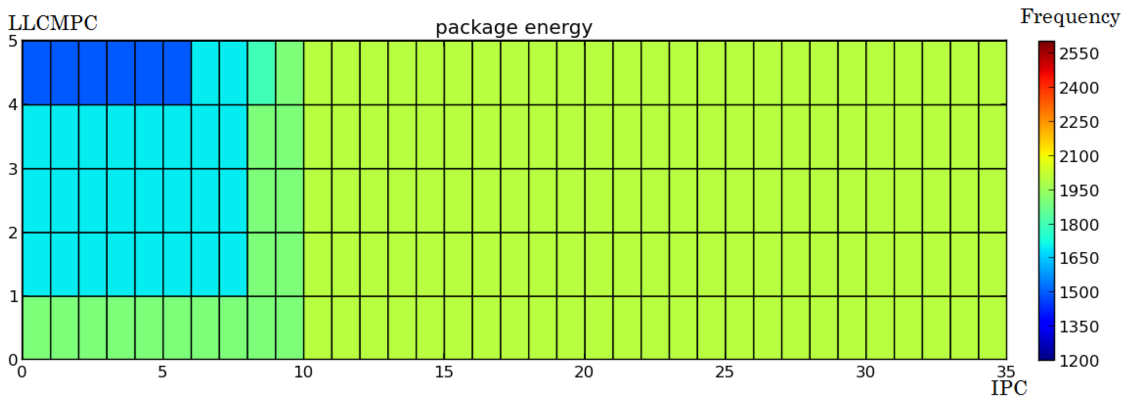
14 active cores



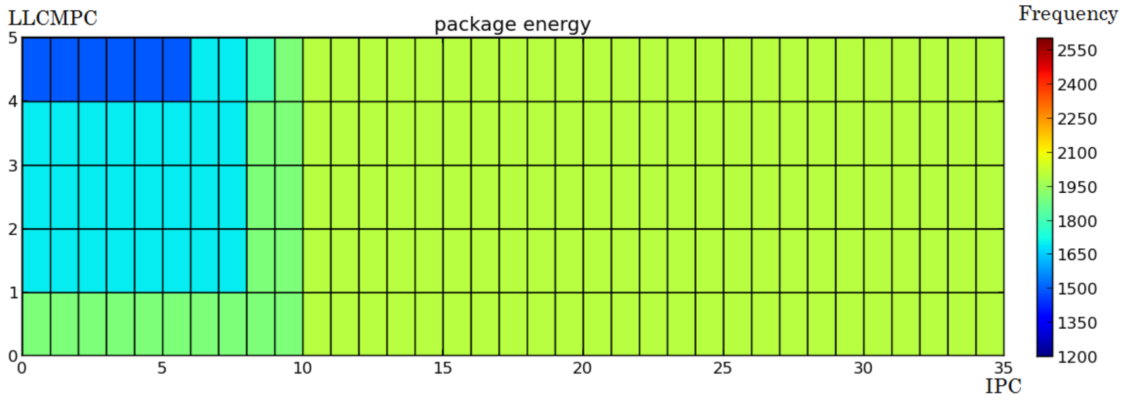
13 active cores



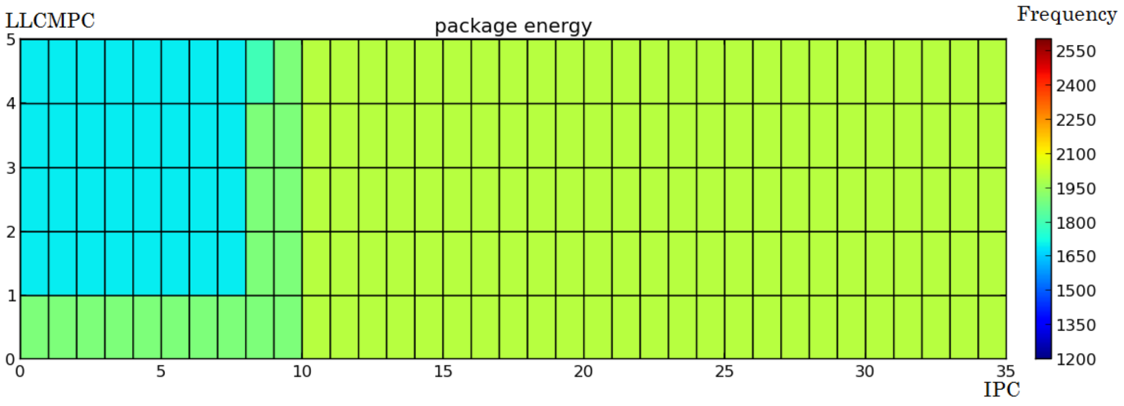
12 active cores



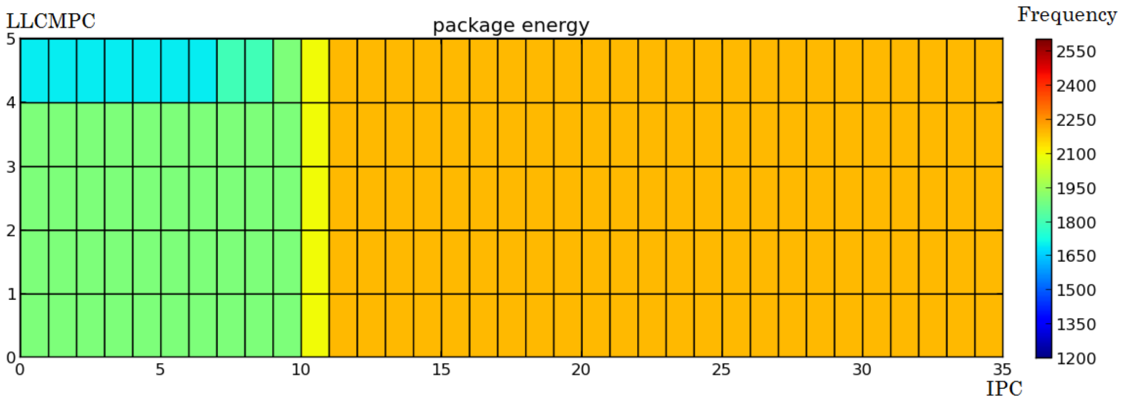
11 active cores



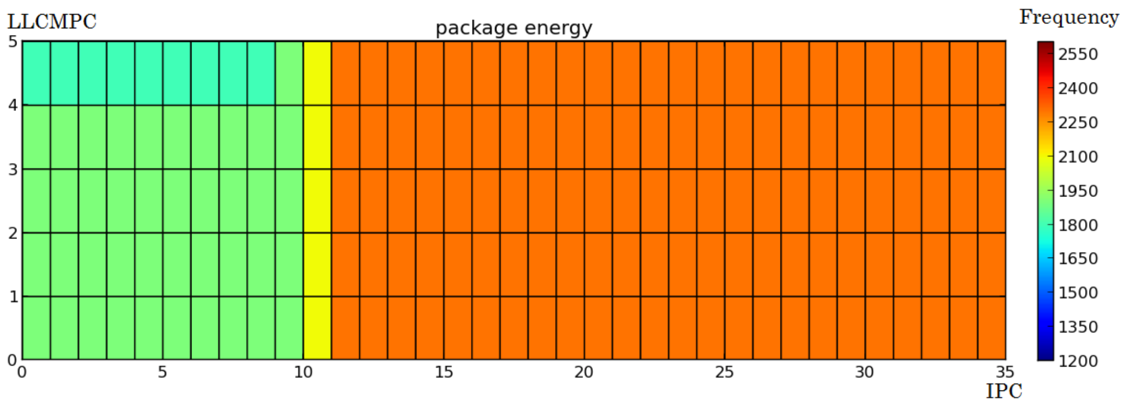
10 active cores



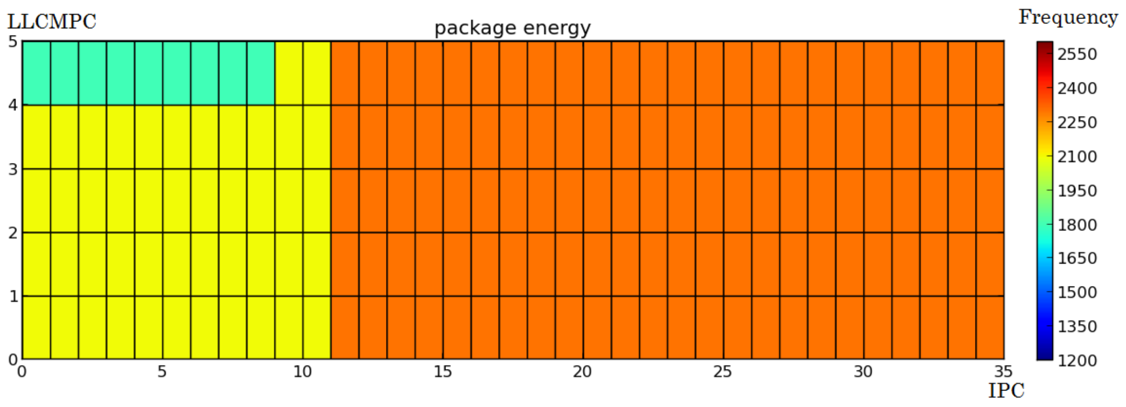
9 active cores



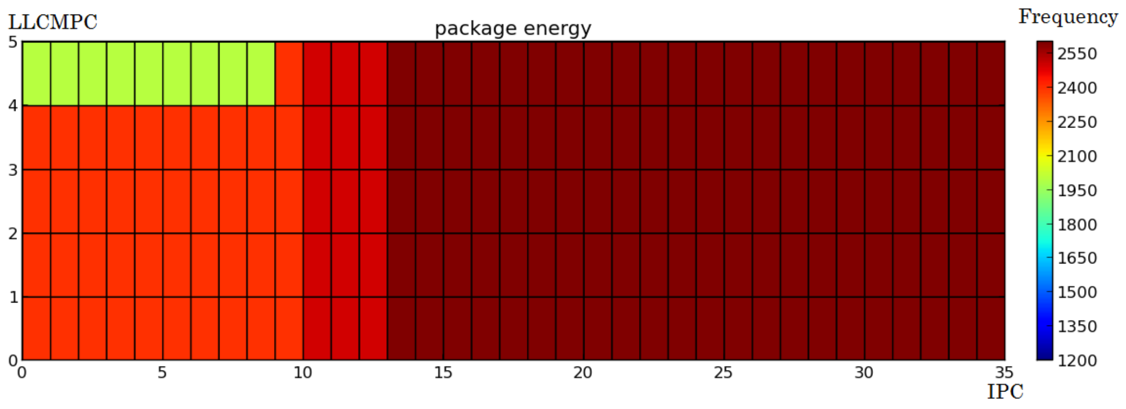
8 active cores



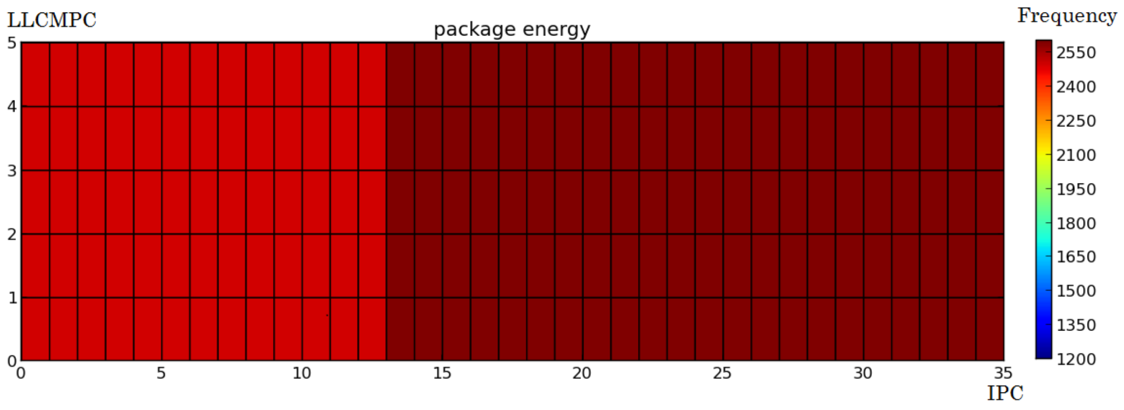
7 active cores



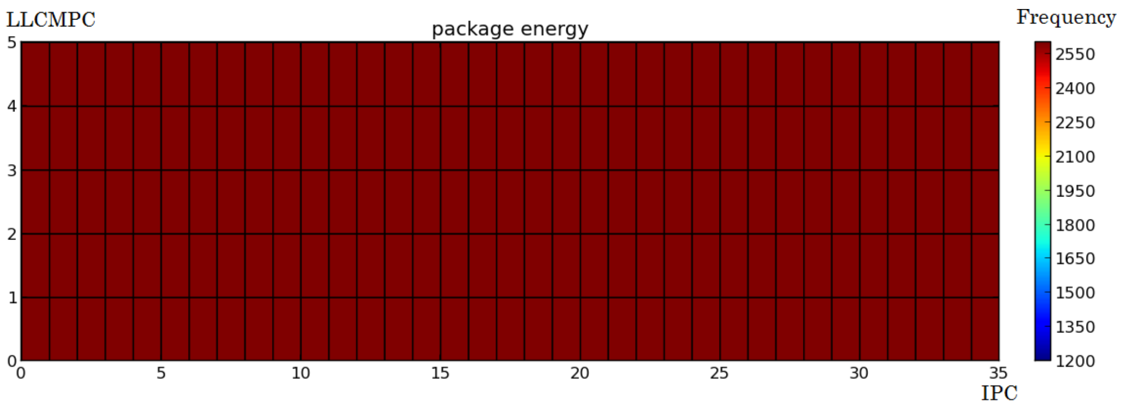
6 active cores



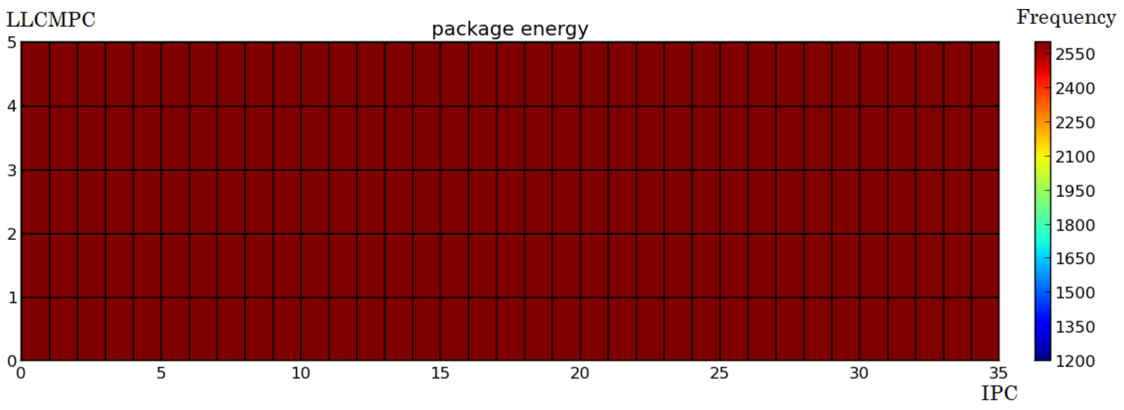
5 active cores



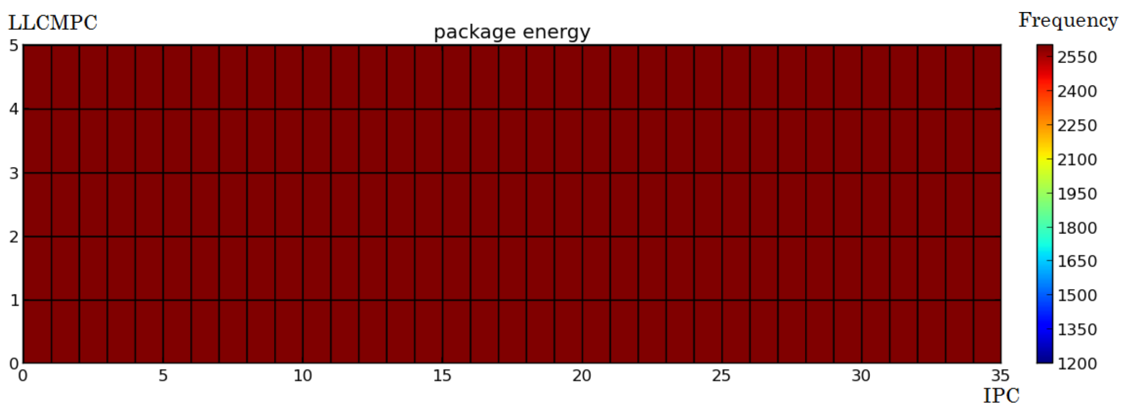
4 active cores



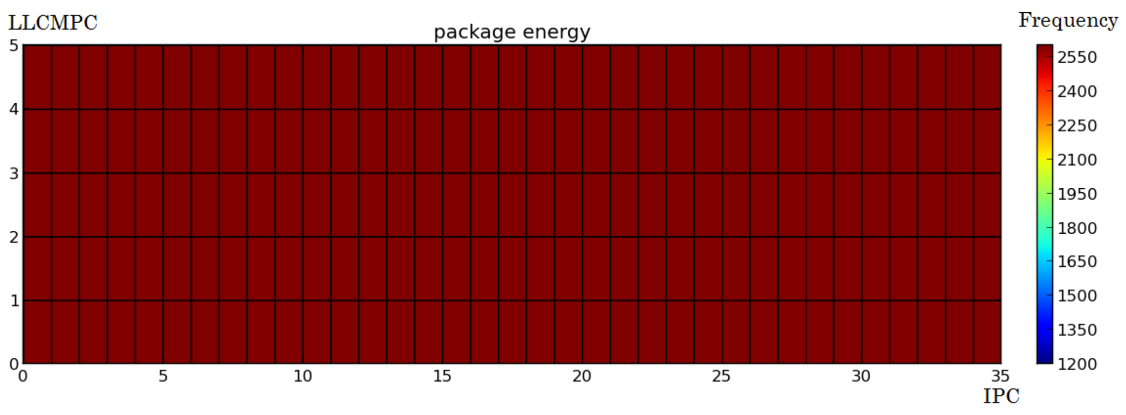
3 active cores



2 active cores

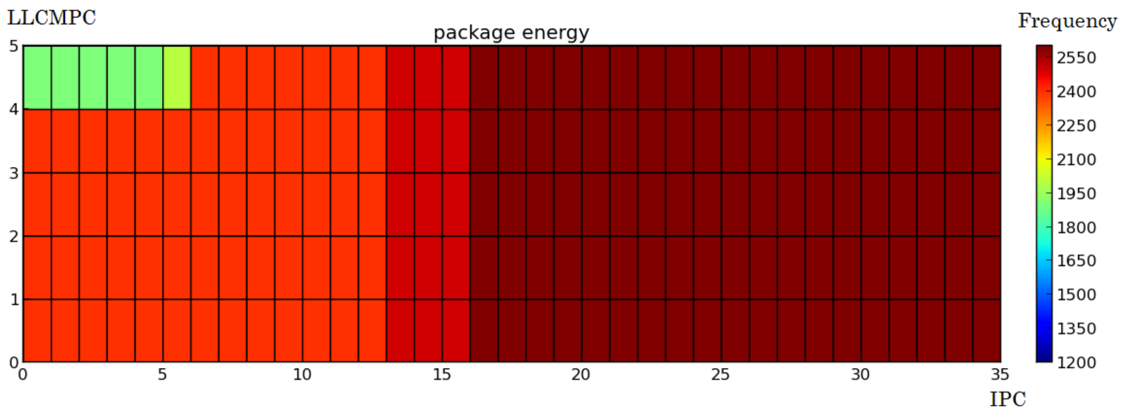


1 active core

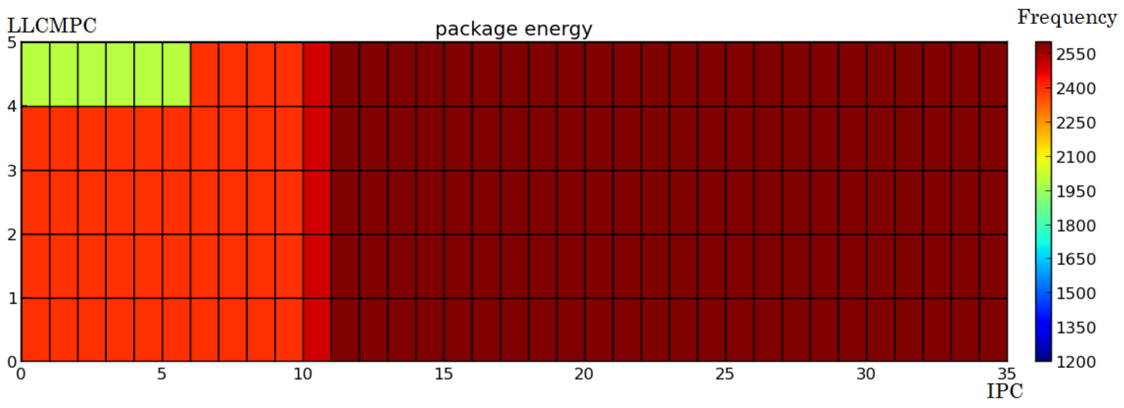


Energy Delay Product

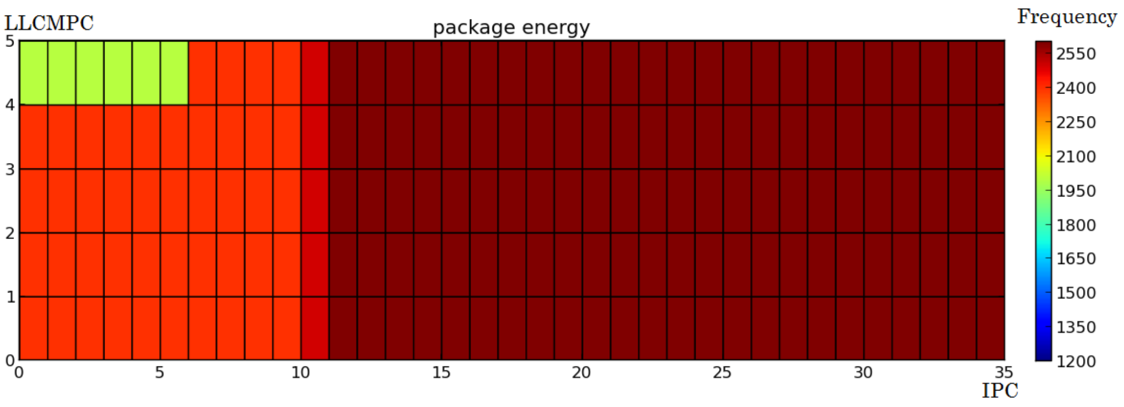
16 active cores



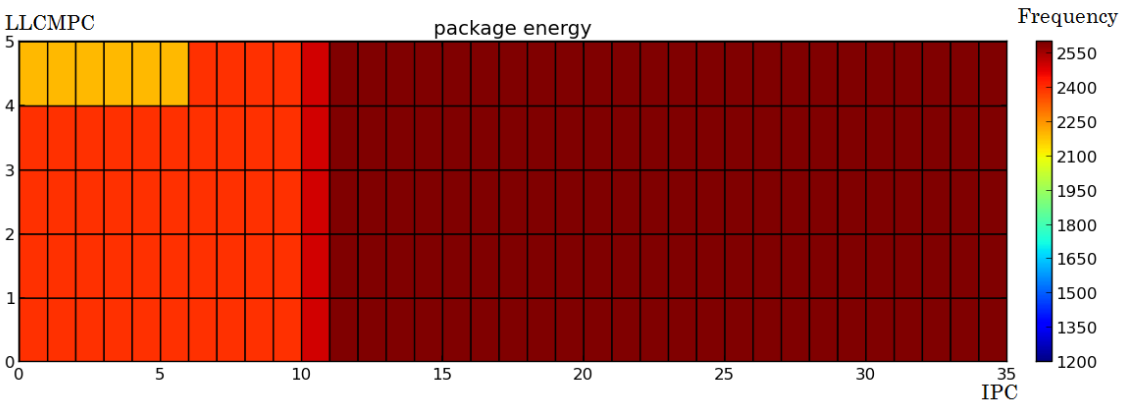
15 active cores



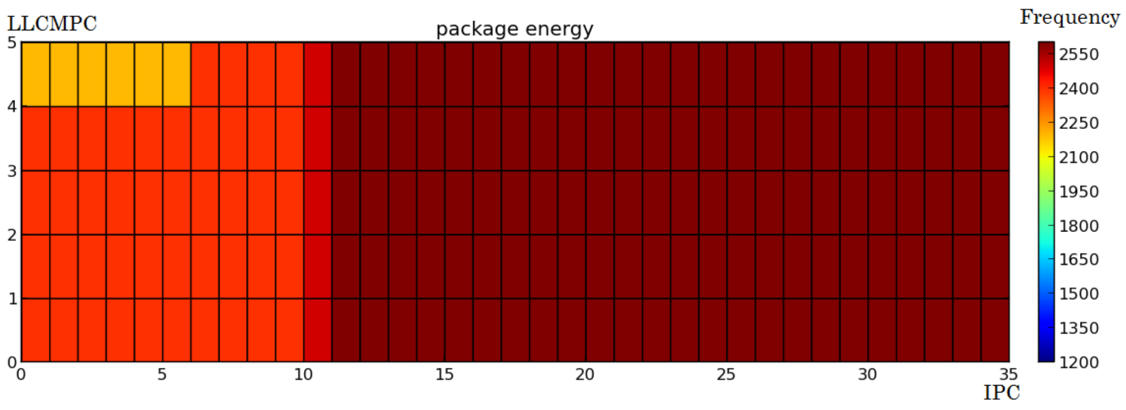
14 active cores



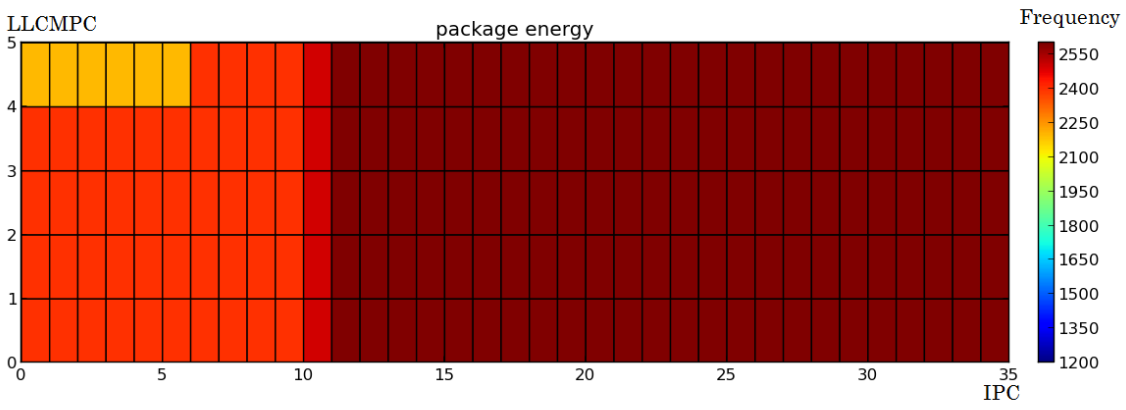
13 active cores



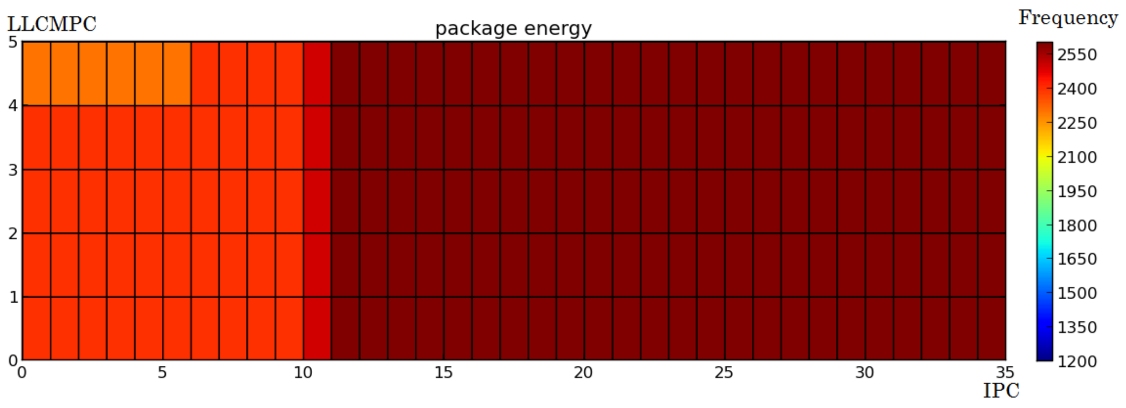
12 active cores



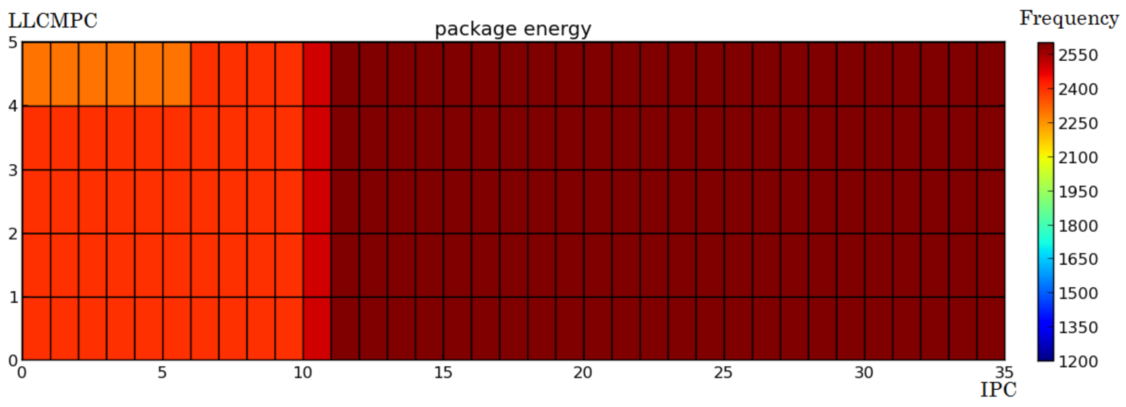
11 active cores



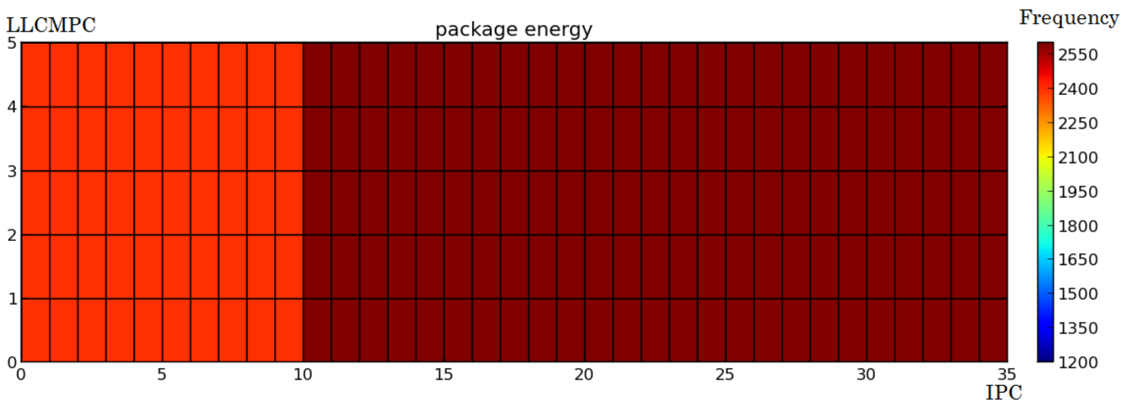
10 active cores



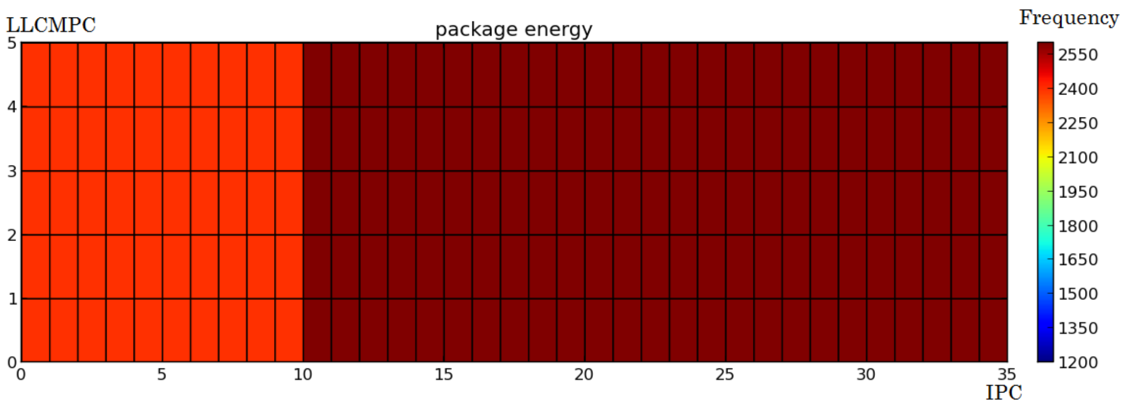
9 active cores



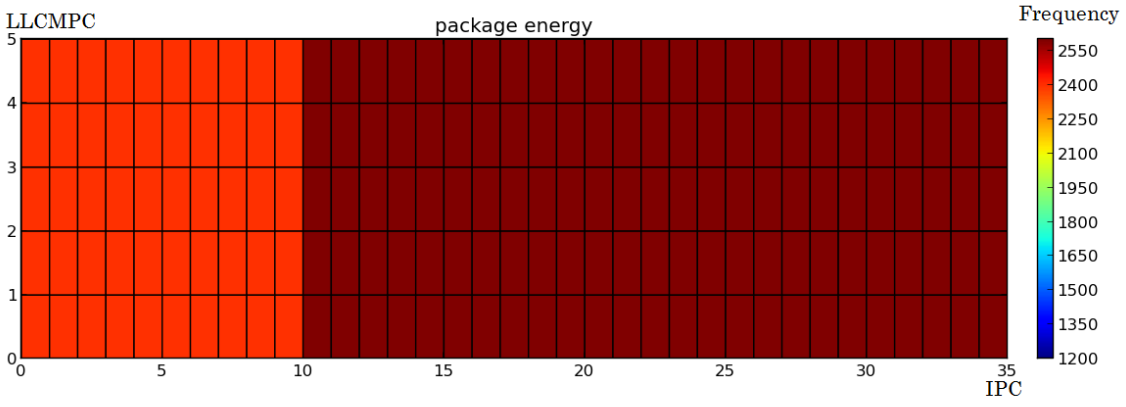
8 active cores



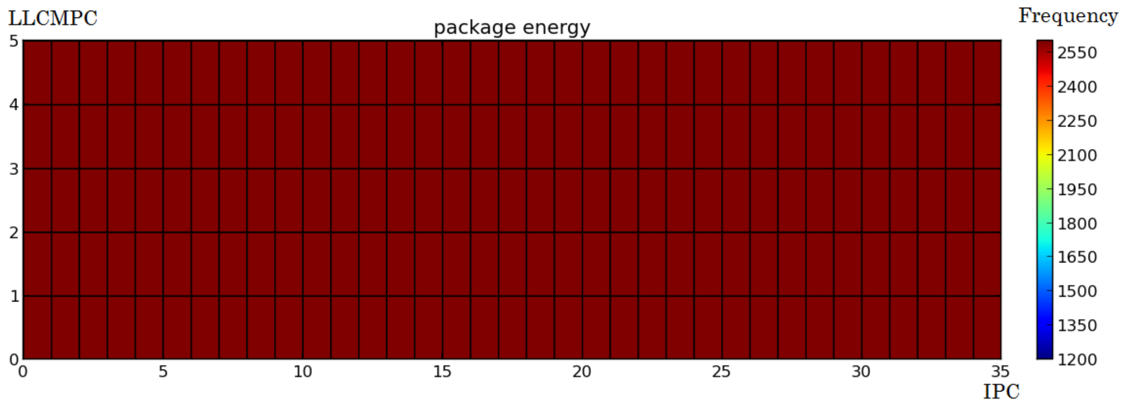
7 active cores



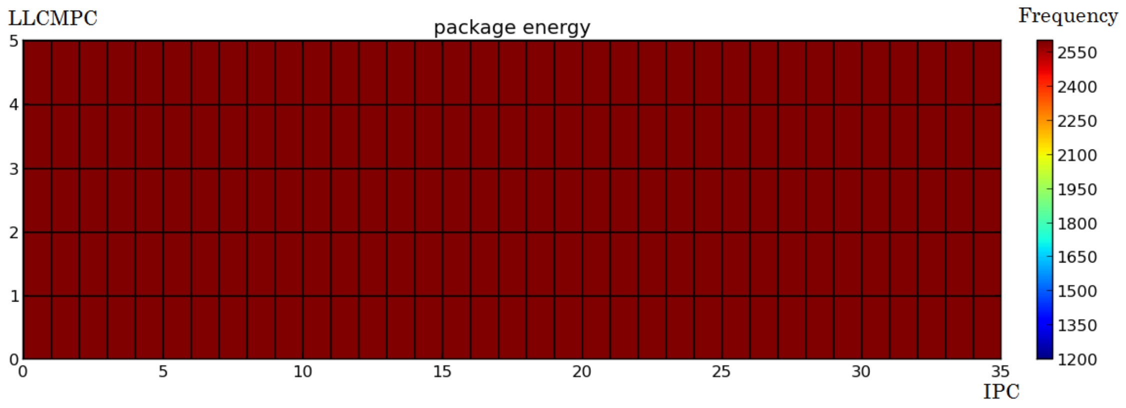
6 active cores



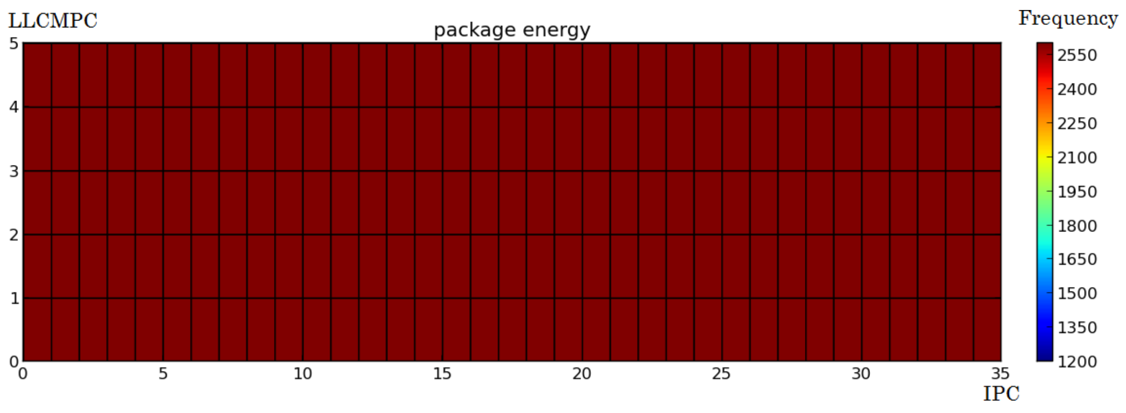
5 active cores



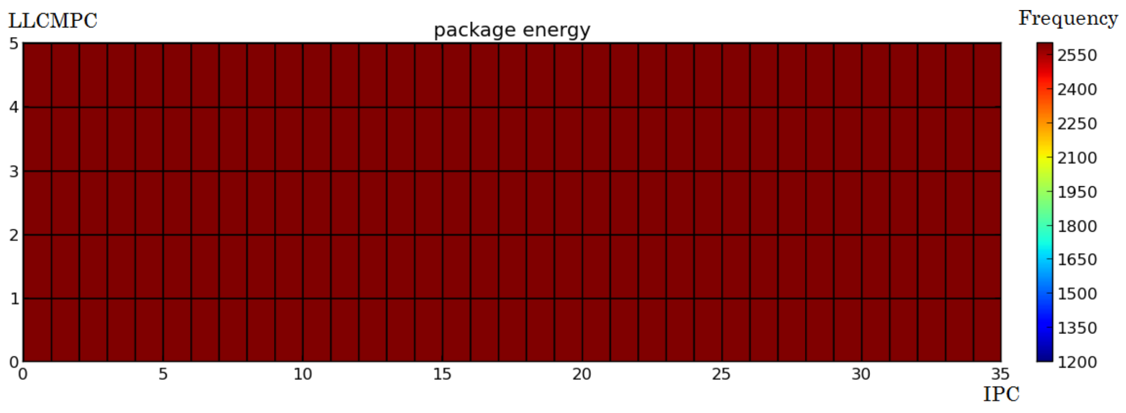
4 active cores



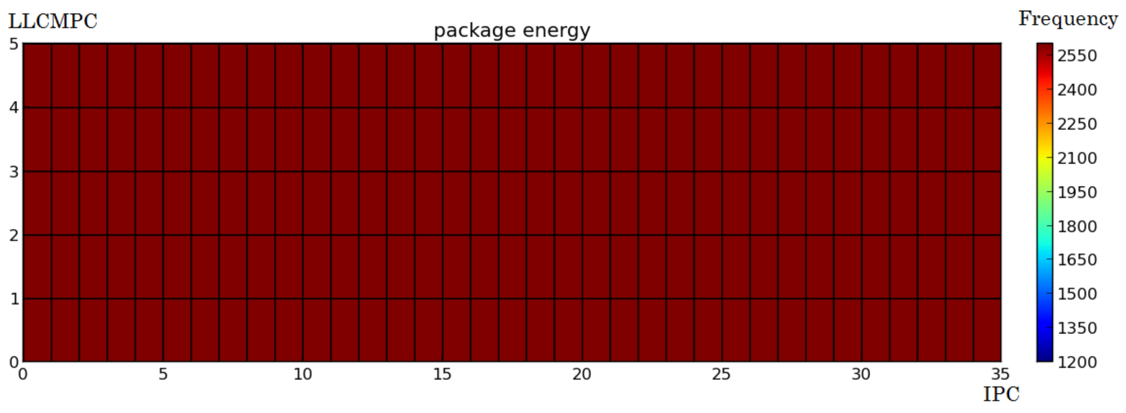
3 active cores



2 active cores

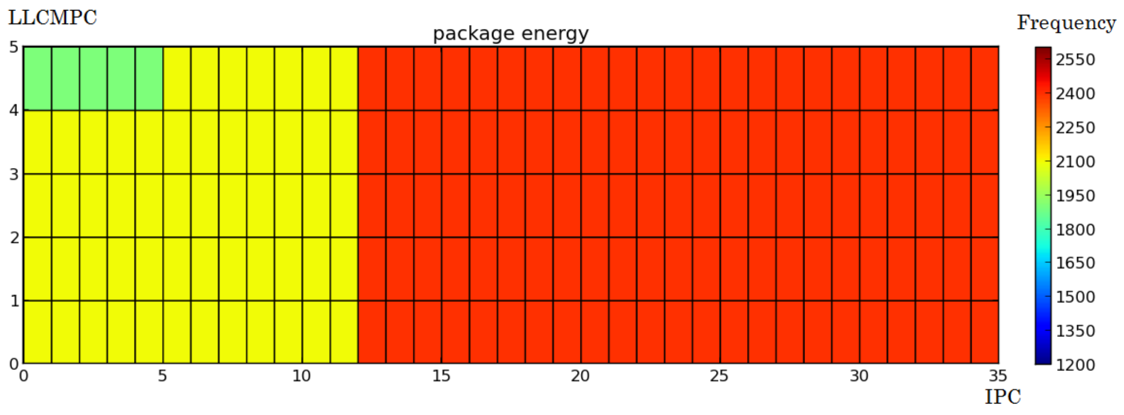


1 active core

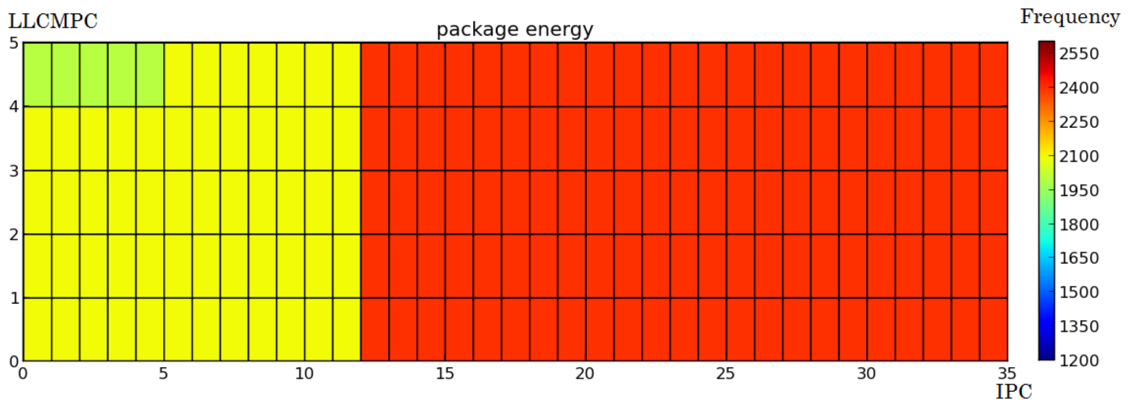


Operations per Joule under the constraint that performance cannot suffer by more than 10%

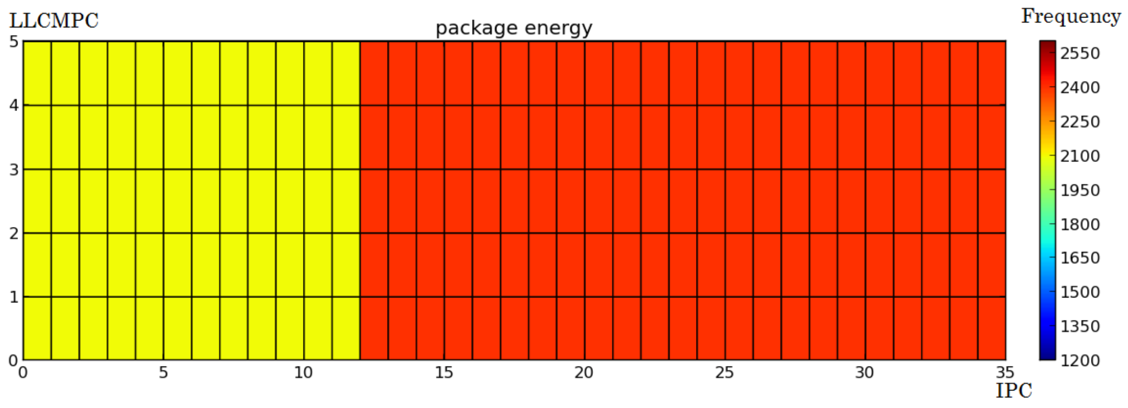
16 active cores



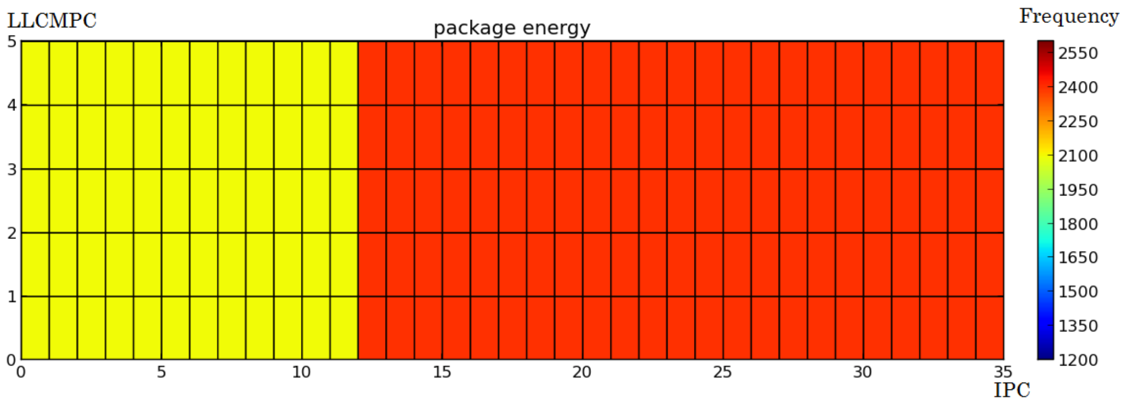
15 active cores



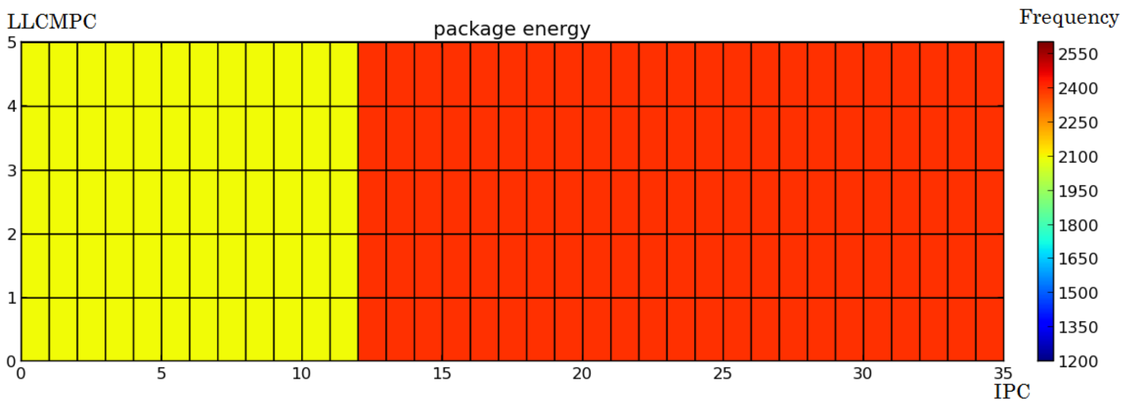
14 active cores



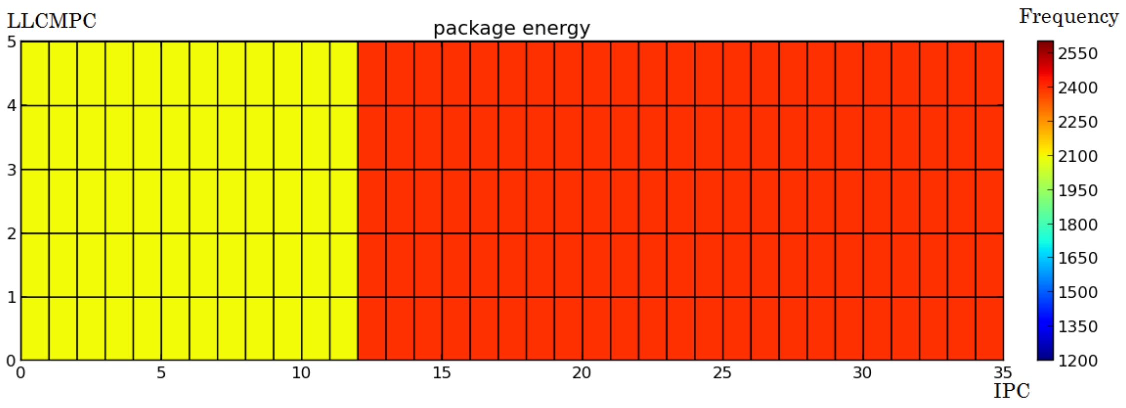
13 active cores



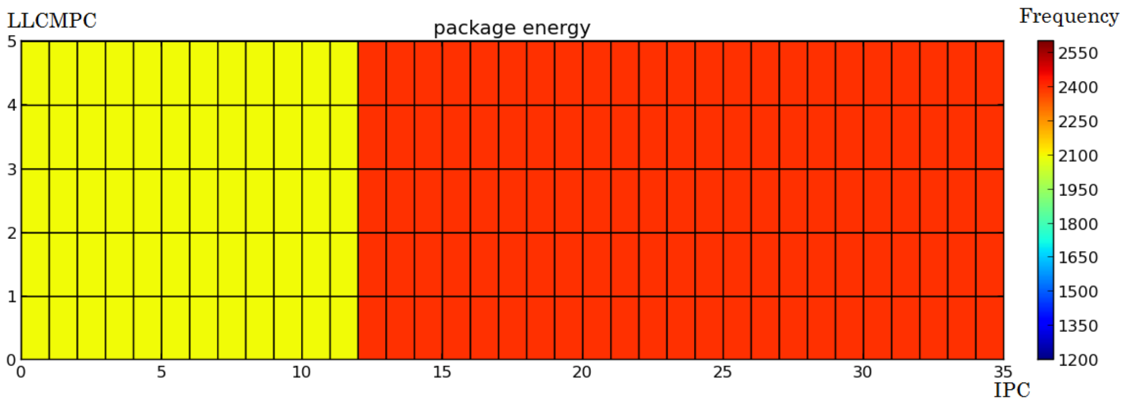
12 active cores



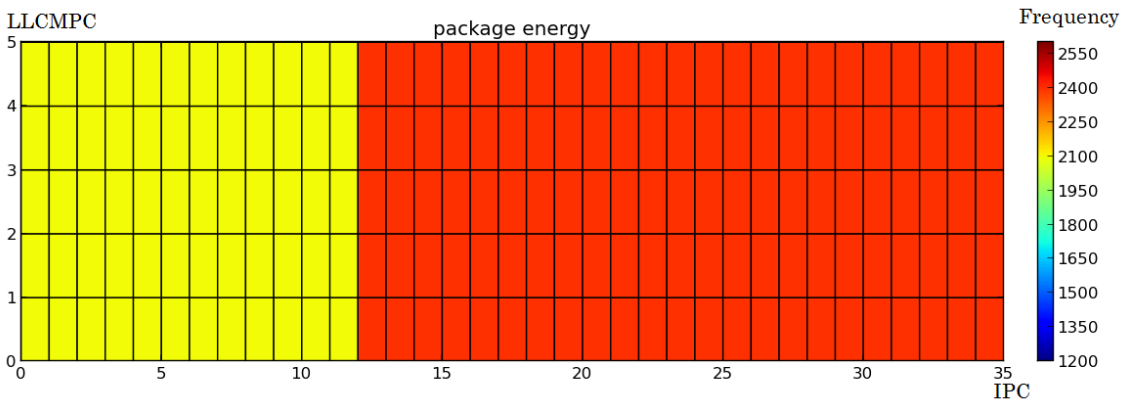
11 active cores



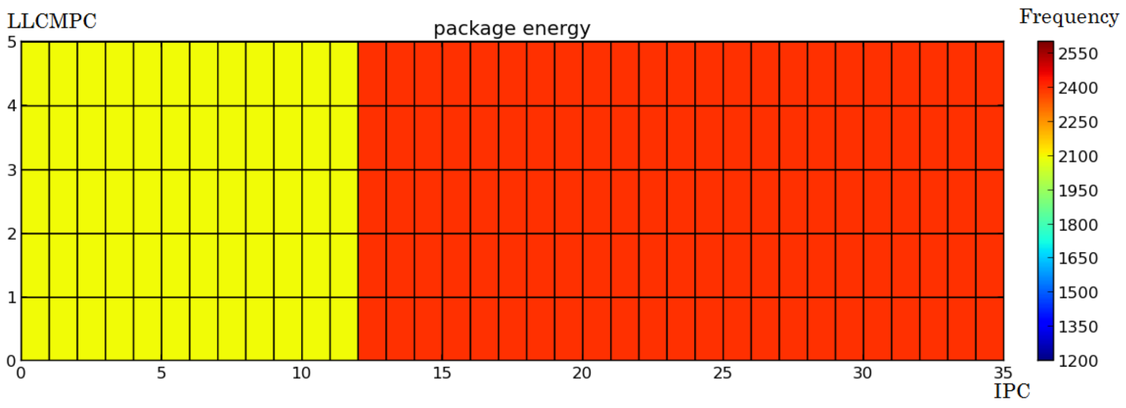
10 active cores



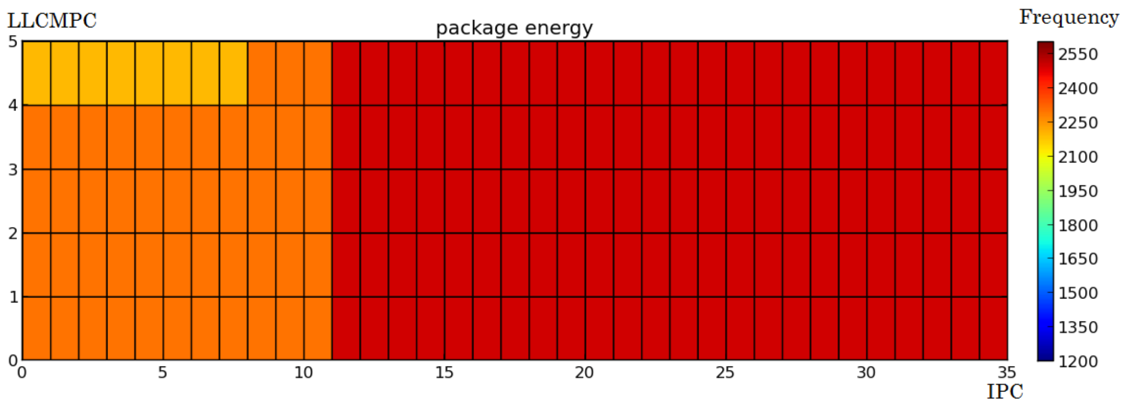
9 active cores



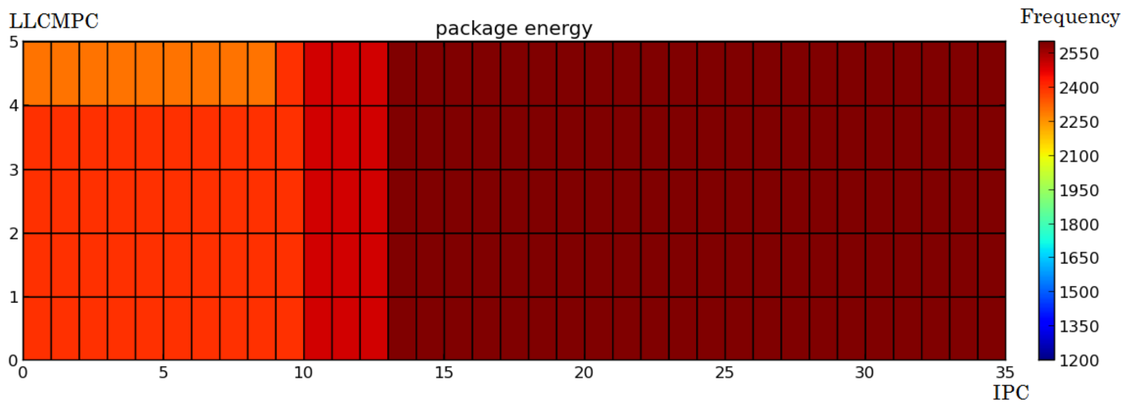
8 active cores



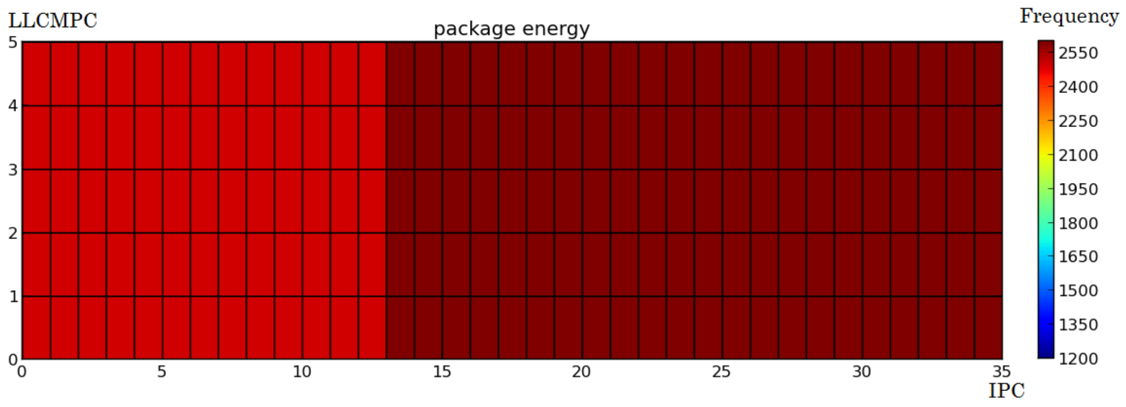
7 active cores



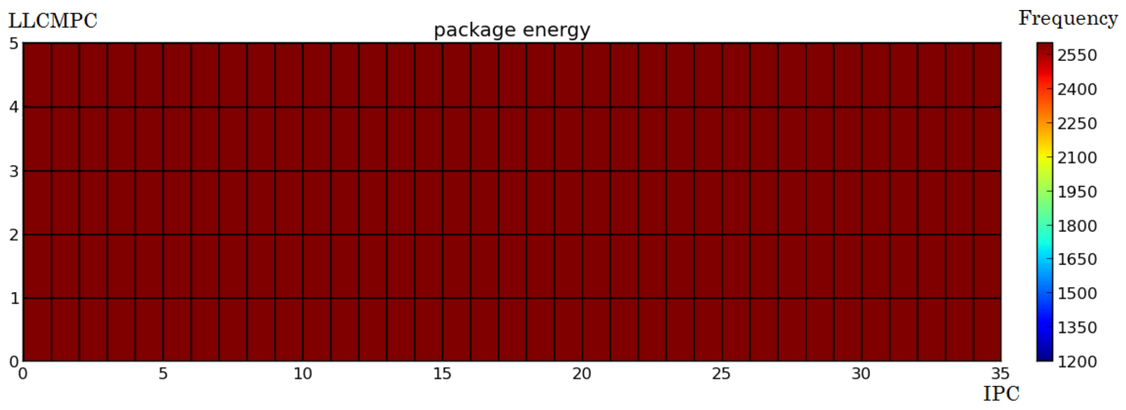
6 active cores



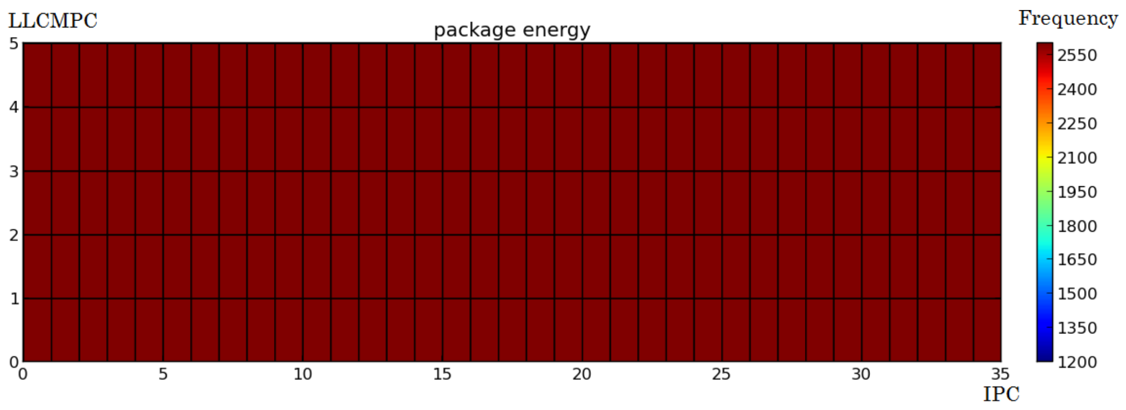
5 active cores



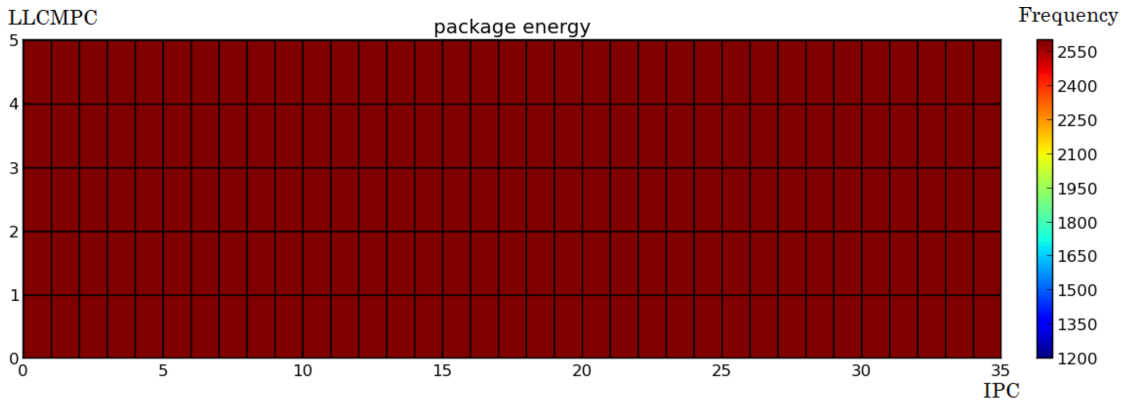
4 active cores



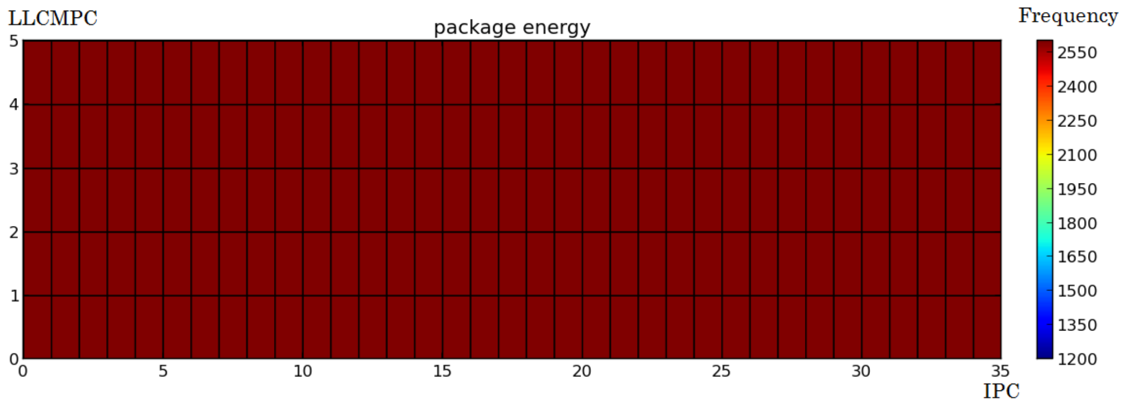
3 active cores



2 active cores

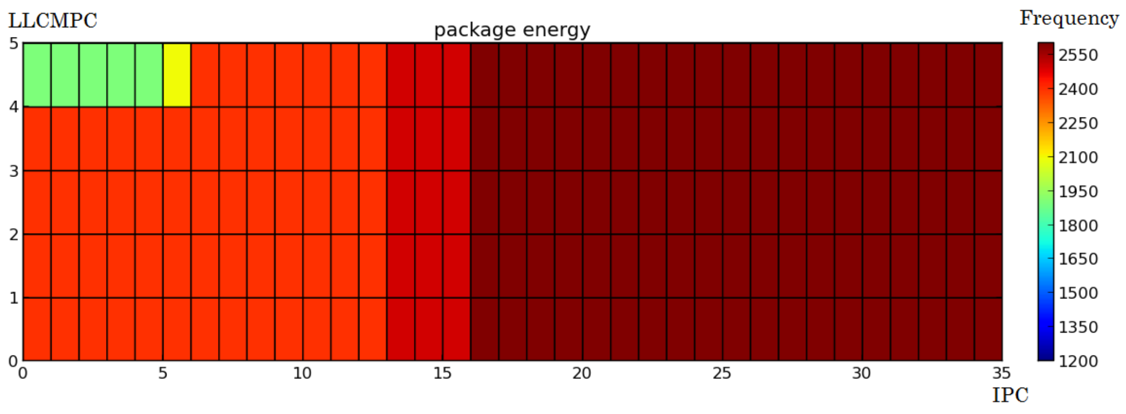


1 active core

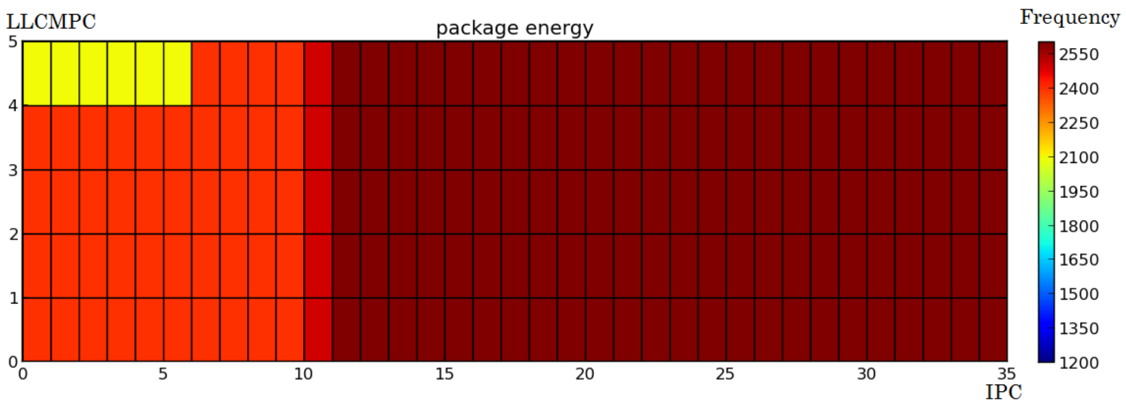


Energy Delay Product under the constraint that performance cannot suffer by more than 10%

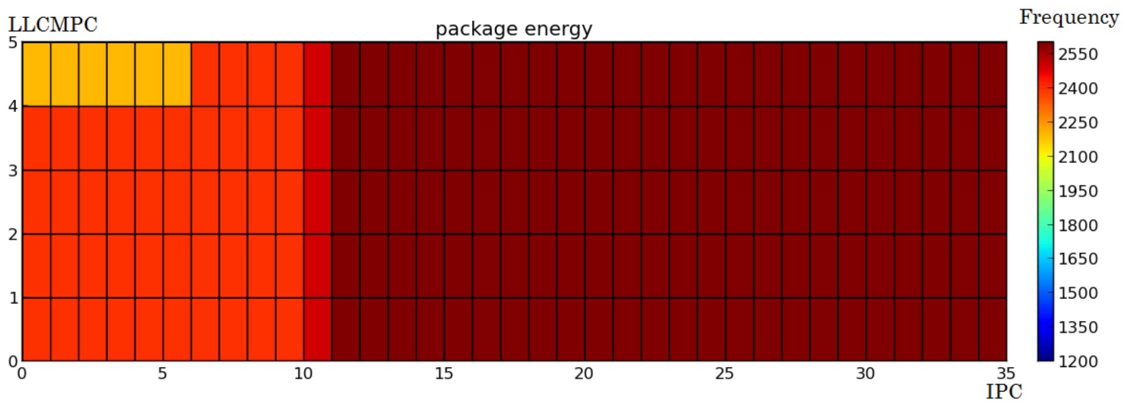
16 active cores



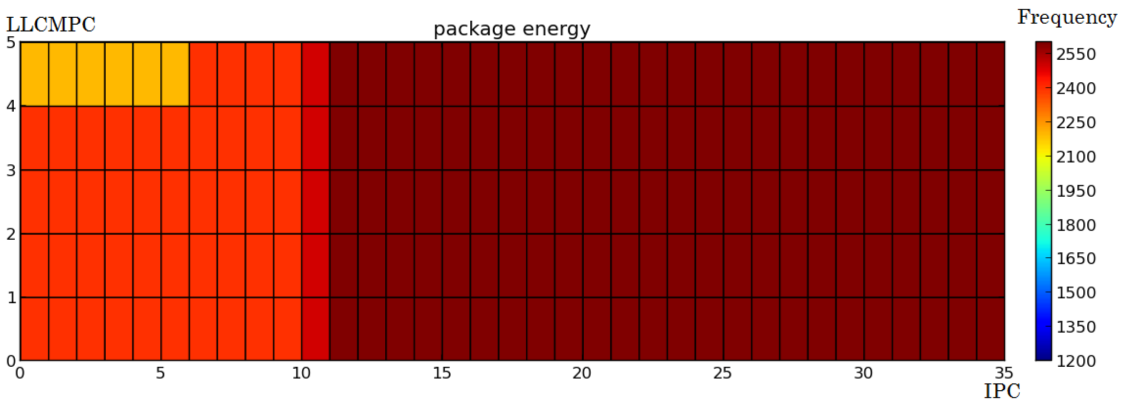
15 active cores



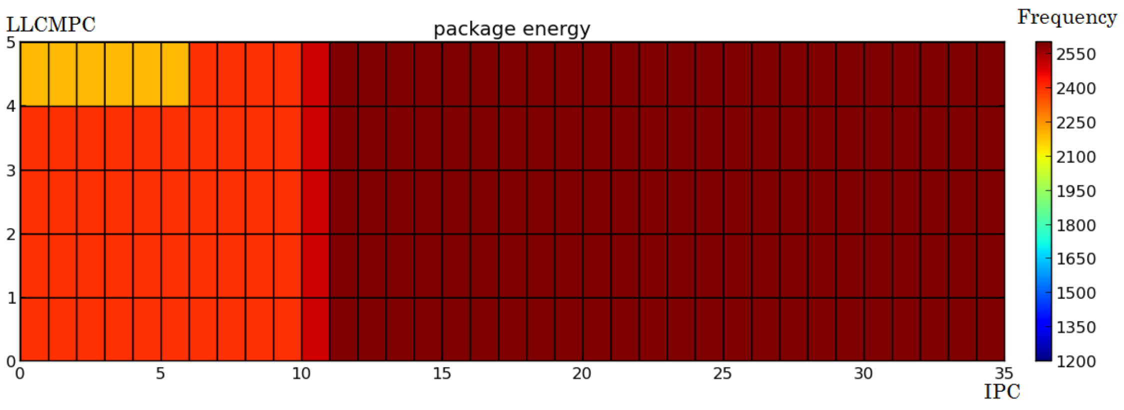
14 active cores



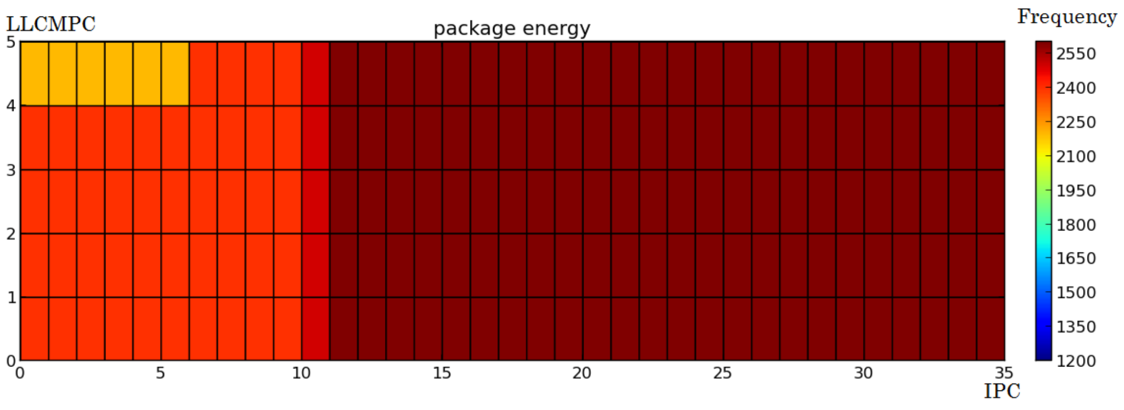
13 active cores



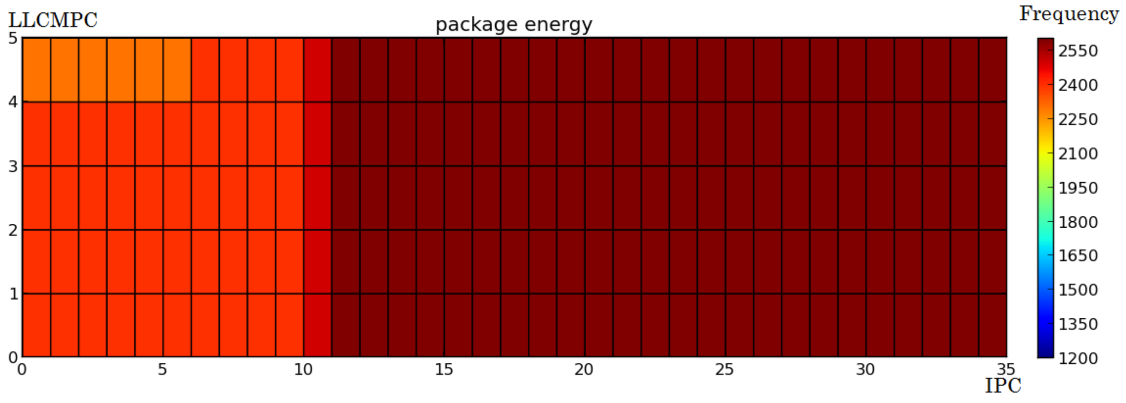
12 active cores



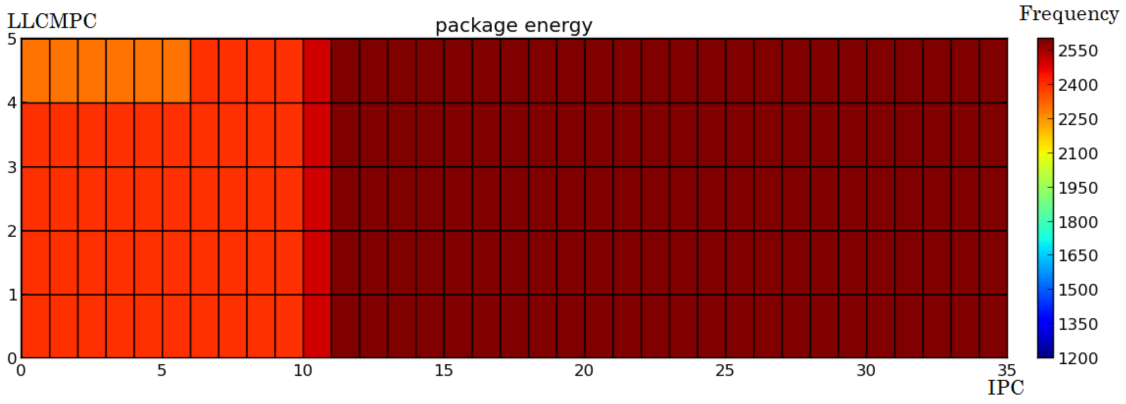
11 active cores



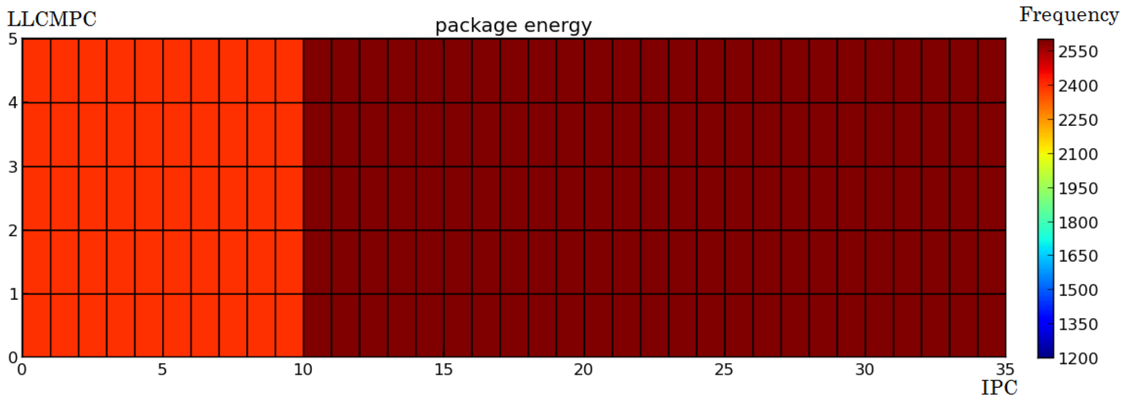
10 active cores



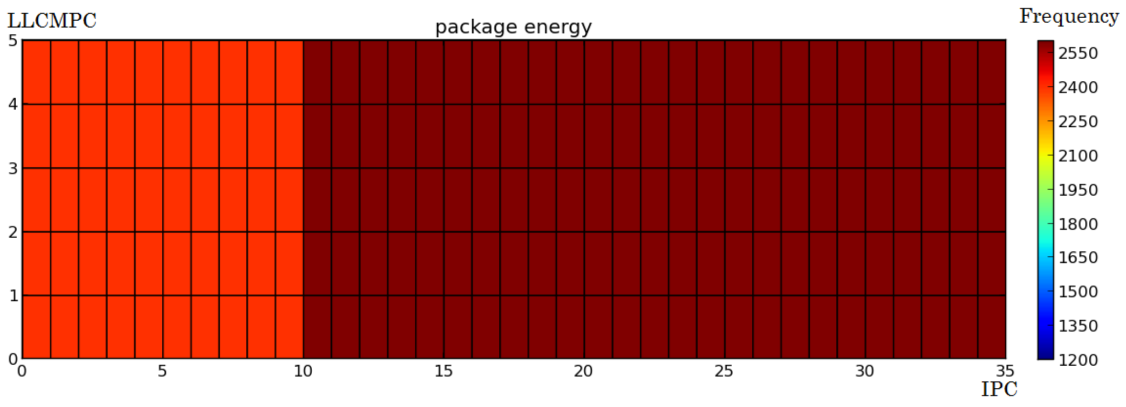
9 active cores



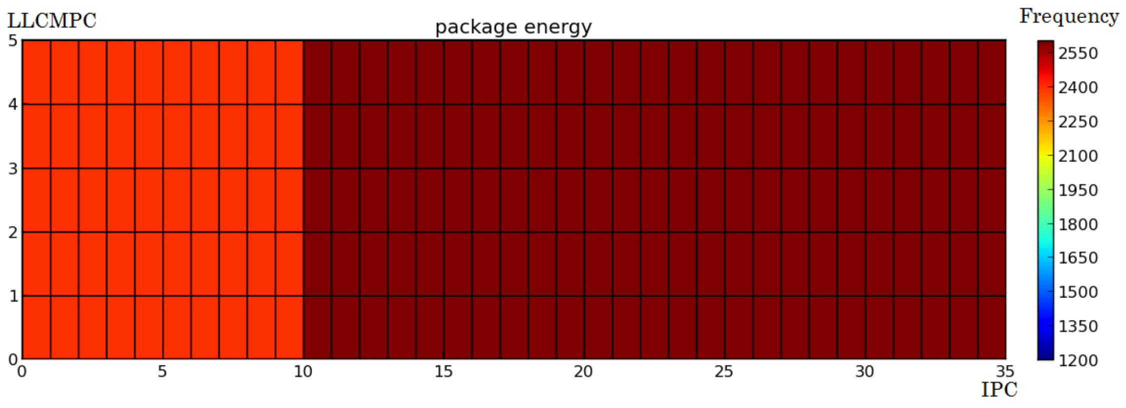
8 active cores



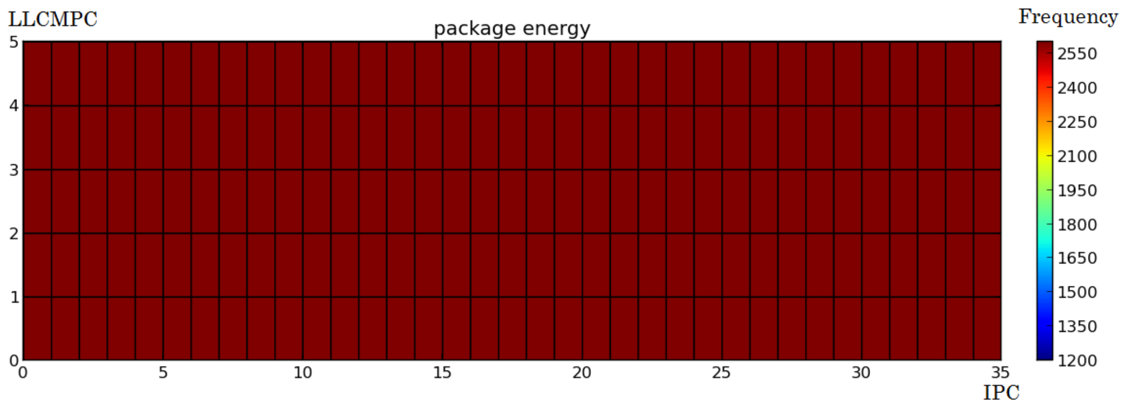
7 active cores



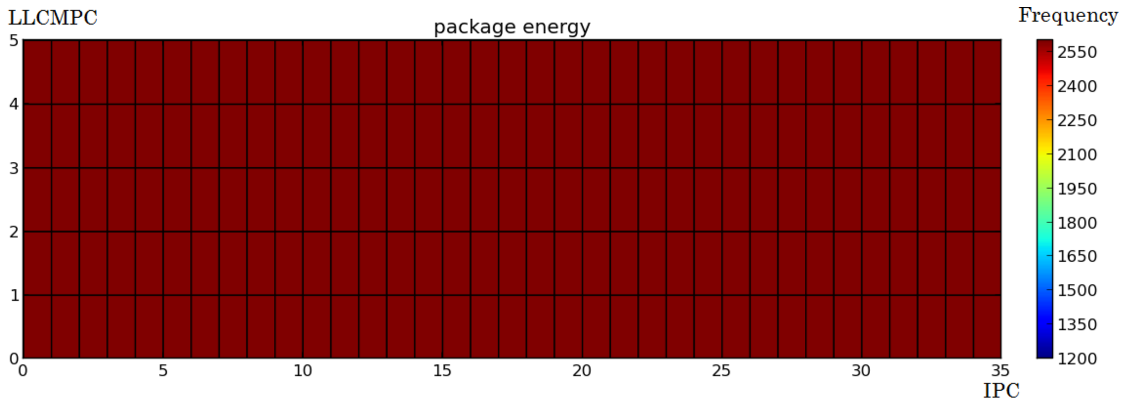
6 active cores



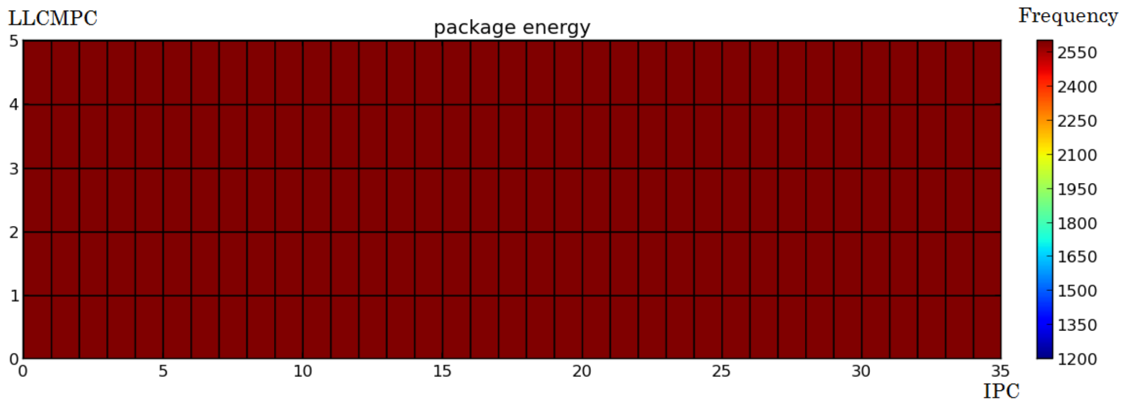
5 active cores



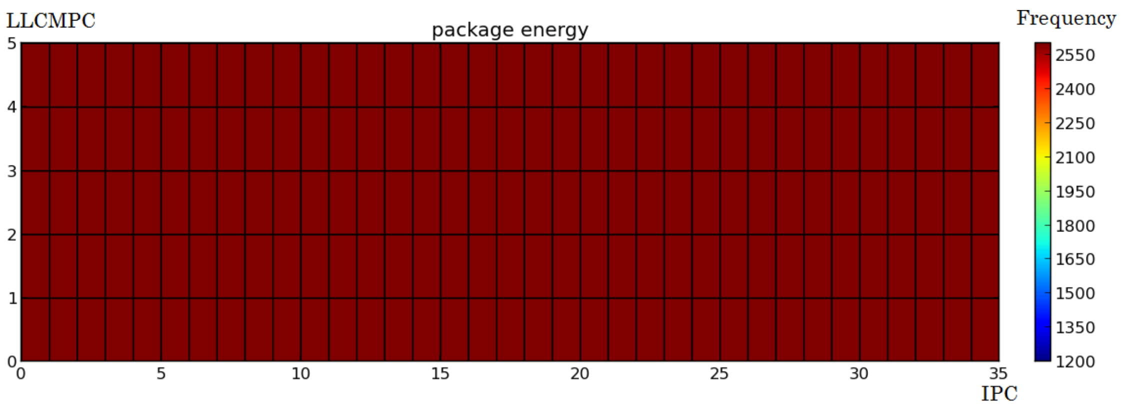
4 active cores



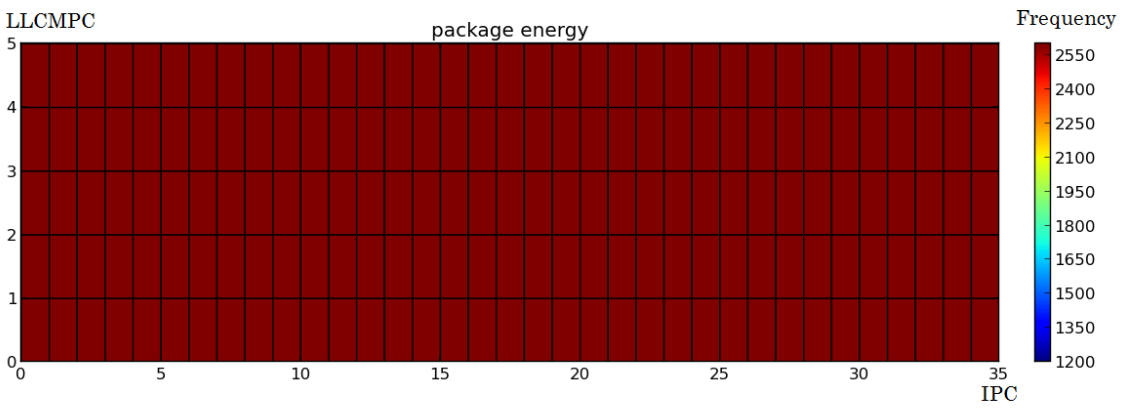
3 active cores



2 active cores



1 active core

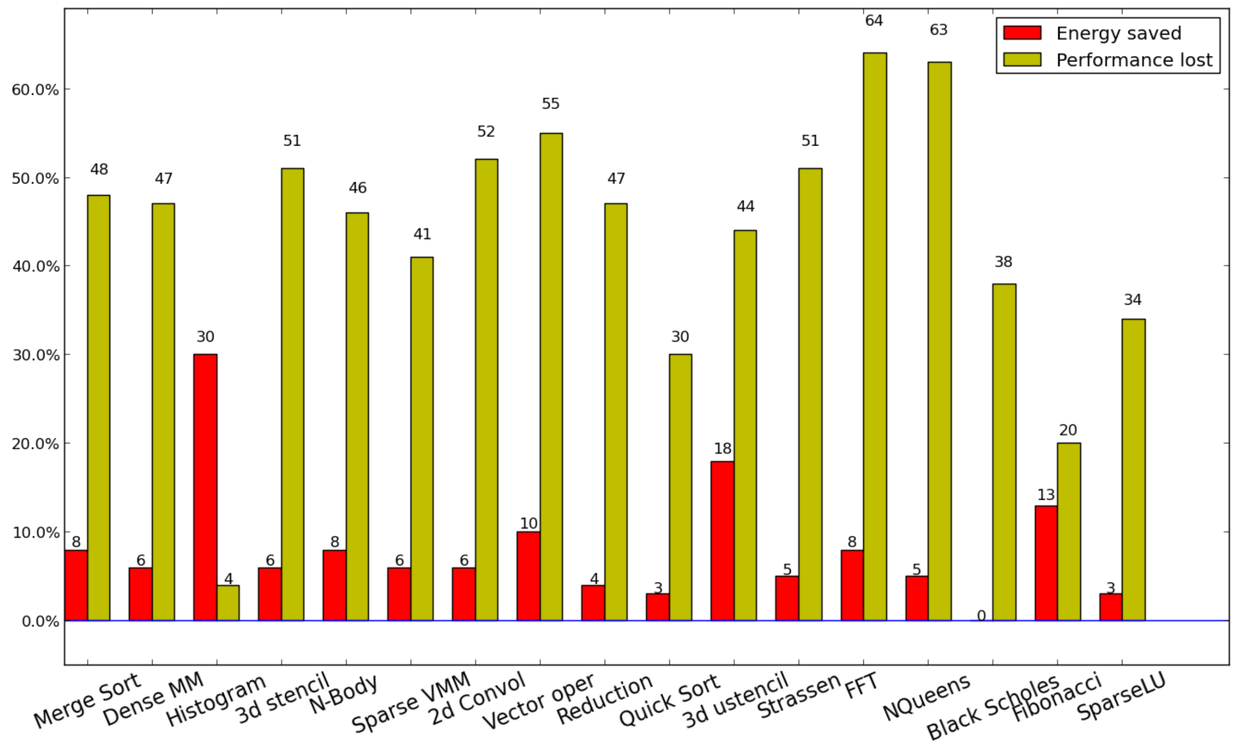


Experiment results

Results for the metric Operations per Joule

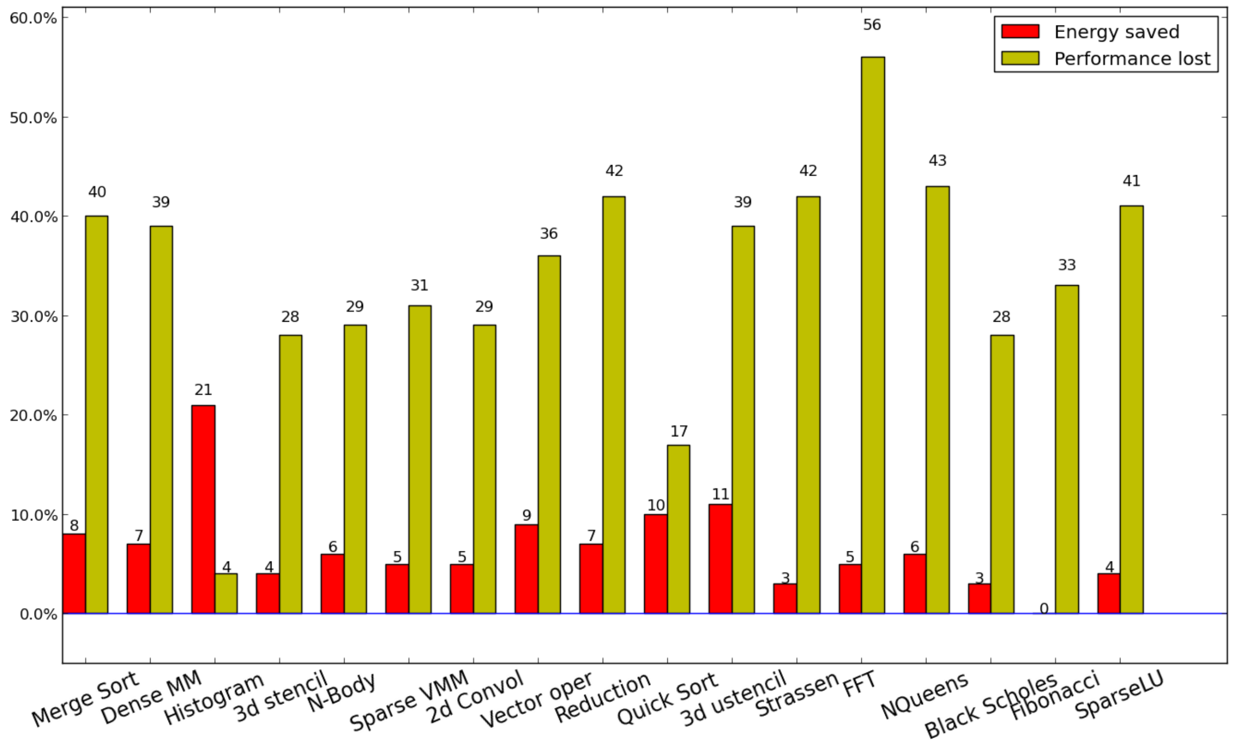
0.0.7 Results for 16 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	1700 MHz	1713 MHz	0.76 %
Reduction	1700 MHz	1758 MHz	3.41 %
Histogram	1200 MHz	1689 MHz	40.75 %
2d convolution	1900 MHz	1700 MHz	10.53 %
3d stencil	1700 MHz	1704 MHz	0.24 %
Dense matrix multiplication	1700 MHz	1722 MHz	1.29 %
Sparse matrix vector multiplication	2100 MHz	1849 MHz	11.95 %
Vector operation	1700 MHz	1600 MHz	5.88 %
N-body	2000 MHz	1700 MHz	15.00 %
FFT	1600 MHz	1434 MHz	10.38 %
SparseLU	1800 MHz	1933 MHz	7.39 %
NQueens	1800 MHz	1600 MHz	11.11 %
Strassen	1800 MHz	1600 MHz	11.11 %
Black Scholes	1800 MHz	1927 MHz	7.06 %
Fibonacci	1400 MHz	1738 MHz	24.14 %
Quick Sort	2100 MHz	1964 MHz	6.48 %
Unstructured 3d stencil	1200 MHz	1360 MHz	13.33 %



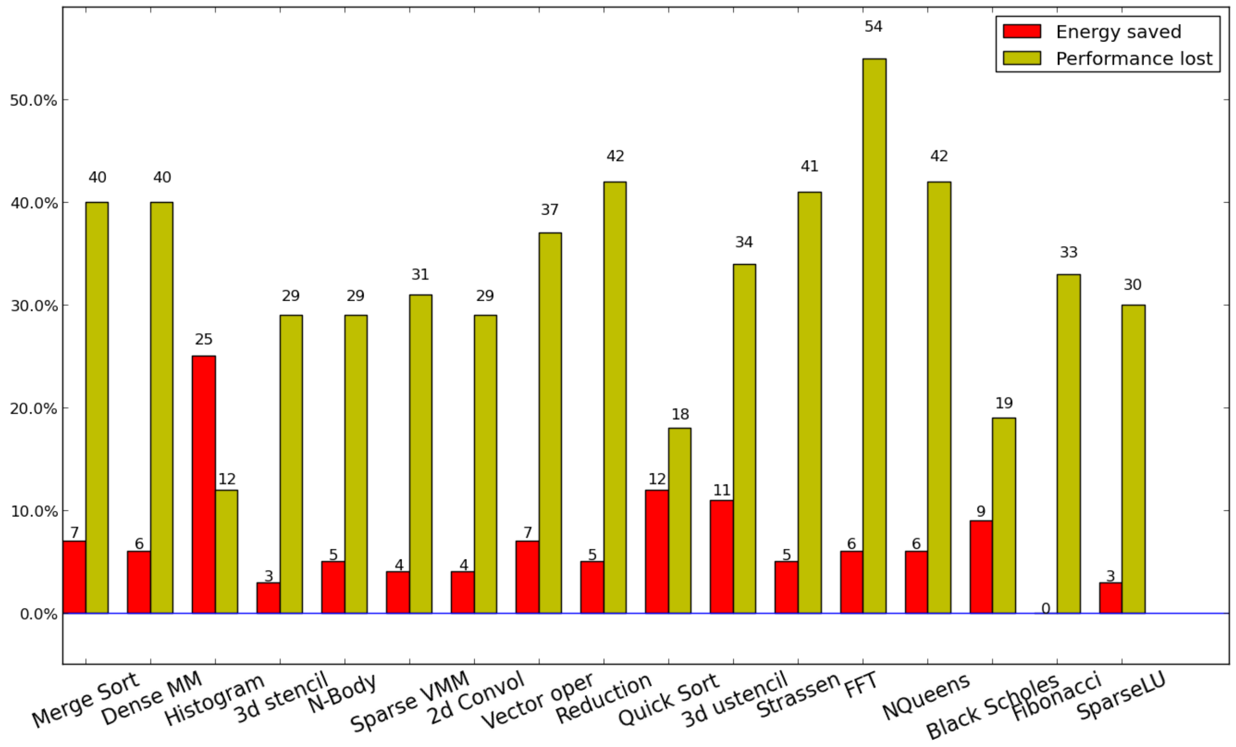
0.0.8 Results for 15 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	1700 MHz	1800 MHz	5.88 %
Reduction	1800 MHz	1803 MHz	0.17 %
Histogram	1300 MHz	1756 MHz	35.08 %
2d convolution	2000 MHz	2018 MHz	0.90 %
3d stencil	1800 MHz	2016 MHz	12.00 %
Dense matrix multiplication	1700 MHz	1808 MHz	6.35 %
Sparse matrix vector multiplication	2000 MHz	1973 MHz	1.35 %
Vector operation	1700 MHz	1800 MHz	5.88 %
N-body	2400 MHz	2006 MHz	16.42 %
FFT	1400 MHz	1509 MHz	7.79 %
SparseLU	1800 MHz	1844 MHz	2.44 %
NQueens	1800 MHz	1800 MHz	0.00 %
Strassen	1800 MHz	1800 MHz	0.00 %
Black Scholes	1900 MHz	1982 MHz	4.32 %
Fibonacci	2100 MHz	1950 MHz	7.14 %
Quick Sort	2100 MHz	1967 MHz	6.33 %
Unstructured 3d stencil	1400 MHz	1596 MHz	14.00 %



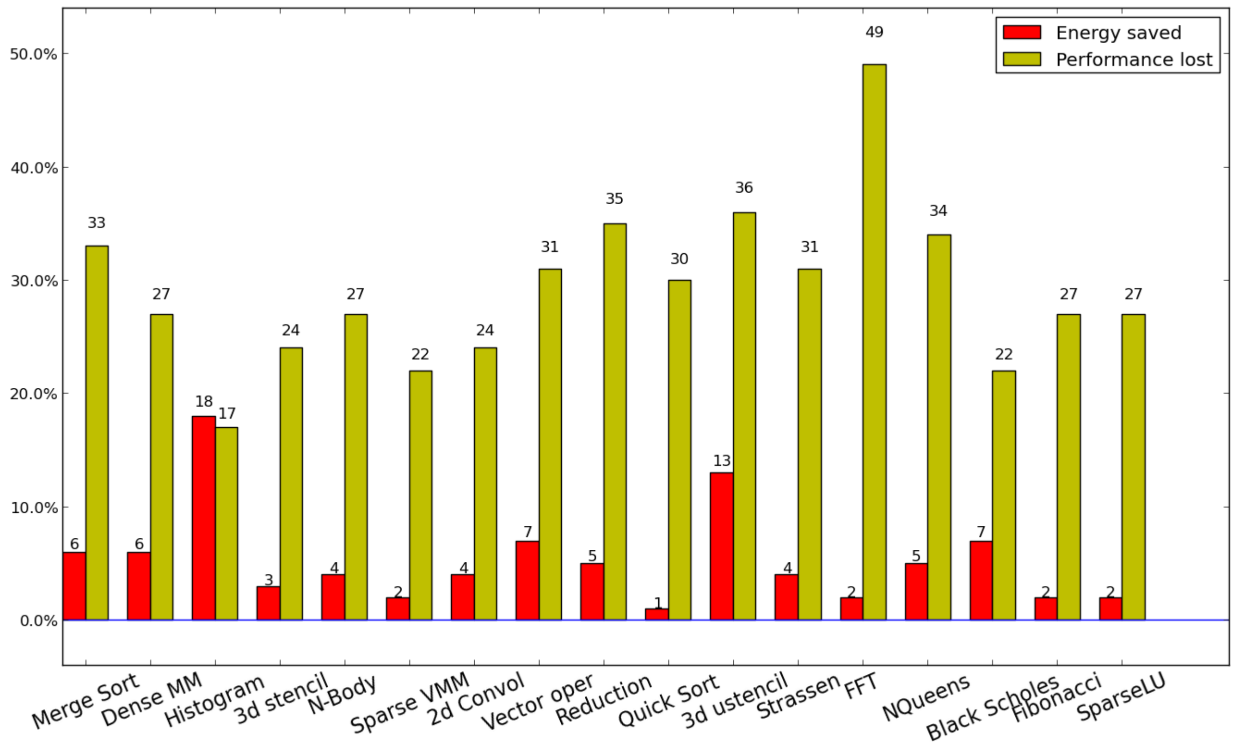
0.0.9 Results for 14 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	1900 MHz	1800 MHz	5.26 %
Reduction	1900 MHz	1808 MHz	4.84 %
Histogram	1300 MHz	1707 MHz	31.31 %
2d convolution	2100 MHz	2018 MHz	3.90 %
3d stencil	1800 MHz	2004 MHz	11.33 %
Dense matrix multiplication	1800 MHz	1818 MHz	1.00 %
Sparse matrix vector multiplication	1900 MHz	1985 MHz	4.47 %
Vector operation	1700 MHz	1800 MHz	5.88 %
N-body	2000 MHz	2006 MHz	0.30 %
FFT	1700 MHz	1539 MHz	9.47 %
SparseLU	2000 MHz	2005 MHz	0.25 %
NQueens	2000 MHz	1800 MHz	10.00 %
Strassen	1800 MHz	1800 MHz	0.00 %
Black Scholes	1900 MHz	2006 MHz	5.58 %
Fibonacci	2200 MHz	1940 MHz	11.82 %
Quick Sort	2100 MHz	1941 MHz	7.57 %
Unstructured 3d stencil	1500 MHz	1654 MHz	10.27 %



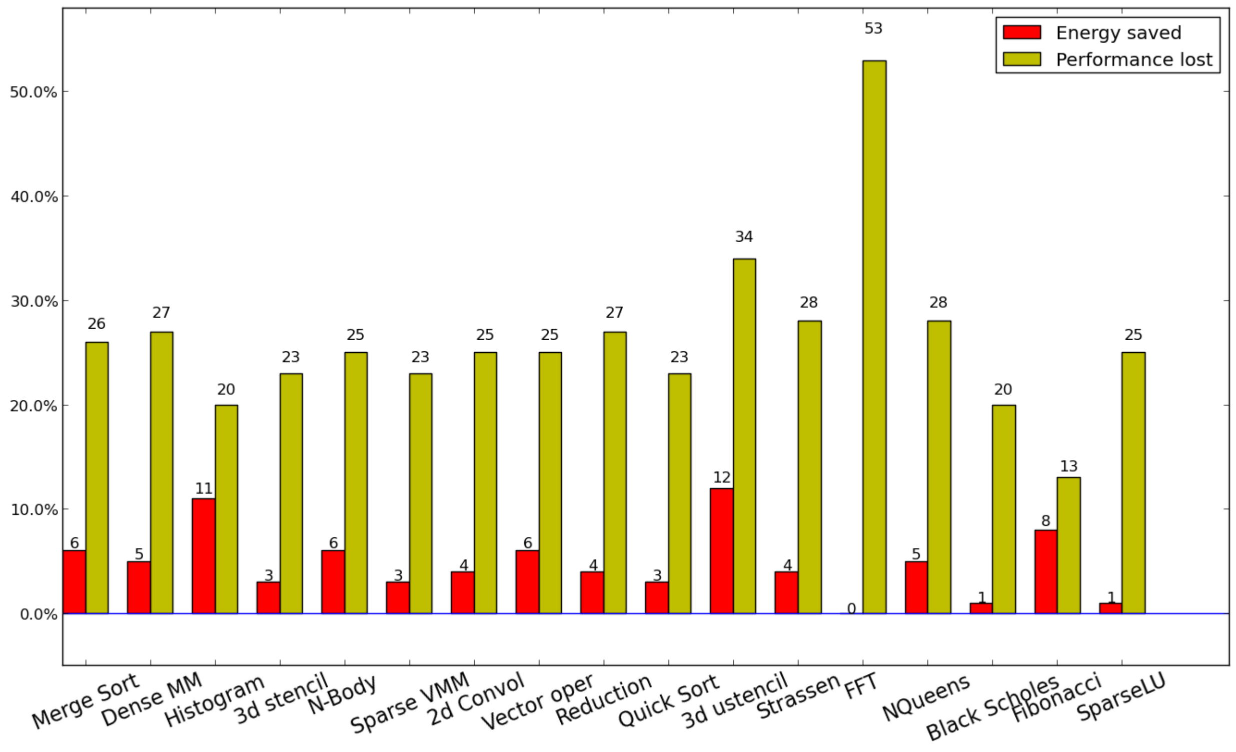
0.0.10 Results for 13 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	1900 MHz	1900 MHz	0.00 %
Reduction	1900 MHz	1906 MHz	0.32 %
Histogram	1400 MHz	1823 MHz	30.21 %
2d convolution	2100 MHz	2092 MHz	0.38 %
3d stencil	2000 MHz	2108 MHz	5.40 %
Dense matrix multiplication	1800 MHz	2000 MHz	11.11 %
Sparse matrix vector multiplication	2000 MHz	2137 MHz	6.85 %
Vector operation	1700 MHz	1900 MHz	11.76 %
N-body	2000 MHz	2017 MHz	0.85 %
FFT	1500 MHz	1599 MHz	6.60 %
SparseLU	2000 MHz	2052 MHz	2.60 %
NQueens	1900 MHz	1900 MHz	0.00 %
Strassen	2000 MHz	1900 MHz	5.00 %
Black Scholes	2400 MHz	1995 MHz	16.88 %
Fibonacci	2000 MHz	2037 MHz	1.85 %
Quick Sort	2100 MHz	2023 MHz	3.67 %
Unstructured 3d stencil	1500 MHz	1560 MHz	4.00 %



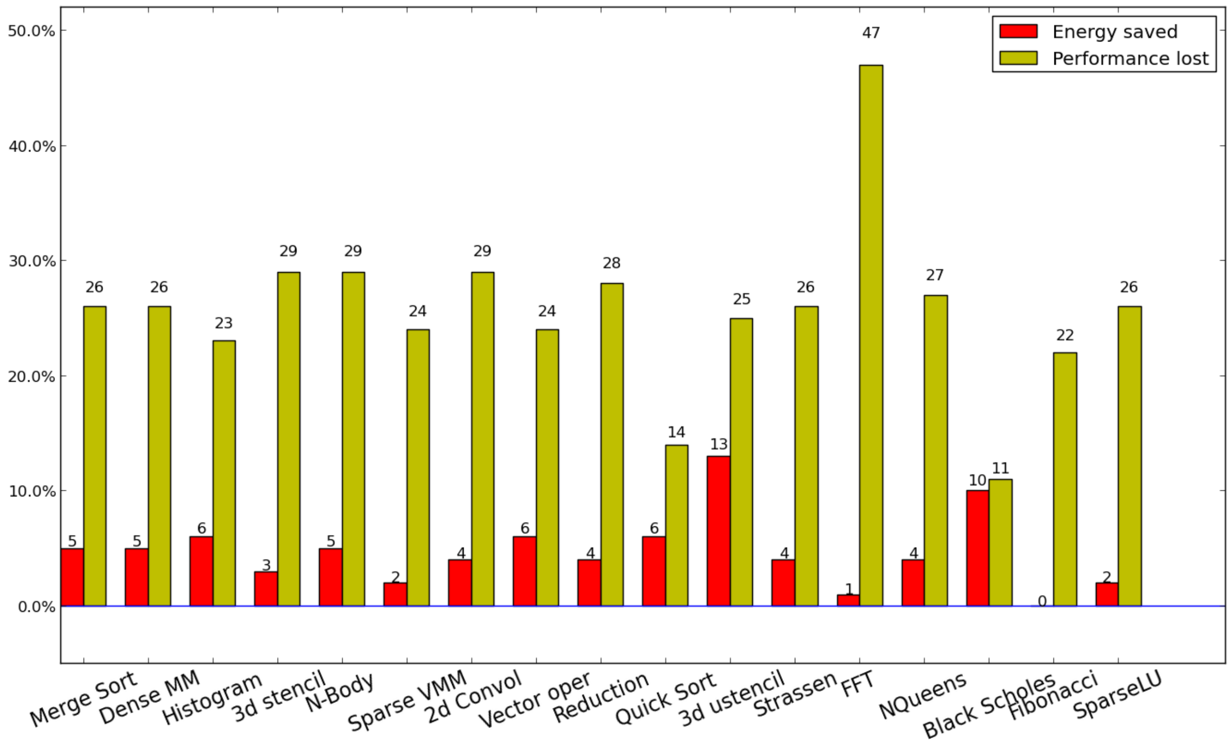
0.0.11 Results for 12 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2000 MHz	2000 MHz	0.00 %
Reduction	1900 MHz	2019 MHz	6.26 %
Histogram	1400 MHz	1892 MHz	35.14 %
2d convolution	2000 MHz	2061 MHz	3.05 %
3d stencil	1900 MHz	2083 MHz	9.63 %
Dense matrix multiplication	2000 MHz	2010 MHz	0.50 %
Sparse matrix vector multiplication	2000 MHz	2104 MHz	5.20 %
Vector operation	1700 MHz	2000 MHz	17.65 %
N-body	2000 MHz	2060 MHz	3.00 %
FFT	1700 MHz	1644 MHz	3.29 %
SparseLU	1900 MHz	2093 MHz	10.16 %
NQueens	2000 MHz	2000 MHz	0.00 %
Strassen	1800 MHz	2000 MHz	11.11 %
Black Scholes	1900 MHz	2132 MHz	12.21 %
Fibonacci	2300 MHz	2051 MHz	10.83 %
Quick Sort	2100 MHz	2103 MHz	0.14 %
Unstructured 3d stencil	1500 MHz	1612 MHz	7.47 %



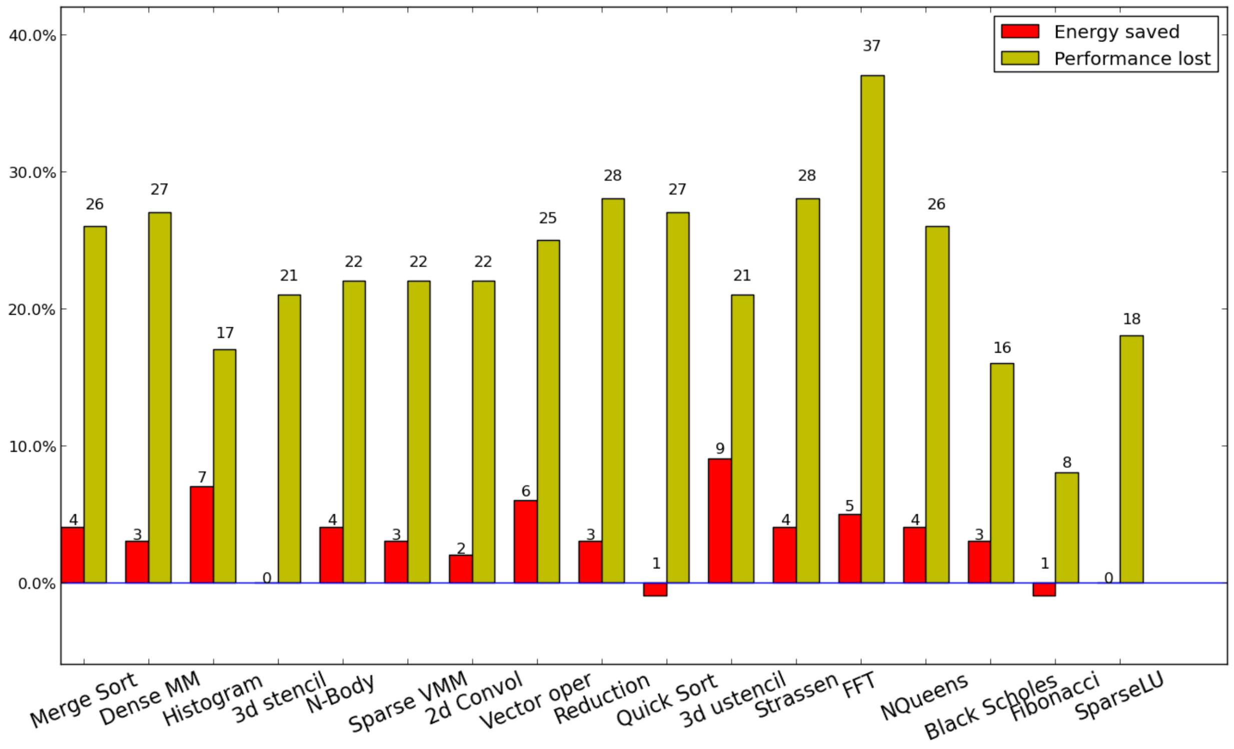
0.0.12 Results for 11 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2000 MHz	2000 MHz	0.00 %
Reduction	1900 MHz	2004 MHz	5.47 %
Histogram	1700 MHz	1921 MHz	13.00 %
2d convolution	2000 MHz	2000 MHz	0.00 %
3d stencil	2000 MHz	2000 MHz	0.00 %
Dense matrix multiplication	2000 MHz	2009 MHz	0.45 %
Sparse matrix vector multiplication	2200 MHz	2094 MHz	4.82 %
Vector operation	1800 MHz	2000 MHz	11.11 %
N-body	2000 MHz	2000 MHz	0.00 %
FFT	1700 MHz	1594 MHz	6.24 %
SparseLU	2000 MHz	2056 MHz	2.80 %
NQueens	2100 MHz	2000 MHz	4.76 %
Strassen	1900 MHz	2000 MHz	5.26 %
Black Scholes	1700 MHz	2084 MHz	22.59 %
Fibonacci	2300 MHz	2110 MHz	8.26 %
Quick Sort	2200 MHz	2102 MHz	4.45 %
Unstructured 3d stencil	1200 MHz	1754 MHz	46.17 %



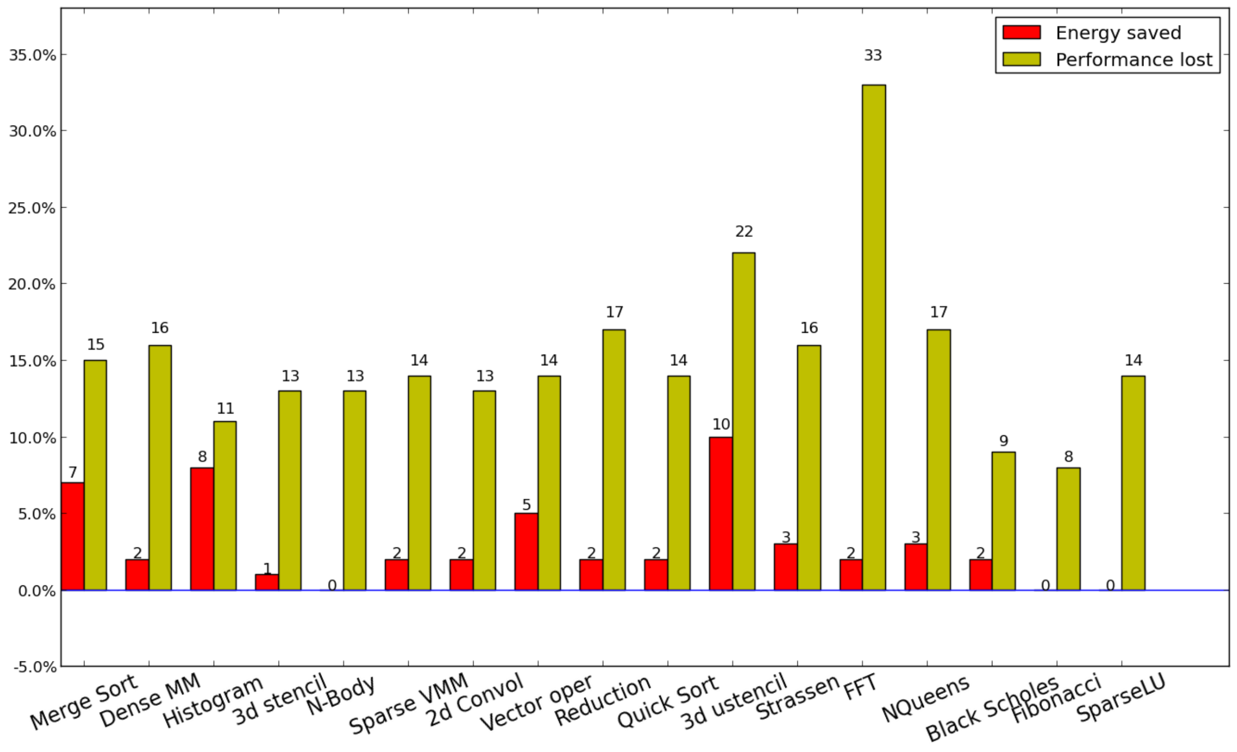
0.0.13 Results for 10 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2000 MHz	2000 MHz	0.00 %
Reduction	1900 MHz	2020 MHz	6.32 %
Histogram	1500 MHz	1920 MHz	28.00 %
2d convolution	2200 MHz	2120 MHz	3.64 %
3d stencil	2100 MHz	2126 MHz	1.24 %
Dense matrix multiplication	2000 MHz	2008 MHz	0.40 %
Sparse matrix vector multiplication	2200 MHz	2131 MHz	3.14 %
Vector operation	1800 MHz	2000 MHz	11.11 %
N-body	2200 MHz	2123 MHz	3.50 %
FFT	2000 MHz	1757 MHz	12.15 %
SparseLU	2100 MHz	2214 MHz	5.43 %
NQueens	2000 MHz	2000 MHz	0.00 %
Strassen	2100 MHz	2000 MHz	4.76 %
Black Scholes	2000 MHz	2149 MHz	7.45 %
Fibonacci	2600 MHz	2400 MHz	7.69 %
Quick Sort	2300 MHz	2086 MHz	9.30 %
Unstructured 3d stencil	1500 MHz	1878 MHz	25.20 %



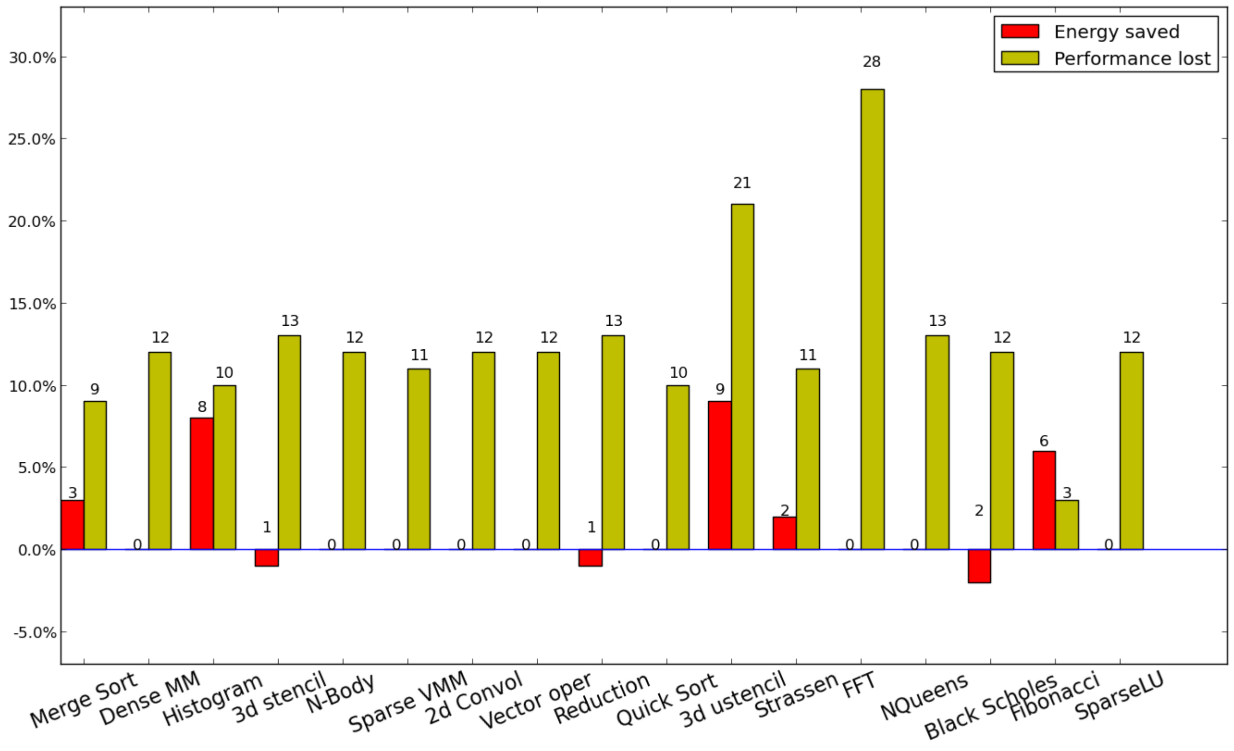
0.0.14 Results for 9 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2200 MHz	2200 MHz	0.00 %
Reduction	2100 MHz	2207 MHz	5.10 %
Histogram	1700 MHz	1951 MHz	14.76 %
2d convolution	2000 MHz	2286 MHz	14.30 %
3d stencil	2200 MHz	2292 MHz	4.18 %
Dense matrix multiplication	2000 MHz	2205 MHz	10.25 %
Sparse matrix vector multiplication	2300 MHz	2282 MHz	0.78 %
Vector operation	2000 MHz	2200 MHz	10.00 %
N-body	2200 MHz	2288 MHz	4.00 %
FFT	1700 MHz	1775 MHz	4.41 %
SparseLU	2400 MHz	2284 MHz	4.83 %
NQueens	2200 MHz	2200 MHz	0.00 %
Strassen	2100 MHz	2150 MHz	2.38 %
Black Scholes	2400 MHz	2302 MHz	4.08 %
Fibonacci	2400 MHz	2385 MHz	0.62 %
Quick Sort	2600 MHz	2254 MHz	13.31 %
Unstructured 3d stencil	1500 MHz	1813 MHz	20.87 %



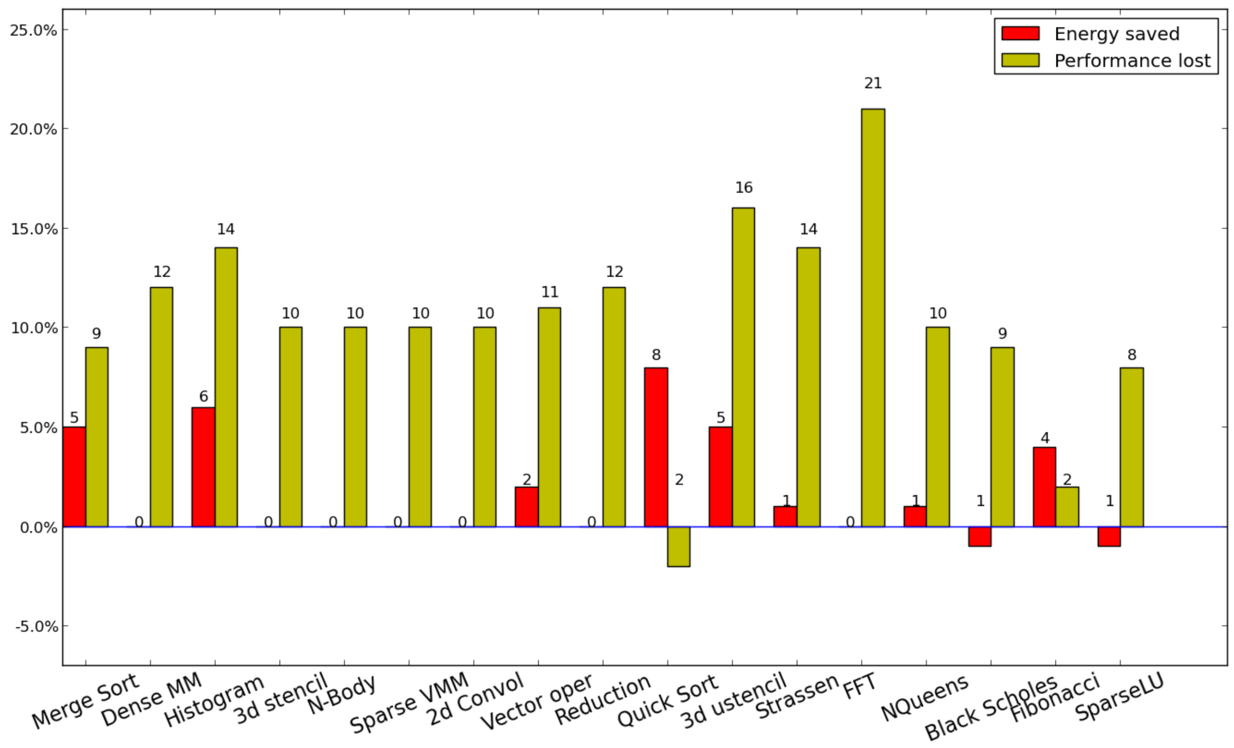
0.0.15 Results for 8 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2300 MHz	11.54 %
Reduction	2500 MHz	2300 MHz	8.00 %
Histogram	1700 MHz	1901 MHz	11.82 %
2d convolution	2500 MHz	2300 MHz	8.00 %
3d stencil	2400 MHz	2300 MHz	4.17 %
Dense matrix multiplication	2200 MHz	2300 MHz	4.55 %
Sparse matrix vector multiplication	2300 MHz	2329 MHz	1.26 %
Vector operation	2200 MHz	2300 MHz	4.55 %
N-body	2400 MHz	2303 MHz	4.04 %
FFT	1900 MHz	1861 MHz	2.05 %
SparseLU	2600 MHz	2318 MHz	10.85 %
NQueens	2600 MHz	2300 MHz	11.54 %
Strassen	2500 MHz	2300 MHz	8.00 %
Black Scholes	2600 MHz	2327 MHz	10.50 %
Fibonacci	2600 MHz	2331 MHz	10.35 %
Quick Sort	2500 MHz	2335 MHz	6.60 %
Unstructured 3d stencil	1800 MHz	1800 MHz	0.00 %



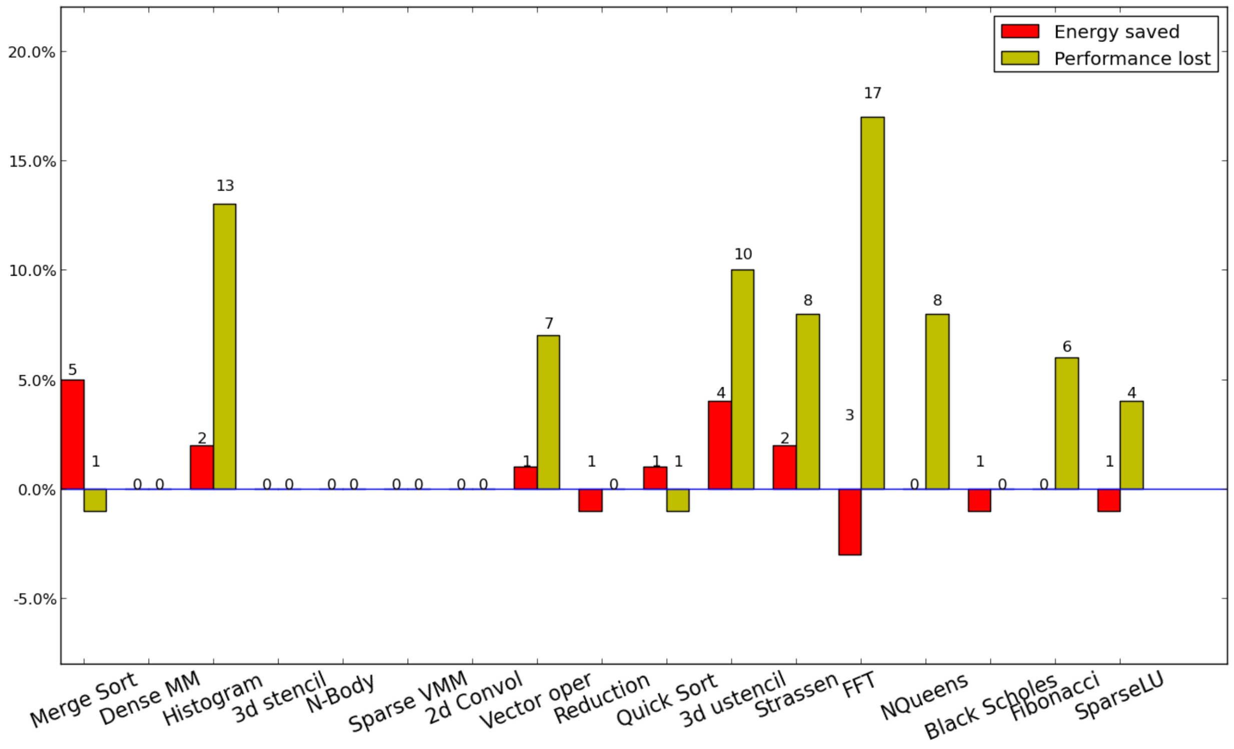
0.0.16 Results for 7 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2500 MHz	2300 MHz	8.00 %
Reduction	2400 MHz	2325 MHz	3.12 %
Histogram	1900 MHz	1924 MHz	1.26 %
2d convolution	2500 MHz	2353 MHz	5.88 %
3d stencil	2600 MHz	2353 MHz	9.50 %
Dense matrix multiplication	2400 MHz	2306 MHz	3.92 %
Sparse matrix vector multiplication	2500 MHz	2355 MHz	5.80 %
Vector operation	2200 MHz	2300 MHz	4.55 %
N-body	2600 MHz	2355 MHz	9.42 %
FFT	1900 MHz	1876 MHz	1.26 %
SparseLU	2600 MHz	2425 MHz	6.73 %
NQueens	2500 MHz	2300 MHz	8.00 %
Strassen	2300 MHz	2300 MHz	0.00 %
Black Scholes	2600 MHz	2368 MHz	8.92 %
Fibonacci	2600 MHz	2375 MHz	8.65 %
Quick Sort	2400 MHz	2352 MHz	2.00 %
Unstructured 3d stencil	1700 MHz	1947 MHz	14.53 %



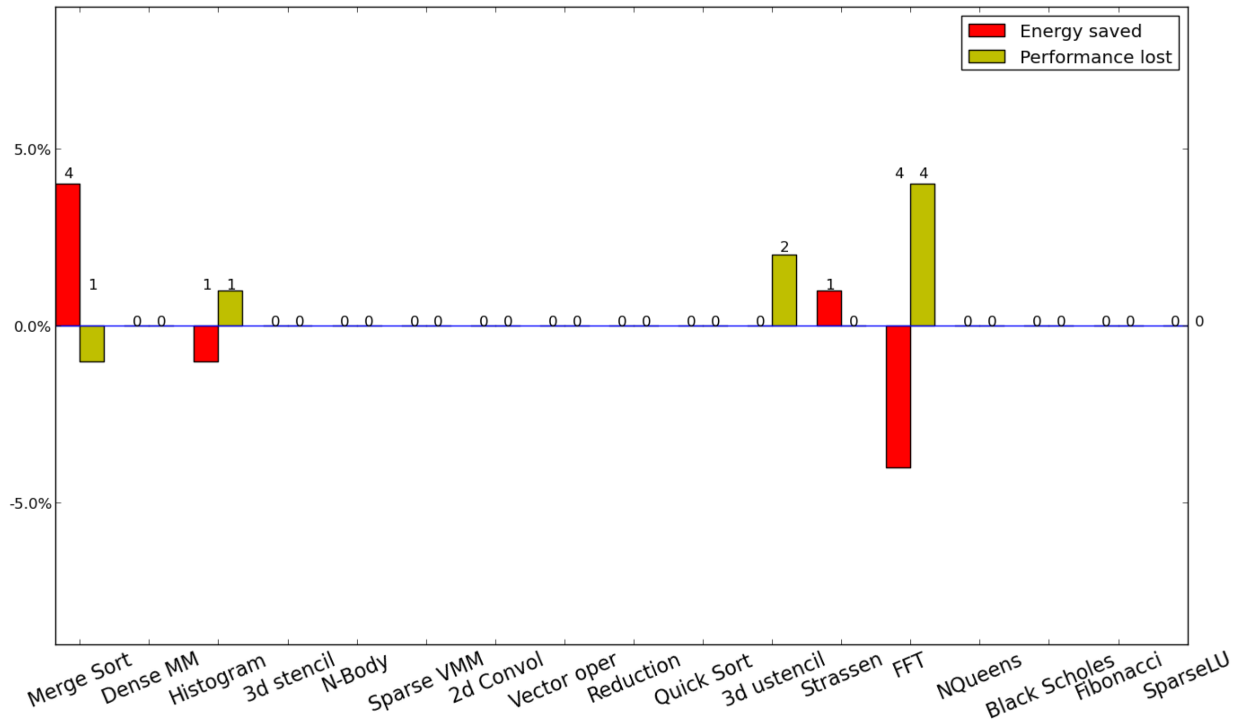
0.0.17 Results for 6 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2500 MHz	2592 MHz	3.68 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	1800 MHz	2098 MHz	16.56 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2500 MHz	2600 MHz	4.00 %
Dense matrix multiplication	2300 MHz	2600 MHz	13.04 %
Sparse matrix vector multiplication	2500 MHz	2600 MHz	4.00 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2100 MHz	2094 MHz	0.29 %
SparseLU	2500 MHz	2511 MHz	0.44 %
NQueens	2600 MHz	2400 MHz	7.69 %
Strassen	2600 MHz	2400 MHz	7.69 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2427 MHz	6.65 %
Quick Sort	2100 MHz	2598 MHz	23.71 %
Unstructured 3d stencil	1900 MHz	2104 MHz	10.74 %



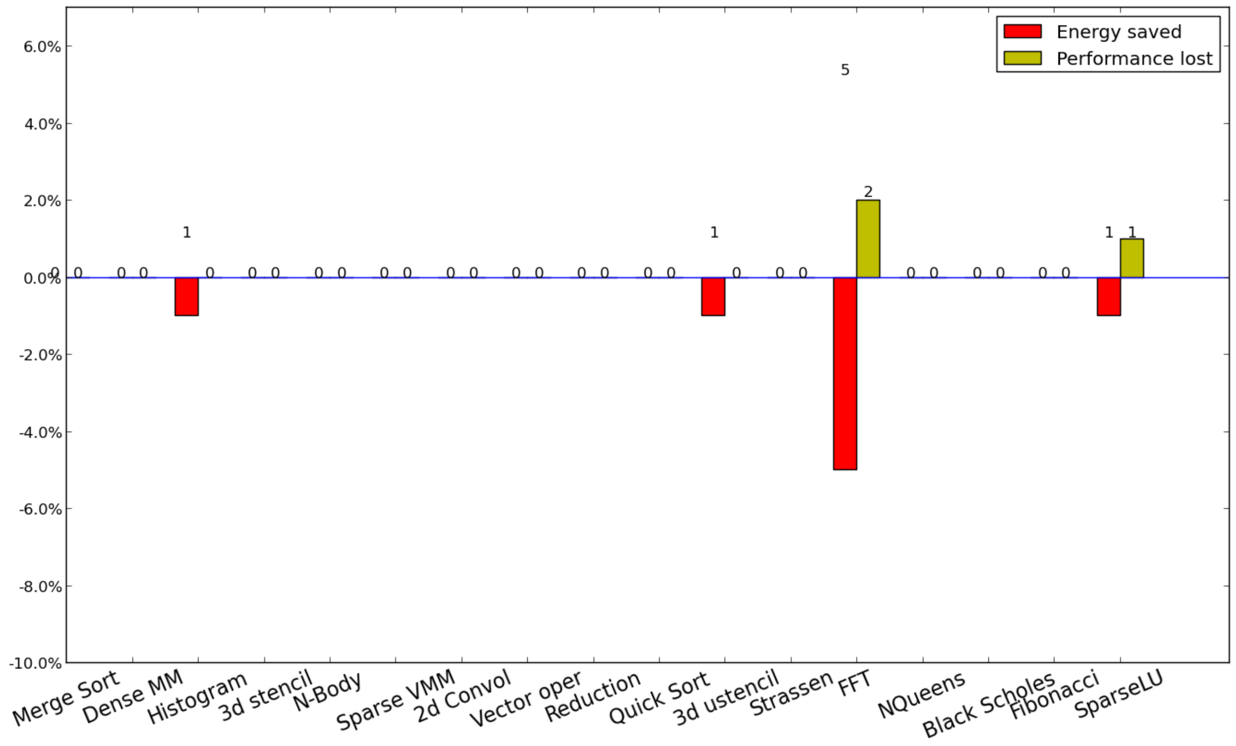
0.0.18 Results for 5 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2590 MHz	0.38 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2000 MHz	2515 MHz	25.75 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2507 MHz	4.46 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2500 MHz	2595 MHz	3.80 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2200 MHz	2600 MHz	18.18 %
Unstructured 3d stencil	2000 MHz	2513 MHz	25.65 %



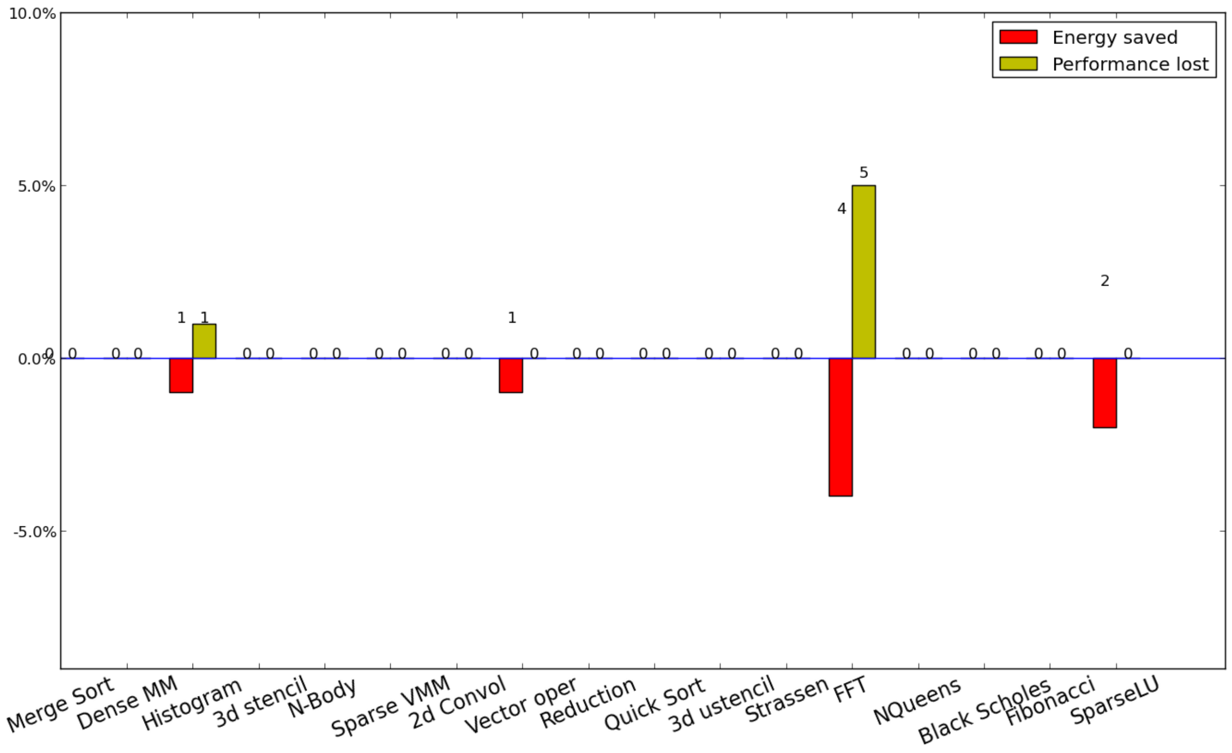
0.0.19 Results for 4 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2500 MHz	2600 MHz	4.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2500 MHz	2600 MHz	4.00 %
Quick Sort	2400 MHz	2600 MHz	8.33 %
Unstructured 3d stencil	1900 MHz	2600 MHz	36.84 %



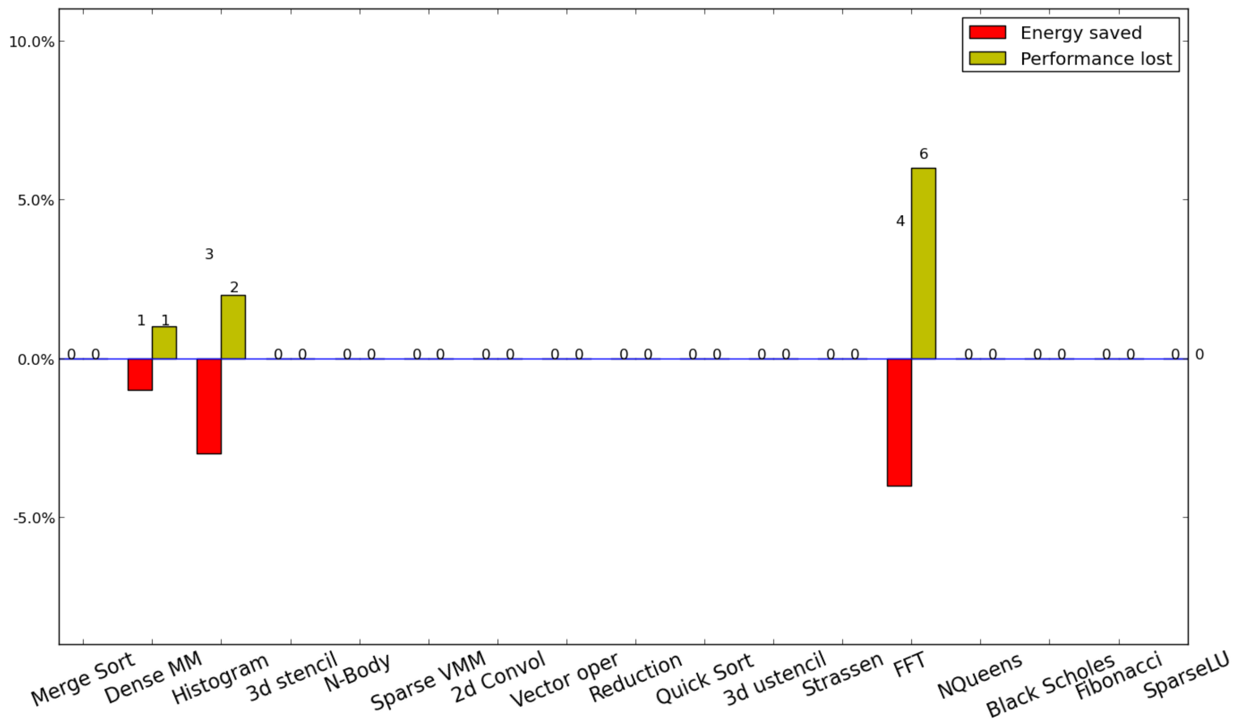
0.0.20 Results for 3 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2100 MHz	2600 MHz	23.81 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2100 MHz	2600 MHz	23.81 %



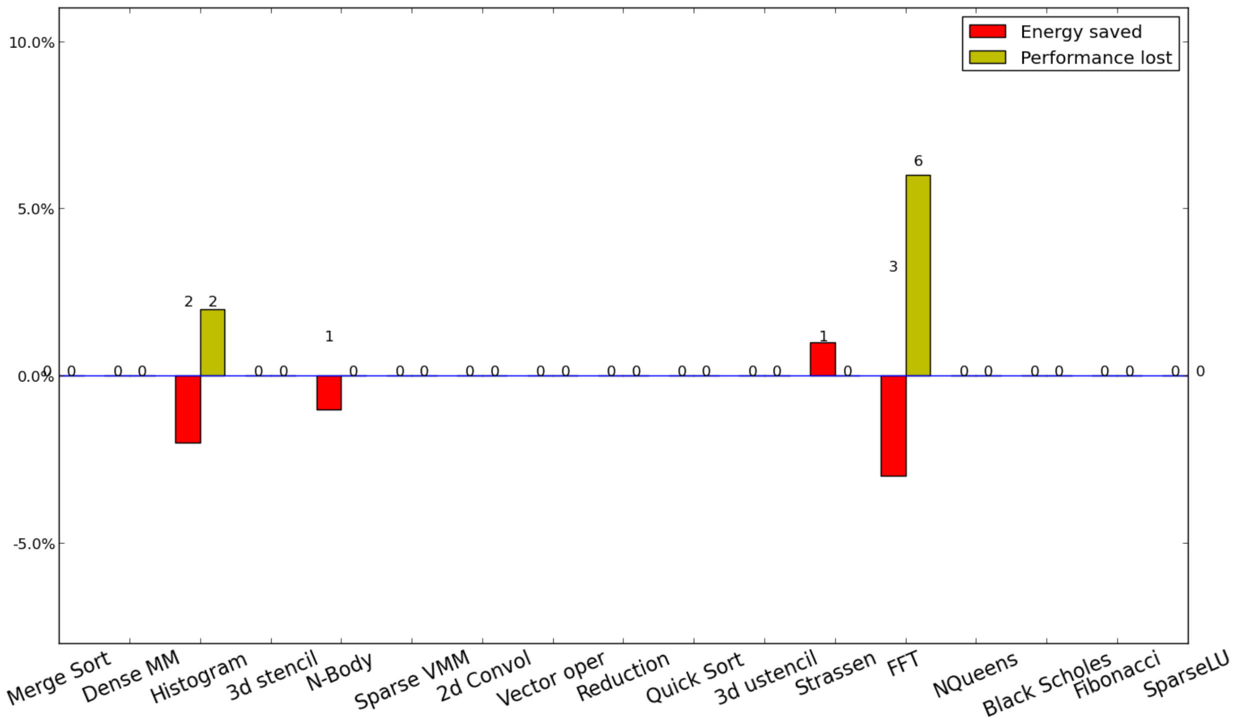
0.0.21 Results for 2 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2200 MHz	2600 MHz	18.18 %



0.0.22 Results for 1 thread

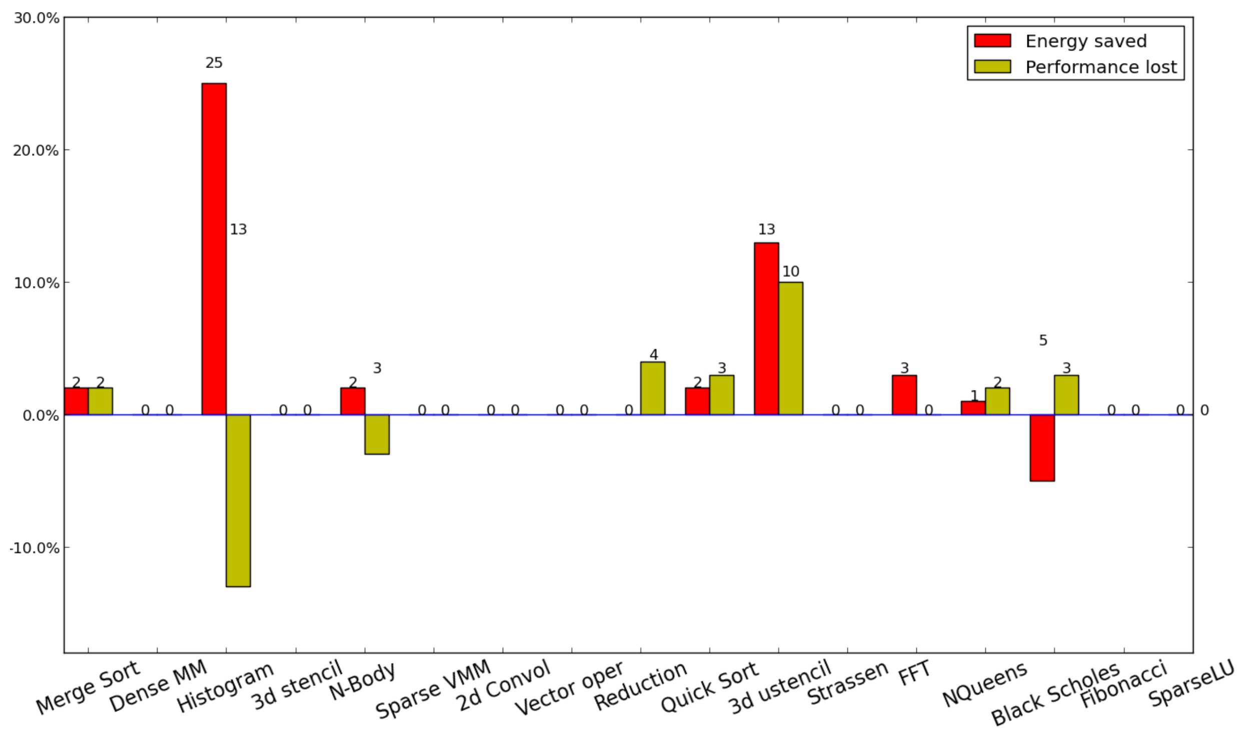
Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2500 MHz	2600 MHz	4.00 %



Results for the metric Energy Delay Product

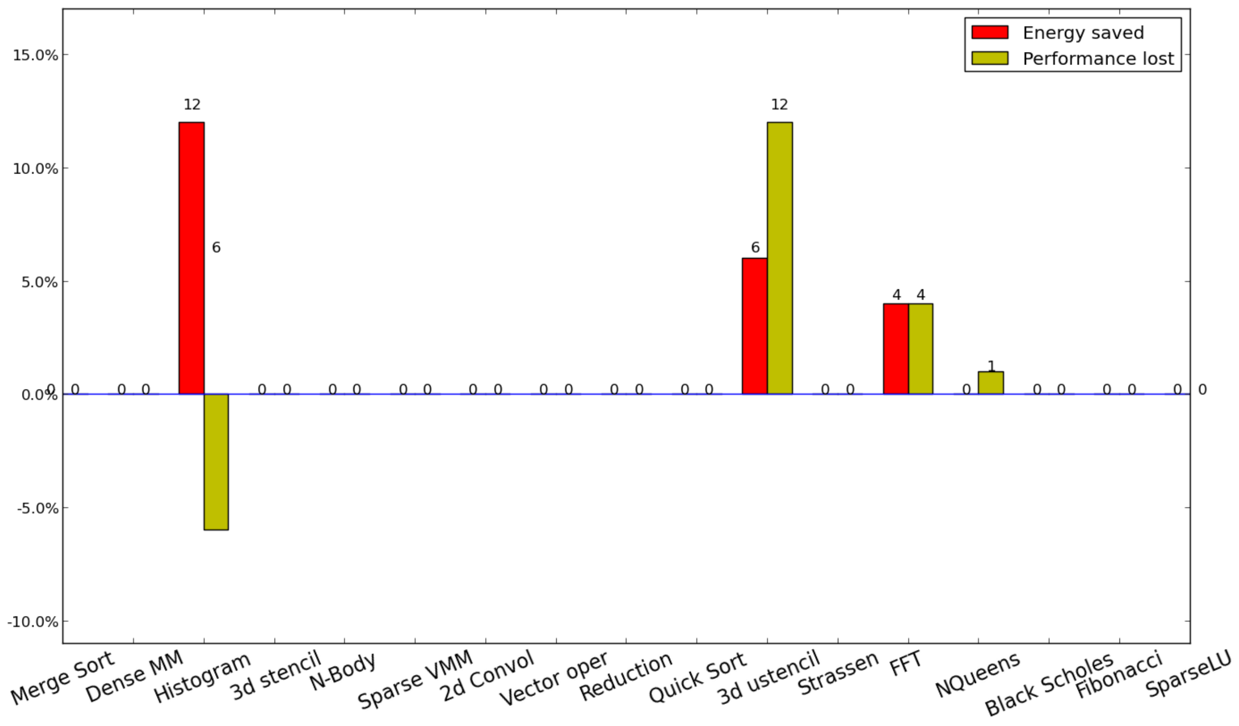
0.0.23 Results for 16 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2490 MHz	4.23 %
Reduction	2600 MHz	2505 MHz	3.65 %
Histogram	2000 MHz	2224 MHz	11.20 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2500 MHz	2598 MHz	3.92 %
FFT	2500 MHz	2389 MHz	4.44 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2549 MHz	1.96 %
Strassen	2400 MHz	2600 MHz	8.33 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2462 MHz	5.31 %
Unstructured 3d stencil	2100 MHz	2067 MHz	1.57 %



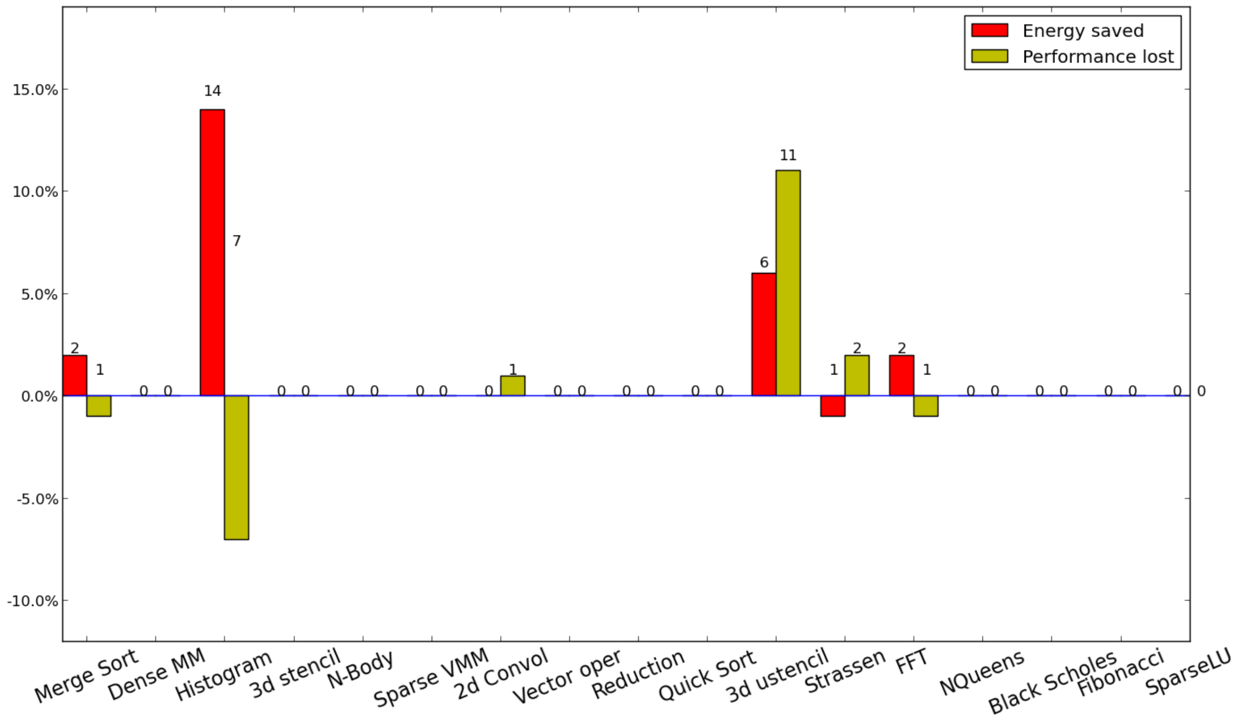
0.0.24 Results for 15 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	1900 MHz	2272 MHz	19.58 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2596 MHz	0.15 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2350 MHz	2.08 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2555 MHz	1.73 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2100 MHz	2600 MHz	23.81 %
Quick Sort	2500 MHz	2600 MHz	4.00 %
Unstructured 3d stencil	2100 MHz	2181 MHz	3.86 %



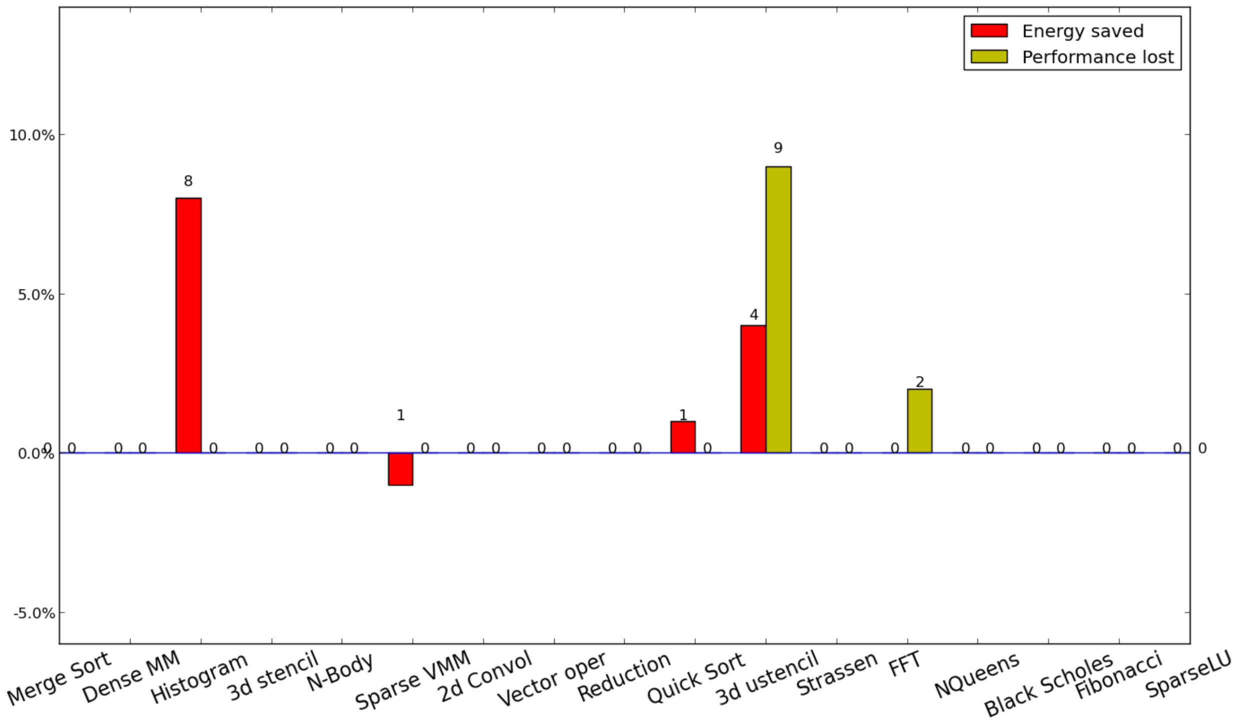
0.0.25 Results for 14 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2595 MHz	0.19 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2100 MHz	2357 MHz	12.24 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2395 MHz	0.21 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2594 MHz	0.23 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2200 MHz	2600 MHz	18.18 %
Quick Sort	2500 MHz	2600 MHz	4.00 %
Unstructured 3d stencil	2200 MHz	2183 MHz	0.77 %



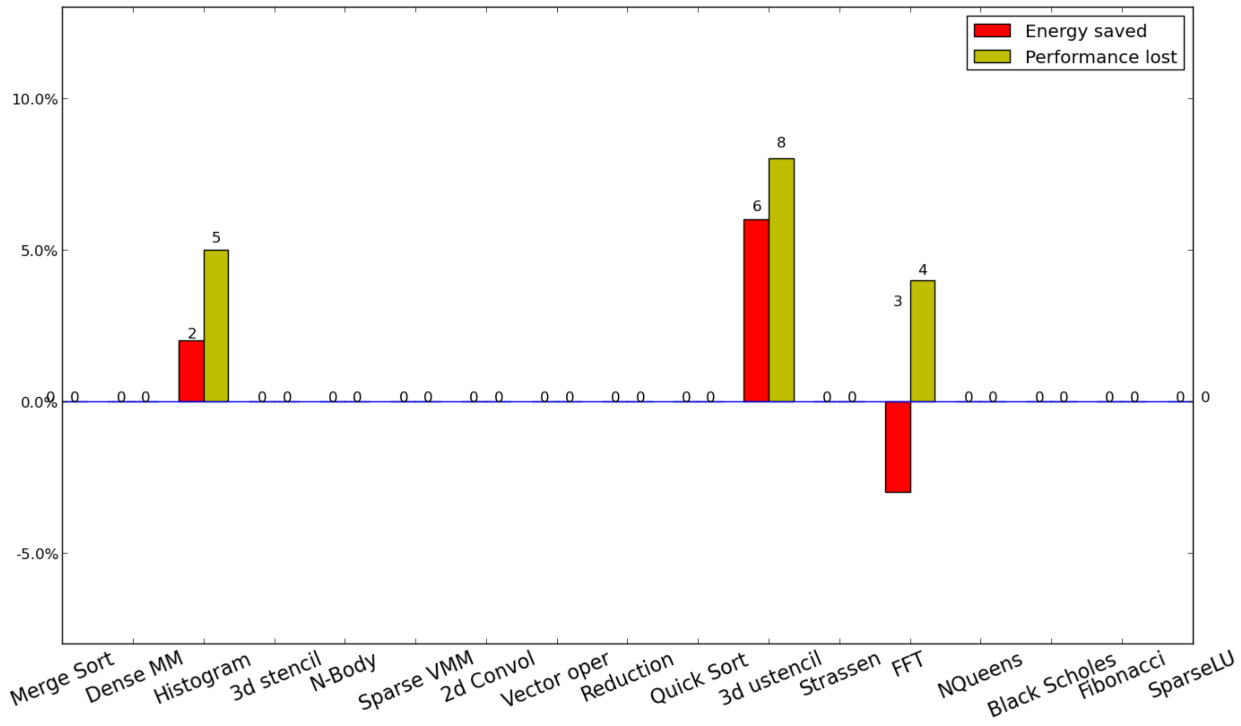
0.0.26 Results for 13 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2000 MHz	2411 MHz	20.55 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2400 MHz	7.69 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2572 MHz	1.08 %
Unstructured 3d stencil	2300 MHz	2270 MHz	1.30 %



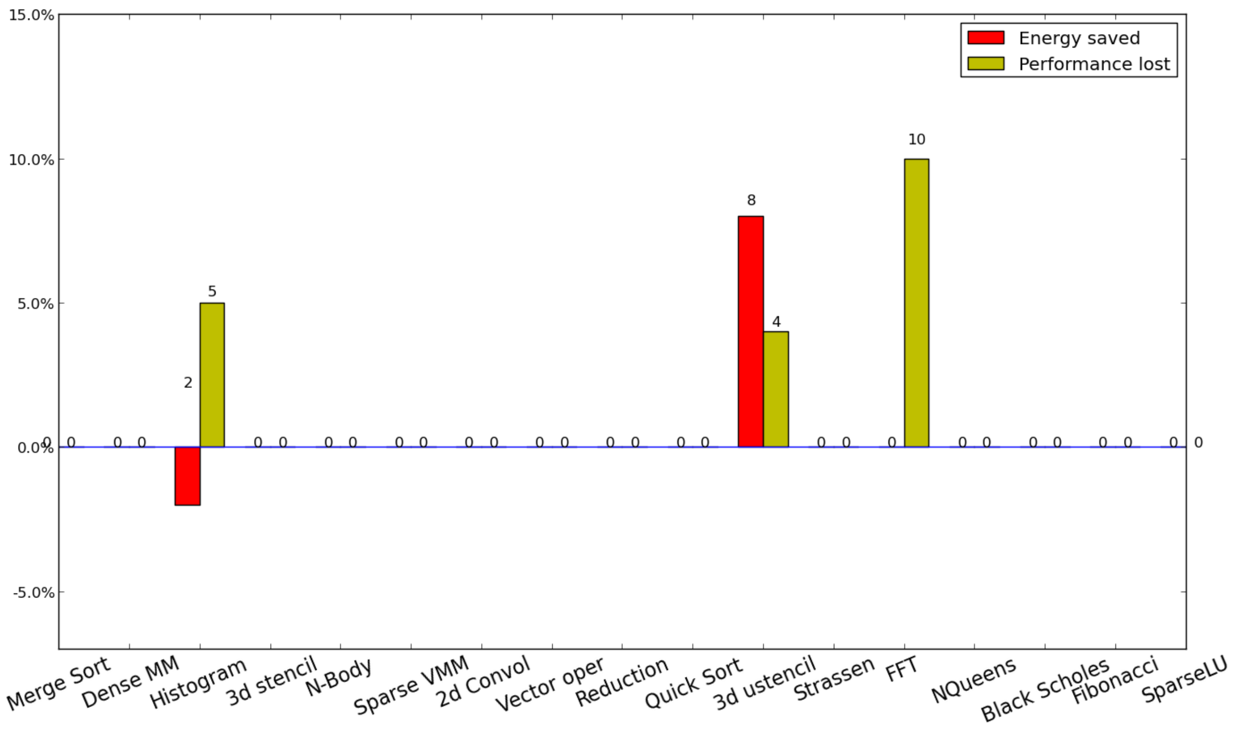
0.0.27 Results for 12 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2500 MHz	2434 MHz	2.64 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2400 MHz	7.69 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2300 MHz	2600 MHz	13.04 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2400 MHz	2248 MHz	6.33 %



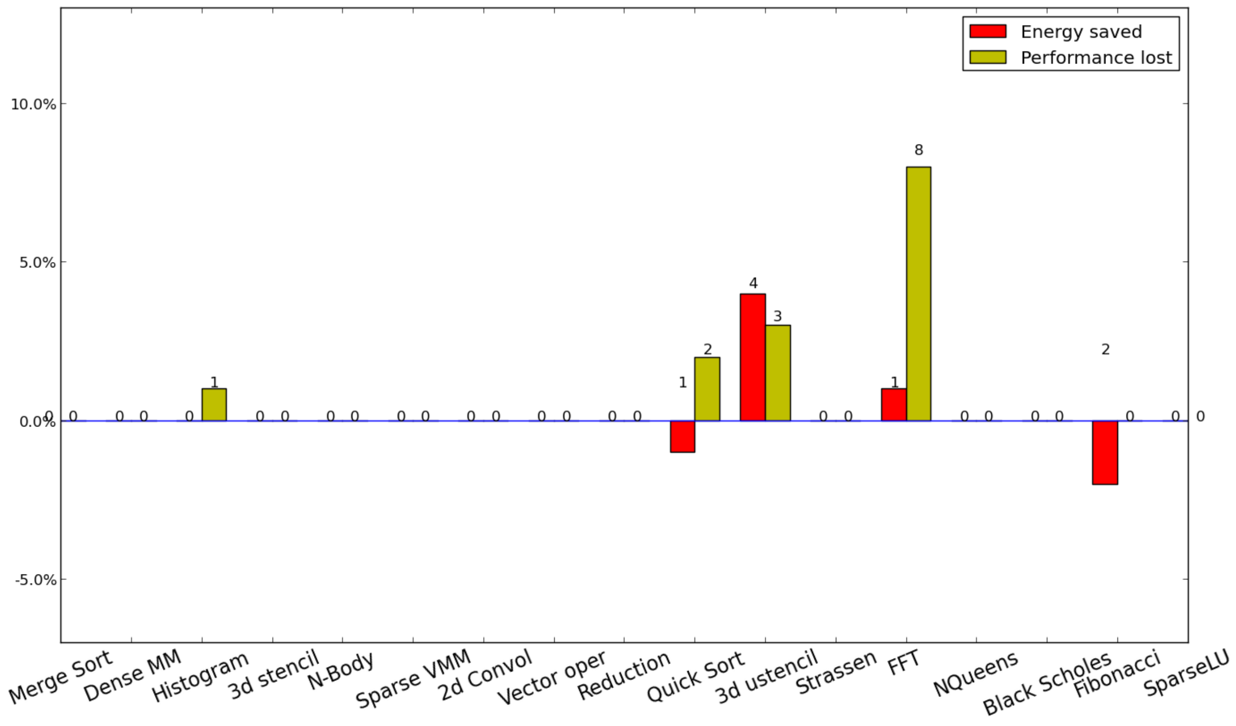
0.0.28 Results for 11 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2400 MHz	2445 MHz	1.88 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2500 MHz	2276 MHz	8.96 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2500 MHz	2600 MHz	4.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2000 MHz	2300 MHz	15.00 %



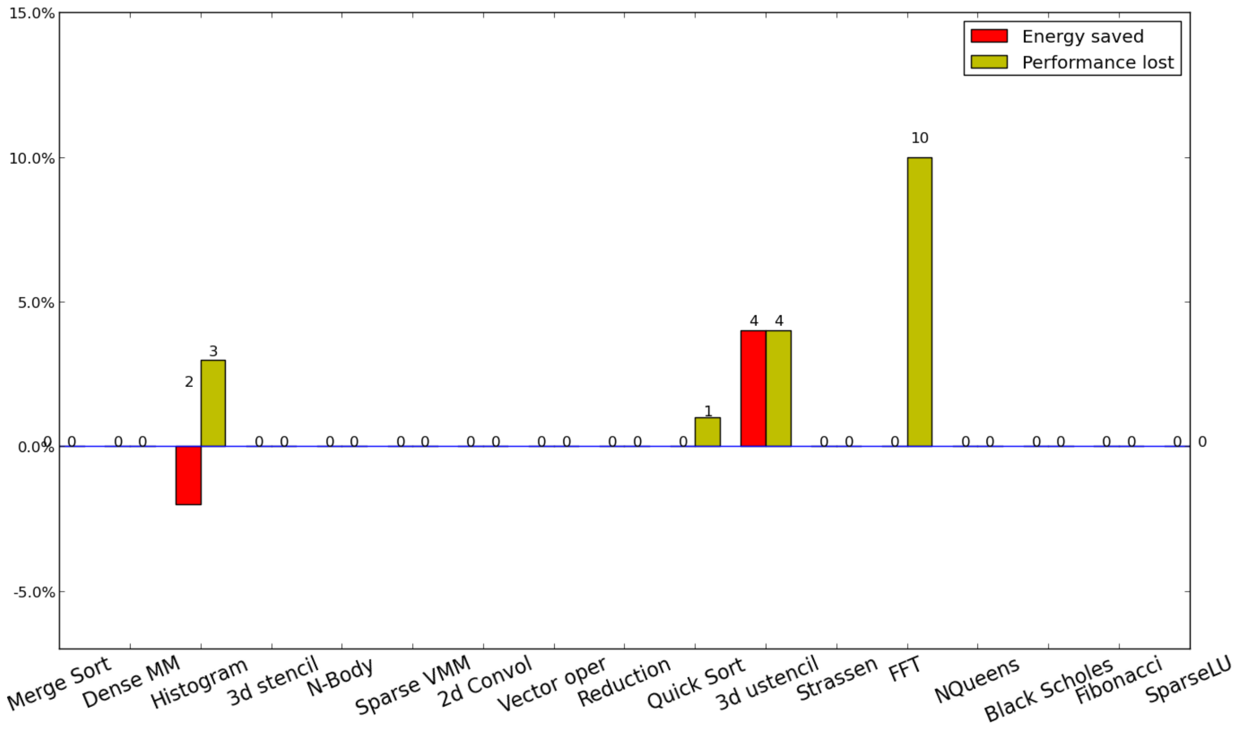
0.0.29 Results for 10 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2200 MHz	2439 MHz	10.86 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2341 MHz	2.46 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2200 MHz	2381 MHz	8.23 %



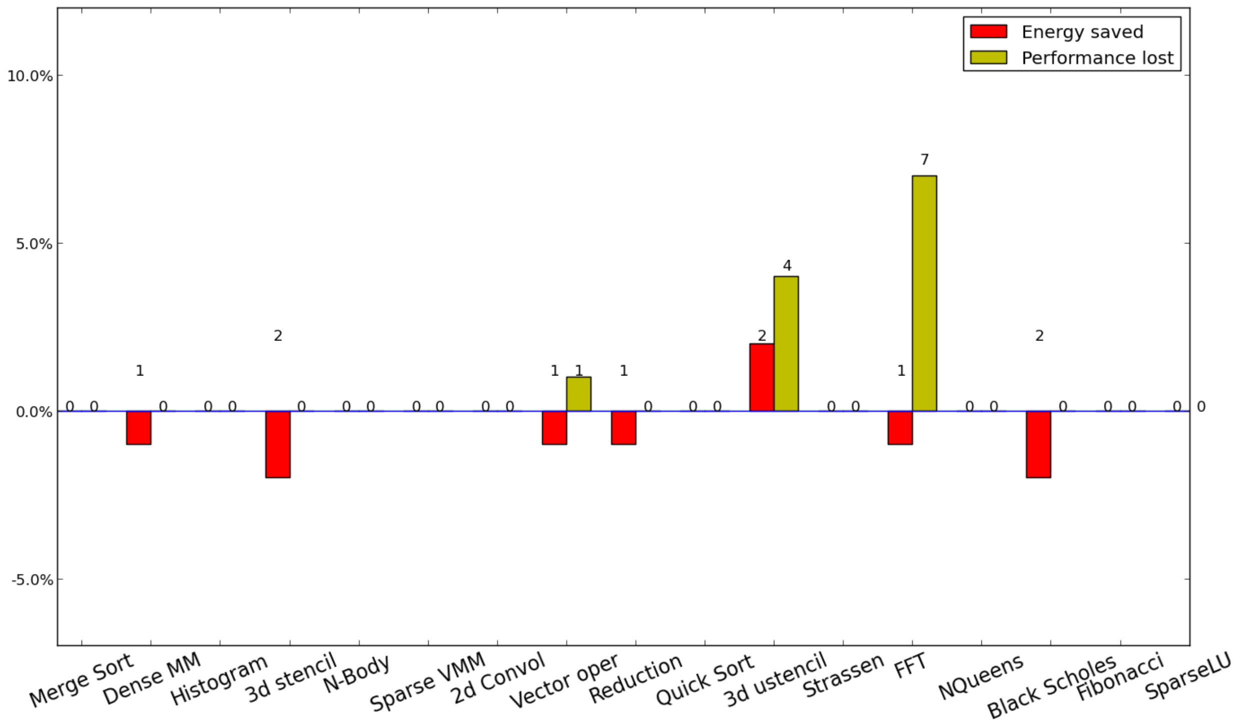
0.0.30 Results for 9 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2500 MHz	2412 MHz	3.52 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2500 MHz	2339 MHz	6.44 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2400 MHz	2600 MHz	8.33 %
Fibonacci	2400 MHz	2600 MHz	8.33 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2400 MHz	2340 MHz	2.50 %



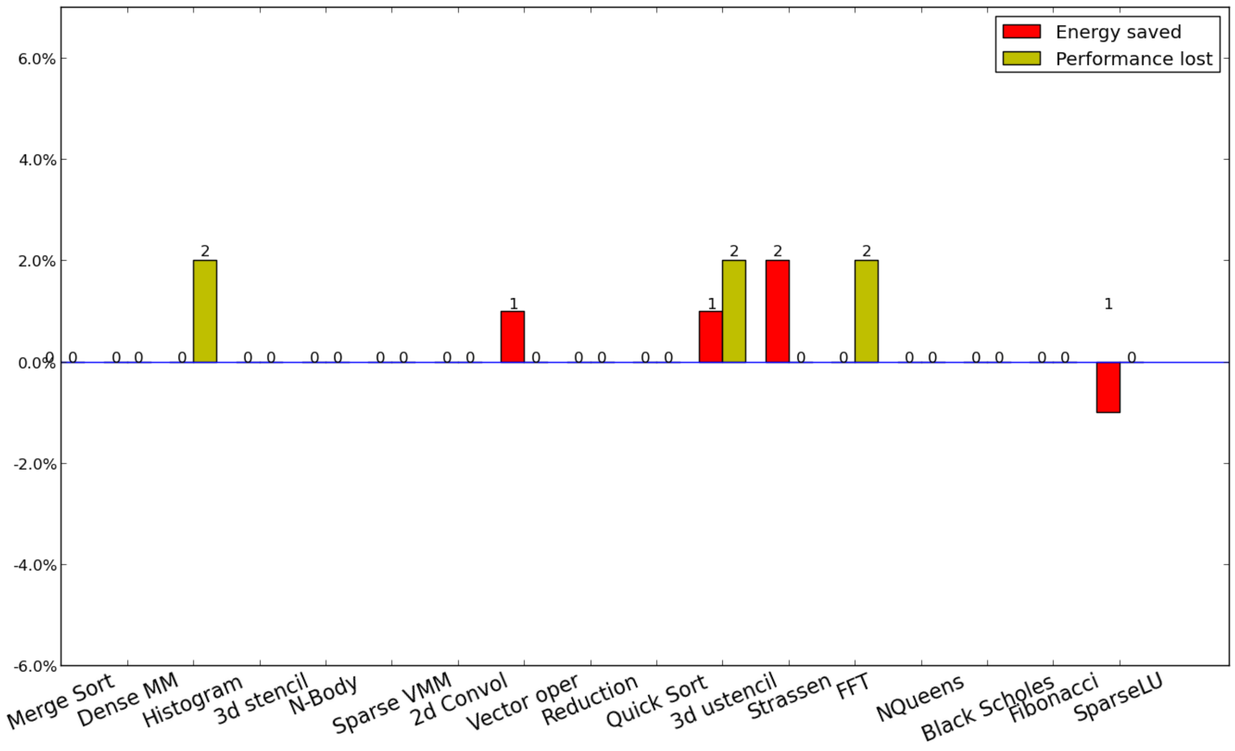
0.0.31 Results for 8 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2300 MHz	2428 MHz	5.57 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2500 MHz	2417 MHz	3.32 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2600 MHz	2400 MHz	7.69 %



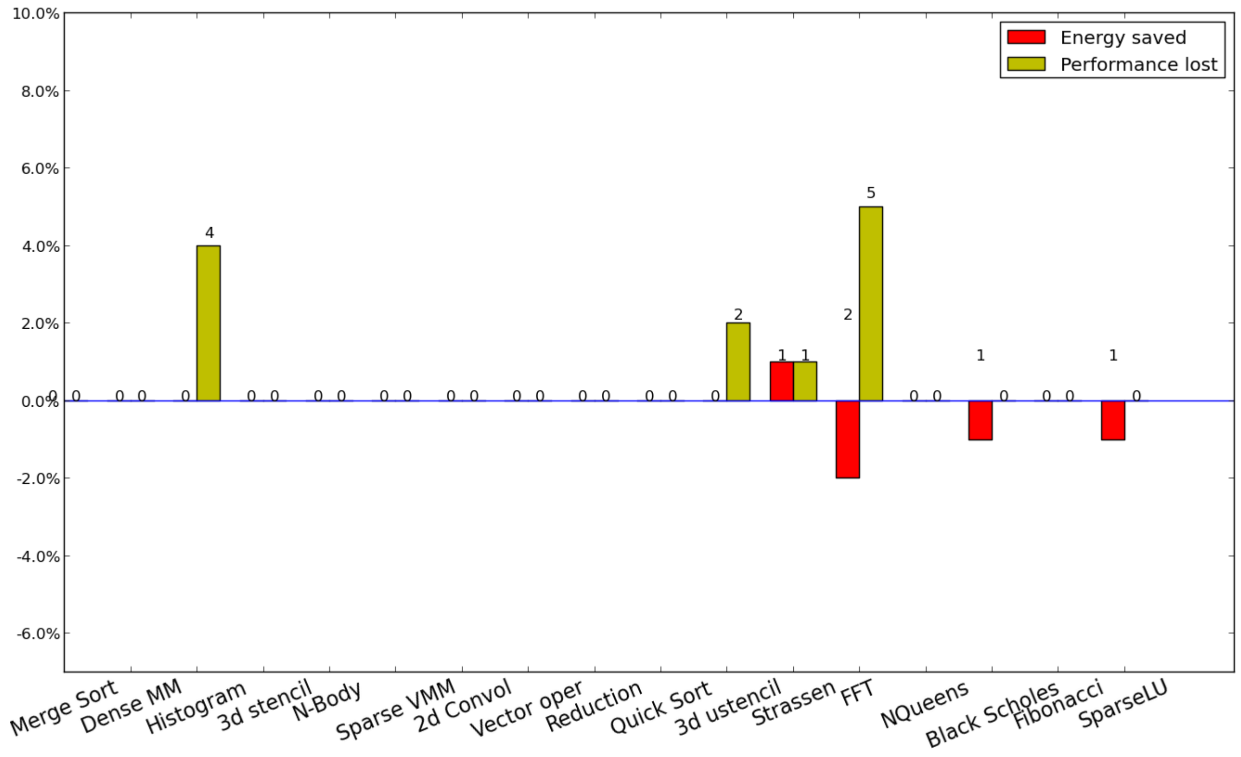
0.0.32 Results for 7 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2400 MHz	2433 MHz	1.38 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2410 MHz	0.42 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2400 MHz	2600 MHz	8.33 %
Unstructured 3d stencil	2600 MHz	2443 MHz	6.04 %



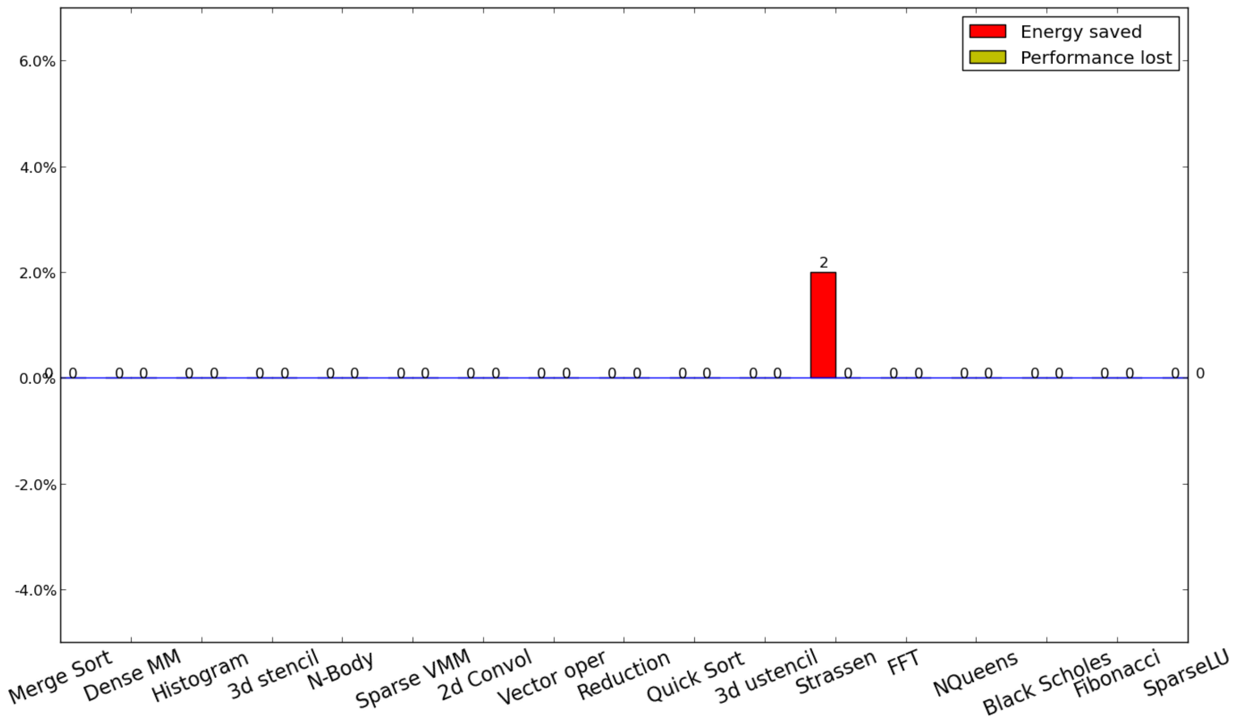
0.0.33 Results for 6 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2433 MHz	6.42 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2414 MHz	7.15 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2500 MHz	2431 MHz	2.76 %



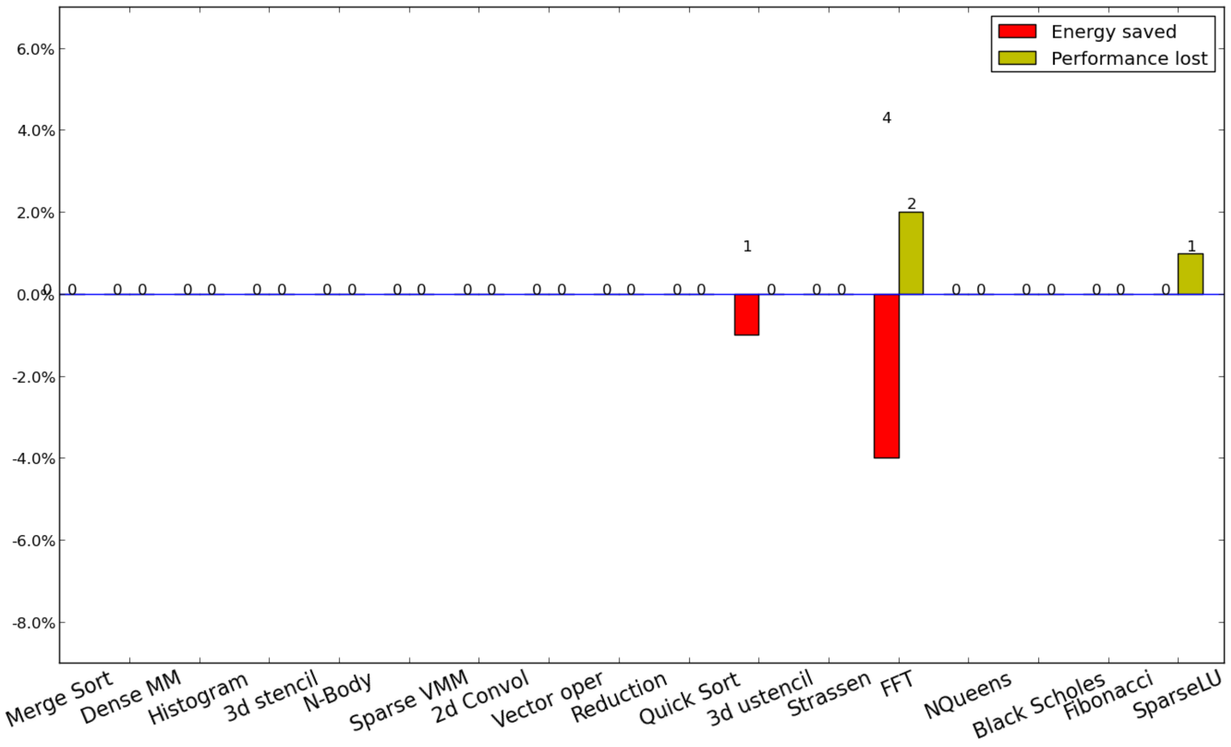
0.0.34 Results for 5 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2500 MHz	2600 MHz	4.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2600 MHz	8.33 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2200 MHz	2600 MHz	18.18 %
Unstructured 3d stencil	2600 MHz	2600 MHz	0.00 %



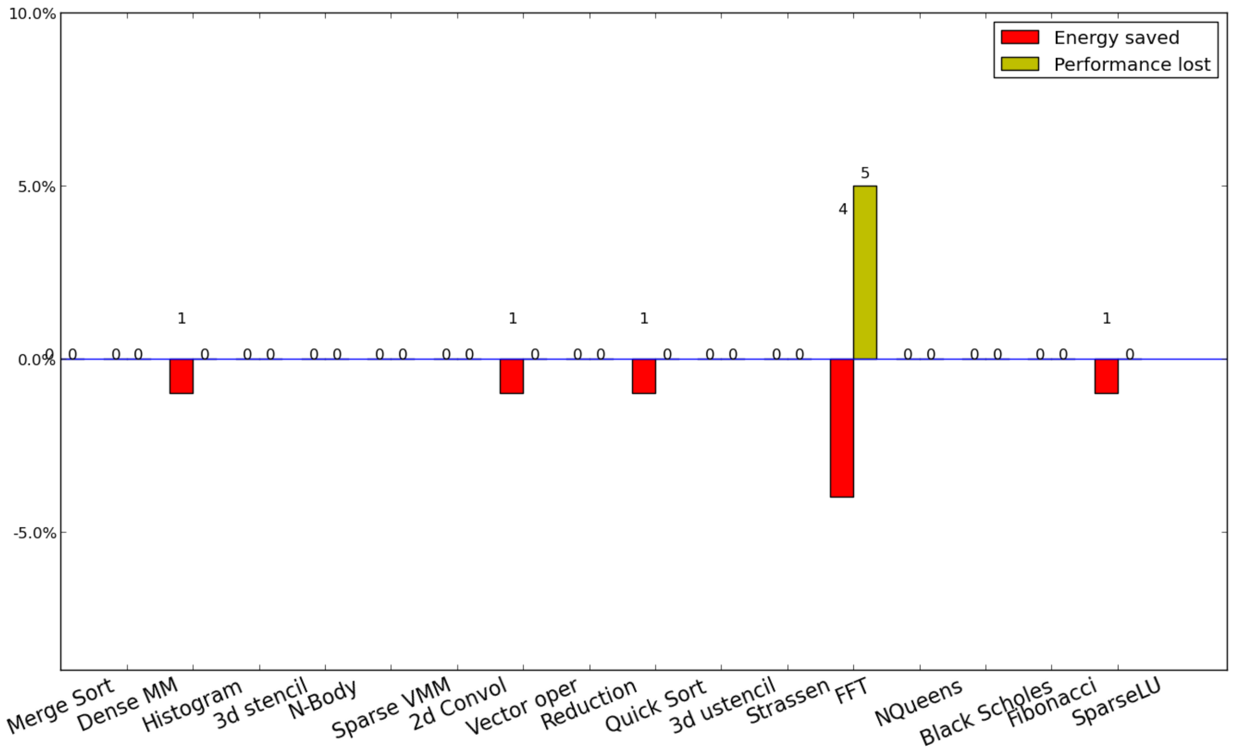
0.0.35 Results for 4 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2500 MHz	2600 MHz	4.00 %
Quick Sort	2400 MHz	2600 MHz	8.33 %
Unstructured 3d stencil	2600 MHz	2600 MHz	0.00 %



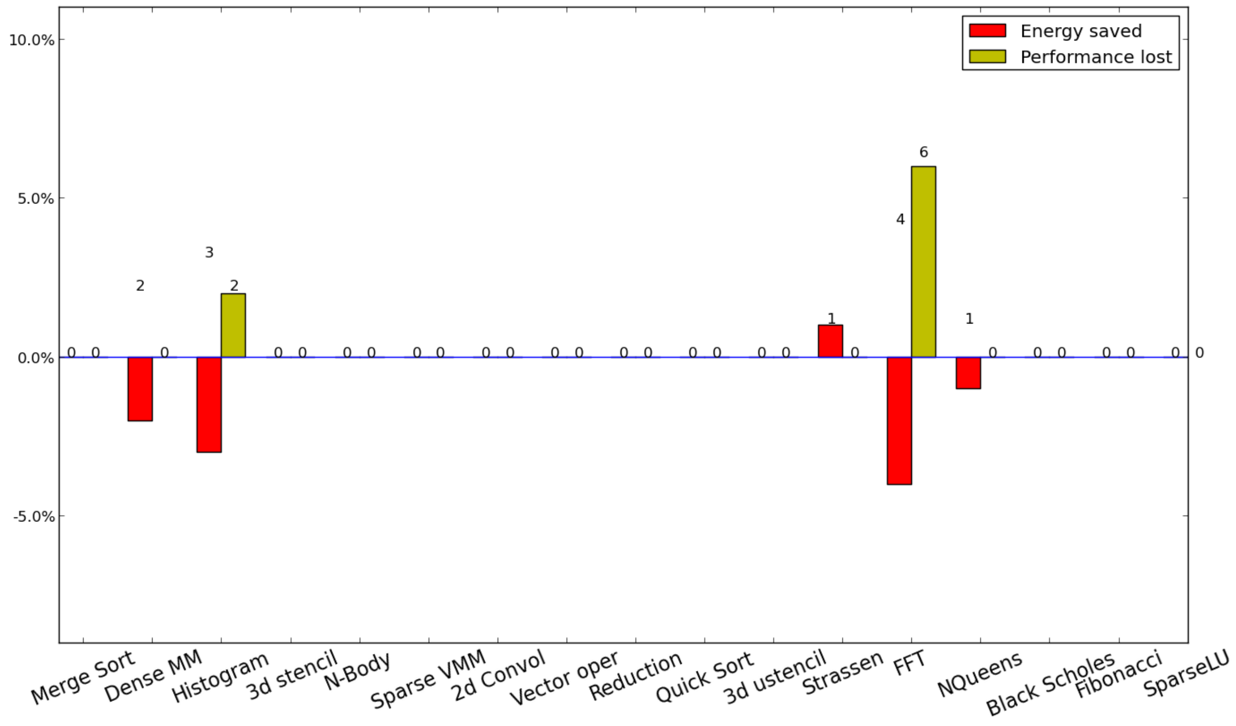
0.0.36 Results for 3 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2100 MHz	2600 MHz	23.81 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2600 MHz	2600 MHz	0.00 %



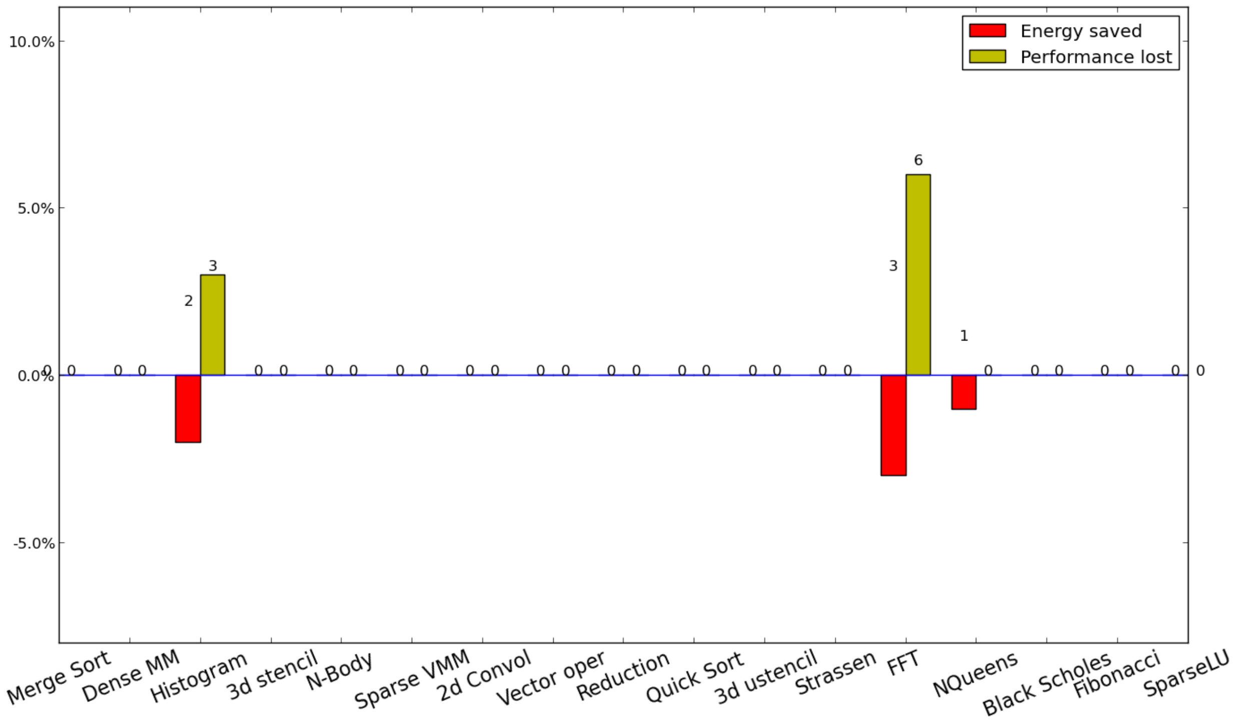
0.0.37 Results for 2 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2600 MHz	2600 MHz	0.00 %



0.0.38 Results for 1 thread

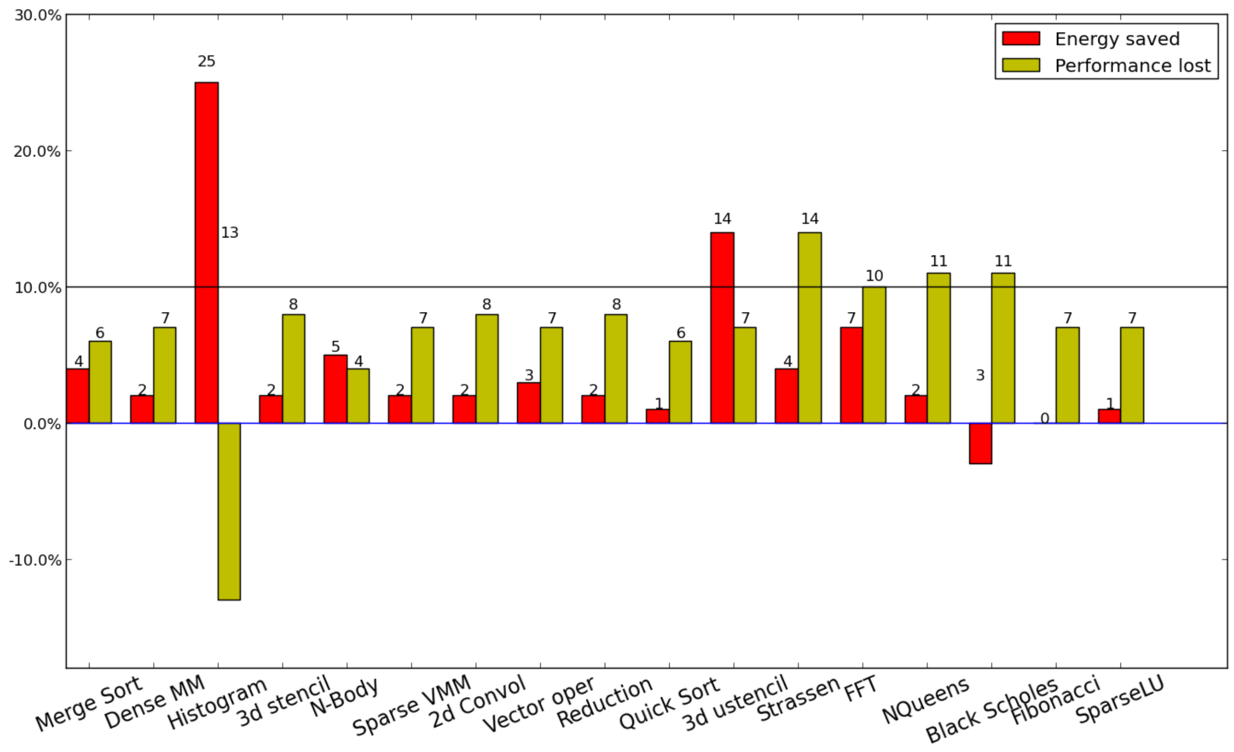
Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2600 MHz	2600 MHz	0.00 %



Results for the metric Operations per Joule under the constraint that performance cannot suffer by more than 10%

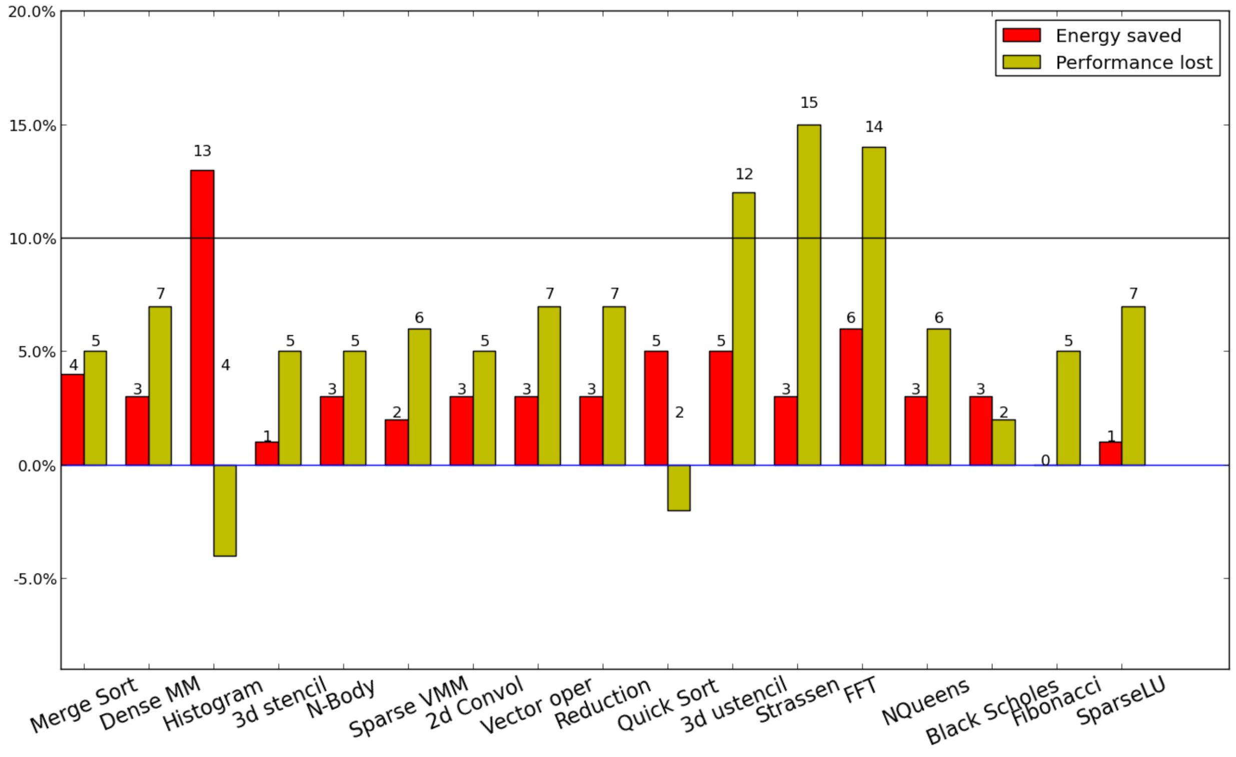
0.0.39 Results for 16 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2400 MHz	2393 MHz	0.29 %
Reduction	2400 MHz	2400 MHz	0.00 %
Histogram	2000 MHz	2197 MHz	9.85 %
2d convolution	2400 MHz	2400 MHz	0.00 %
3d stencil	2400 MHz	2400 MHz	0.00 %
Dense matrix multiplication	2400 MHz	2405 MHz	0.21 %
Sparse matrix vector multiplication	2500 MHz	2432 MHz	2.72 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2500 MHz	2403 MHz	3.88 %
FFT	2200 MHz	2095 MHz	4.77 %
SparseLU	2400 MHz	2425 MHz	1.04 %
NQueens	2400 MHz	2400 MHz	0.00 %
Strassen	2300 MHz	2242 MHz	2.52 %
Black Scholes	2400 MHz	2422 MHz	0.92 %
Fibonacci	2600 MHz	2418 MHz	7.00 %
Quick Sort	2600 MHz	2410 MHz	7.31 %
Unstructured 3d stencil	2100 MHz	2102 MHz	0.10 %



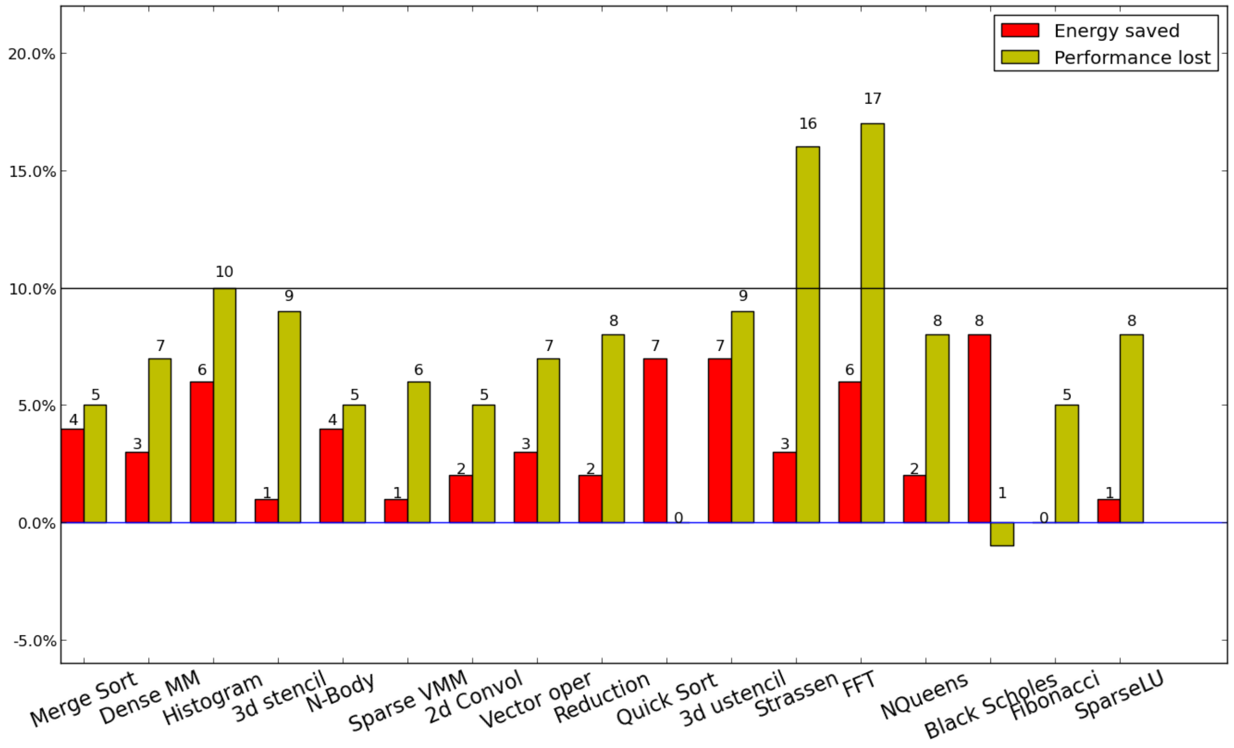
0.0.40 Results for 15 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2400 MHz	2400 MHz	0.00 %
Reduction	2400 MHz	2400 MHz	0.00 %
Histogram	1900 MHz	2218 MHz	16.74 %
2d convolution	2400 MHz	2466 MHz	2.75 %
3d stencil	2500 MHz	2468 MHz	1.28 %
Dense matrix multiplication	2400 MHz	2400 MHz	0.00 %
Sparse matrix vector multiplication	2400 MHz	2446 MHz	1.92 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2400 MHz	2463 MHz	2.62 %
FFT	2200 MHz	2103 MHz	4.41 %
SparseLU	2400 MHz	2426 MHz	1.08 %
NQueens	2400 MHz	2400 MHz	0.00 %
Strassen	2400 MHz	2209 MHz	7.96 %
Black Scholes	2400 MHz	2441 MHz	1.71 %
Fibonacci	2100 MHz	2454 MHz	16.86 %
Quick Sort	2300 MHz	2451 MHz	6.57 %
Unstructured 3d stencil	2200 MHz	2204 MHz	0.18 %



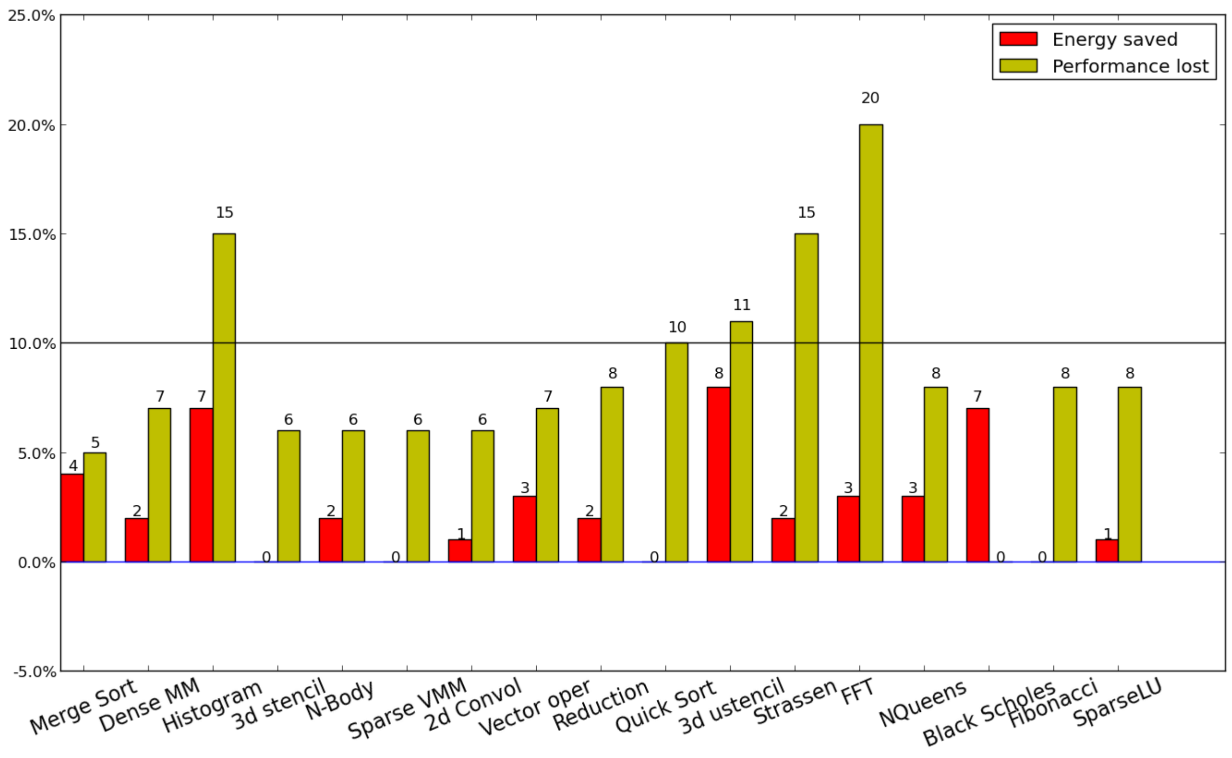
0.0.41 Results for 14 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2400 MHz	2400 MHz	0.00 %
Reduction	2400 MHz	2404 MHz	0.17 %
Histogram	2000 MHz	2190 MHz	9.50 %
2d convolution	2400 MHz	2466 MHz	2.75 %
3d stencil	2400 MHz	2400 MHz	0.00 %
Dense matrix multiplication	2400 MHz	2400 MHz	0.00 %
Sparse matrix vector multiplication	2400 MHz	2448 MHz	2.00 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2400 MHz	2463 MHz	2.62 %
FFT	2200 MHz	2132 MHz	3.09 %
SparseLU	2400 MHz	2427 MHz	1.12 %
NQueens	2400 MHz	2400 MHz	0.00 %
Strassen	2400 MHz	2195 MHz	8.54 %
Black Scholes	2300 MHz	2421 MHz	5.26 %
Fibonacci	2200 MHz	2454 MHz	11.55 %
Quick Sort	2500 MHz	2440 MHz	2.40 %
Unstructured 3d stencil	2200 MHz	2202 MHz	0.09 %



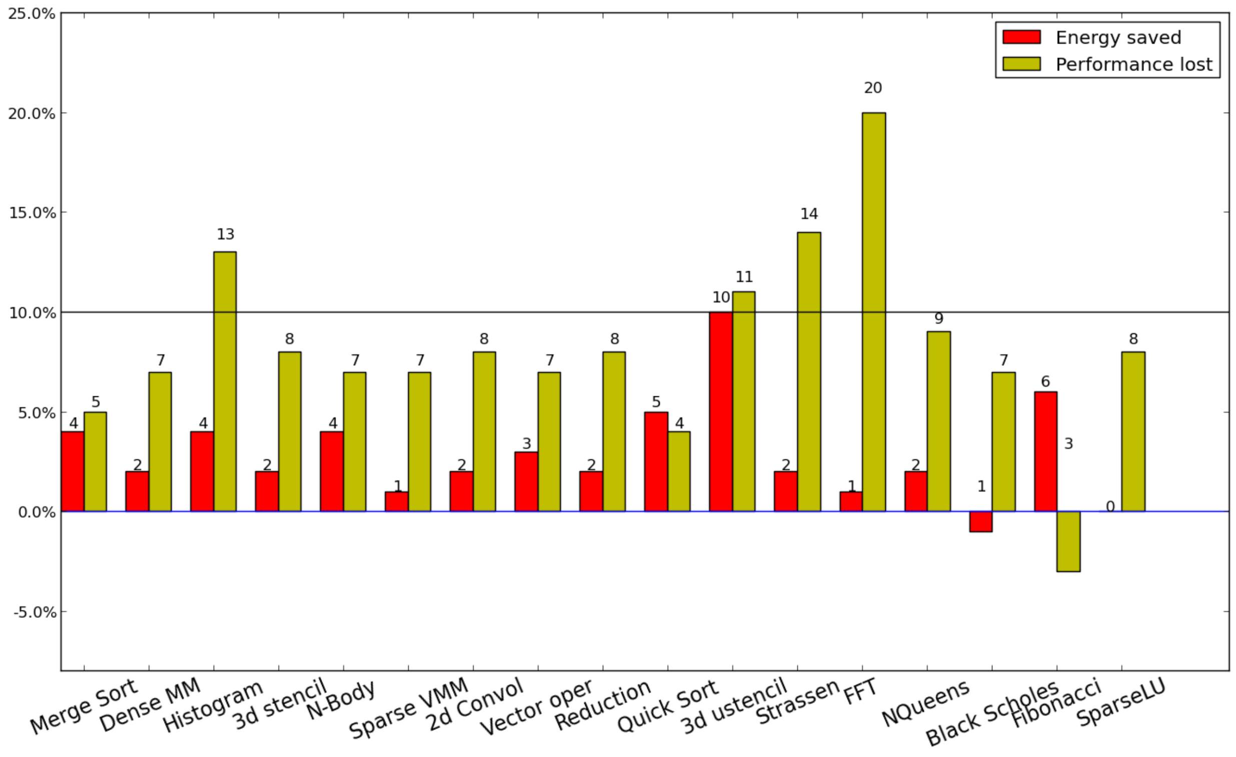
0.0.42 Results for 13 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2400 MHz	2400 MHz	0.00 %
Reduction	2400 MHz	2404 MHz	0.17 %
Histogram	2200 MHz	2174 MHz	1.18 %
2d convolution	2400 MHz	2433 MHz	1.38 %
3d stencil	2400 MHz	2445 MHz	1.88 %
Dense matrix multiplication	2400 MHz	2400 MHz	0.00 %
Sparse matrix vector multiplication	2400 MHz	2445 MHz	1.88 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2400 MHz	2431 MHz	1.29 %
FFT	2500 MHz	2127 MHz	14.92 %
SparseLU	2400 MHz	2419 MHz	0.79 %
NQueens	2400 MHz	2400 MHz	0.00 %
Strassen	2300 MHz	2220 MHz	3.48 %
Black Scholes	2400 MHz	2425 MHz	1.04 %
Fibonacci	2600 MHz	2400 MHz	7.69 %
Quick Sort	2600 MHz	2441 MHz	6.12 %
Unstructured 3d stencil	2200 MHz	2142 MHz	2.64 %



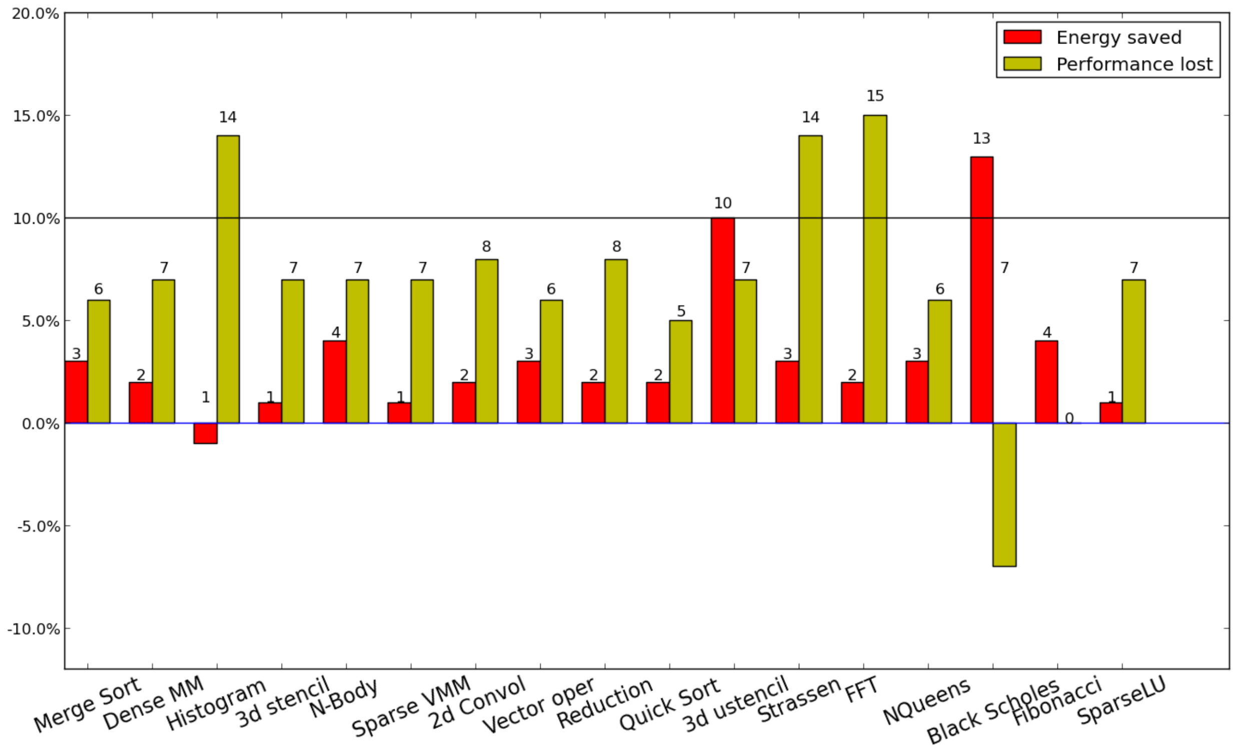
0.0.43 Results for 12 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2400 MHz	2400 MHz	0.00 %
Reduction	2400 MHz	2400 MHz	0.00 %
Histogram	2500 MHz	2220 MHz	11.20 %
2d convolution	2400 MHz	2400 MHz	0.00 %
3d stencil	2400 MHz	2401 MHz	0.04 %
Dense matrix multiplication	2400 MHz	2402 MHz	0.08 %
Sparse matrix vector multiplication	2400 MHz	2423 MHz	0.96 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2400 MHz	2400 MHz	0.00 %
FFT	2500 MHz	2135 MHz	14.60 %
SparseLU	2400 MHz	2424 MHz	1.00 %
NQueens	2400 MHz	2400 MHz	0.00 %
Strassen	2400 MHz	2238 MHz	6.75 %
Black Scholes	2600 MHz	2431 MHz	6.50 %
Fibonacci	2300 MHz	2400 MHz	4.35 %
Quick Sort	2500 MHz	2419 MHz	3.24 %
Unstructured 3d stencil	2100 MHz	2100 MHz	0.00 %



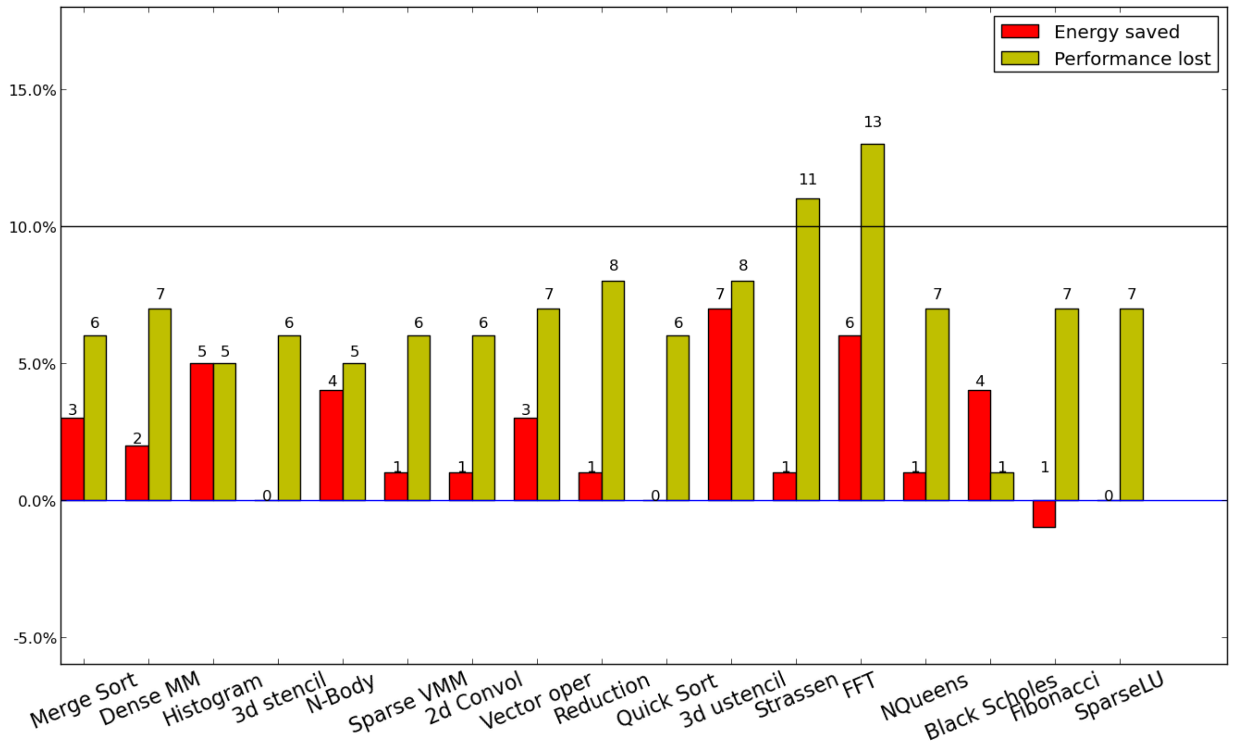
0.0.44 Results for 11 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2400 MHz	2395 MHz	0.21 %
Reduction	2400 MHz	2400 MHz	0.00 %
Histogram	2100 MHz	2235 MHz	6.43 %
2d convolution	2400 MHz	2405 MHz	0.21 %
3d stencil	2400 MHz	2400 MHz	0.00 %
Dense matrix multiplication	2400 MHz	2405 MHz	0.21 %
Sparse matrix vector multiplication	2400 MHz	2433 MHz	1.38 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2400 MHz	2400 MHz	0.00 %
FFT	2300 MHz	2124 MHz	7.65 %
SparseLU	2400 MHz	2418 MHz	0.75 %
NQueens	2400 MHz	2400 MHz	0.00 %
Strassen	2300 MHz	2228 MHz	3.13 %
Black Scholes	2600 MHz	2426 MHz	6.69 %
Fibonacci	2500 MHz	2436 MHz	2.56 %
Quick Sort	2500 MHz	2432 MHz	2.72 %
Unstructured 3d stencil	2100 MHz	2166 MHz	3.14 %



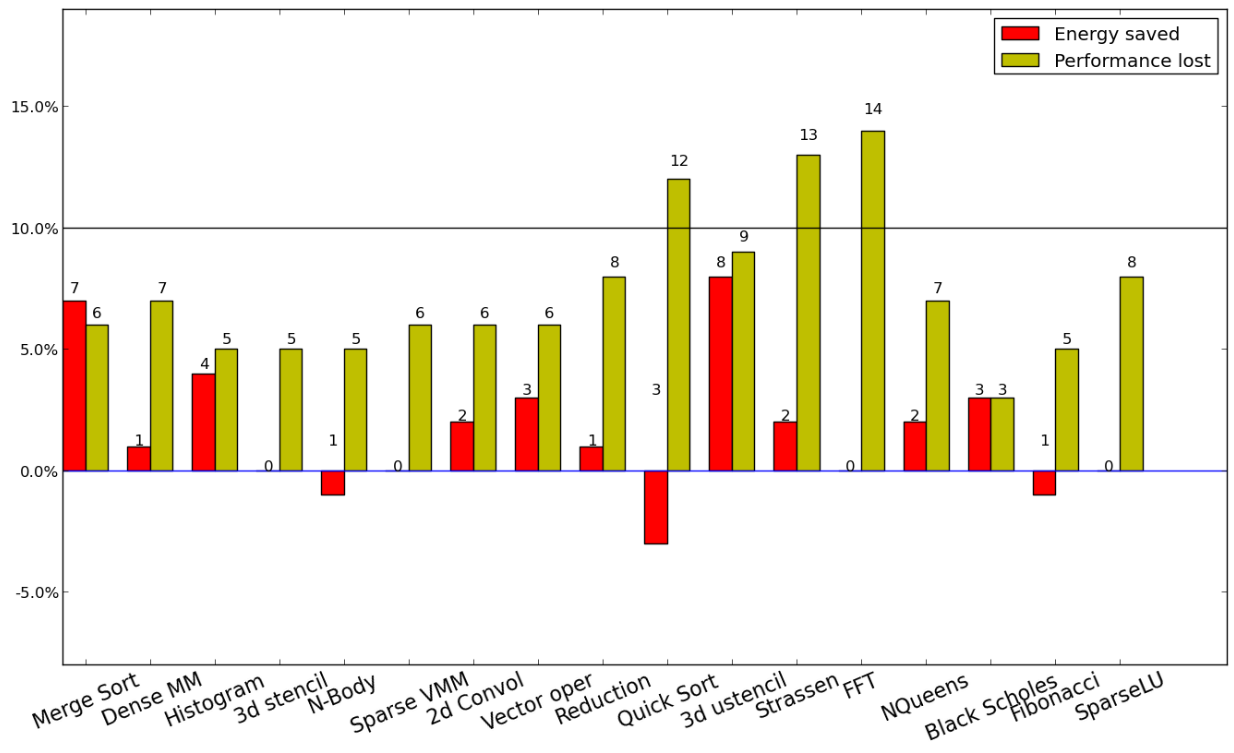
0.0.45 Results for 10 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2400 MHz	2397 MHz	0.12 %
Reduction	2400 MHz	2403 MHz	0.12 %
Histogram	2000 MHz	2200 MHz	10.00 %
2d convolution	2400 MHz	2445 MHz	1.88 %
3d stencil	2400 MHz	2442 MHz	1.75 %
Dense matrix multiplication	2400 MHz	2403 MHz	0.12 %
Sparse matrix vector multiplication	2400 MHz	2448 MHz	2.00 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2400 MHz	2447 MHz	1.96 %
FFT	2200 MHz	2121 MHz	3.59 %
SparseLU	2400 MHz	2426 MHz	1.08 %
NQueens	2400 MHz	2400 MHz	0.00 %
Strassen	2300 MHz	2250 MHz	2.17 %
Black Scholes	2400 MHz	2437 MHz	1.54 %
Fibonacci	2600 MHz	2427 MHz	6.65 %
Quick Sort	2600 MHz	2431 MHz	6.50 %
Unstructured 3d stencil	2100 MHz	2212 MHz	5.33 %



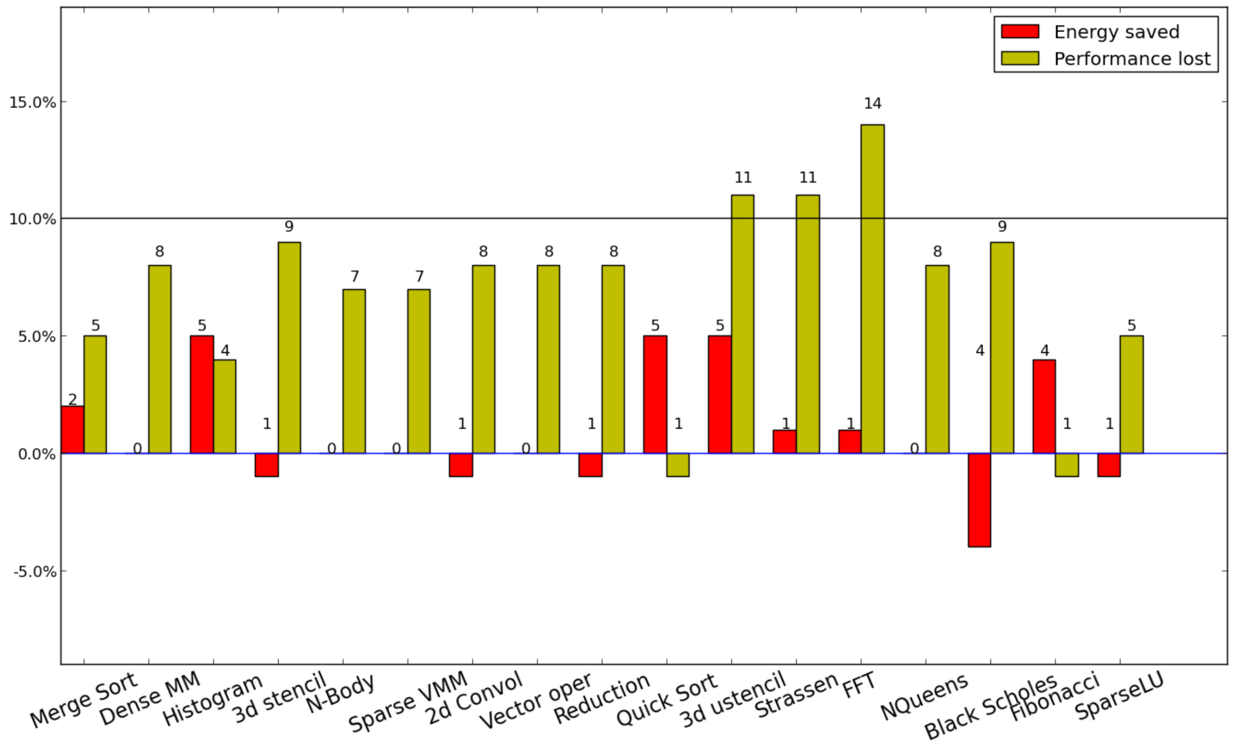
0.0.46 Results for 9 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2400 MHz	2396 MHz	0.17 %
Reduction	2400 MHz	2402 MHz	0.08 %
Histogram	2000 MHz	2193 MHz	9.65 %
2d convolution	2400 MHz	2445 MHz	1.88 %
3d stencil	2400 MHz	2447 MHz	1.96 %
Dense matrix multiplication	2400 MHz	2403 MHz	0.12 %
Sparse matrix vector multiplication	2500 MHz	2443 MHz	2.28 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2400 MHz	2447 MHz	1.96 %
FFT	2400 MHz	2128 MHz	11.33 %
SparseLU	2400 MHz	2427 MHz	1.12 %
NQueens	2400 MHz	2400 MHz	0.00 %
Strassen	2400 MHz	2300 MHz	4.17 %
Black Scholes	2400 MHz	2427 MHz	1.12 %
Fibonacci	2400 MHz	2459 MHz	2.46 %
Quick Sort	2600 MHz	2430 MHz	6.54 %
Unstructured 3d stencil	2200 MHz	2140 MHz	2.73 %



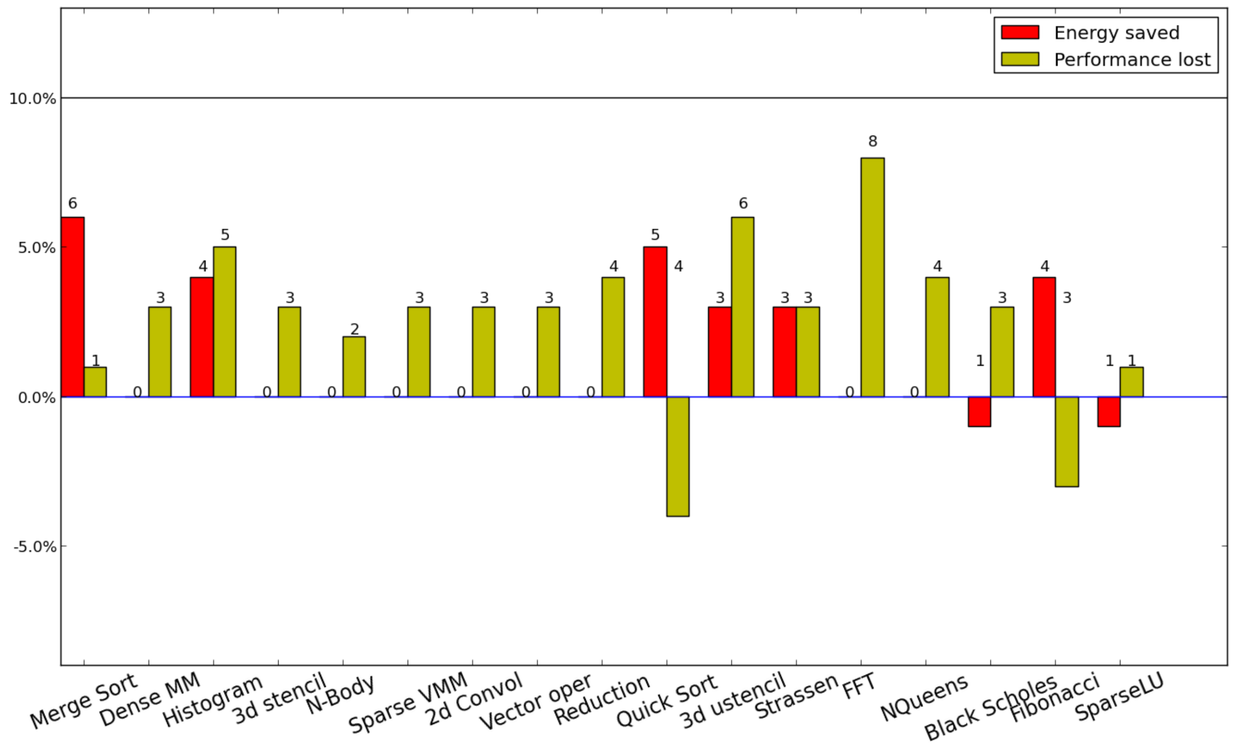
0.0.47 Results for 8 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2393 MHz	7.96 %
Reduction	2500 MHz	2400 MHz	4.00 %
Histogram	1900 MHz	2171 MHz	14.26 %
2d convolution	2500 MHz	2400 MHz	4.00 %
3d stencil	2400 MHz	2401 MHz	0.04 %
Dense matrix multiplication	2500 MHz	2402 MHz	3.92 %
Sparse matrix vector multiplication	2500 MHz	2424 MHz	3.04 %
Vector operation	2400 MHz	2400 MHz	0.00 %
N-body	2400 MHz	2400 MHz	0.00 %
FFT	2300 MHz	2124 MHz	7.65 %
SparseLU	2600 MHz	2486 MHz	4.38 %
NQueens	2600 MHz	2400 MHz	7.69 %
Strassen	2500 MHz	2400 MHz	4.00 %
Black Scholes	2600 MHz	2427 MHz	6.65 %
Fibonacci	2600 MHz	2448 MHz	5.85 %
Quick Sort	2500 MHz	2426 MHz	2.96 %
Unstructured 3d stencil	2200 MHz	2118 MHz	3.73 %



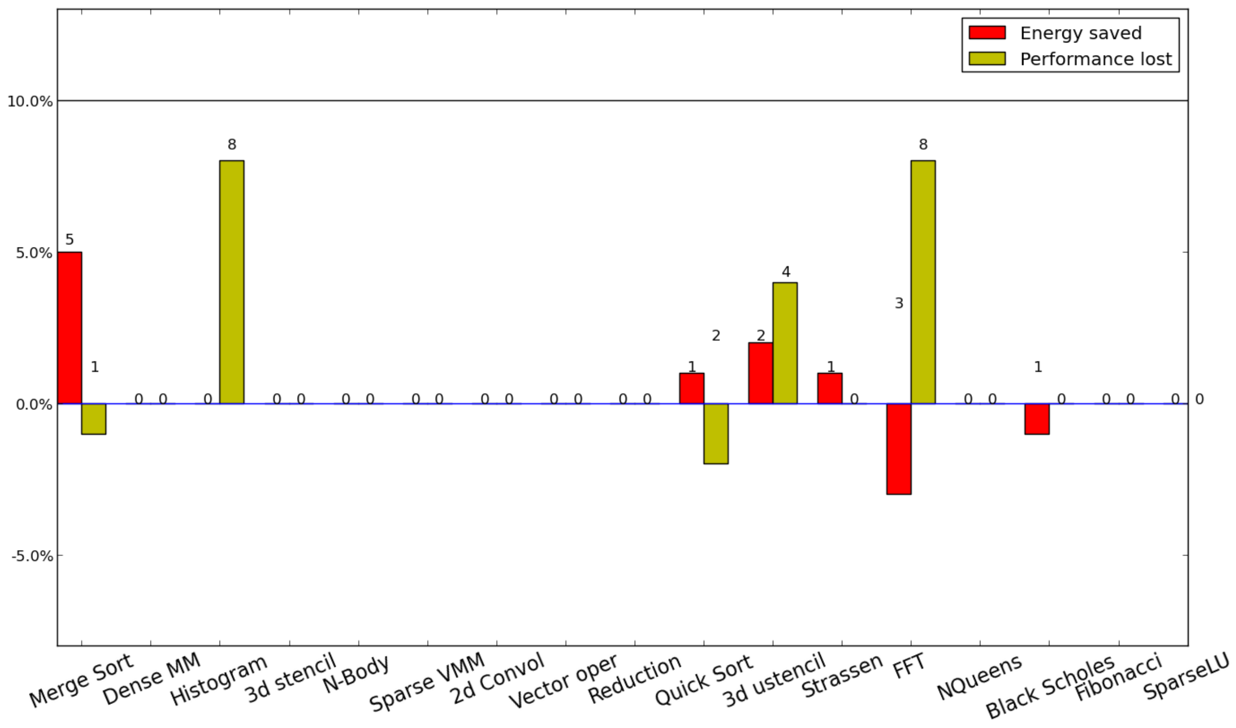
0.0.48 Results for 7 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2500 MHz	2500 MHz	0.00 %
Reduction	2400 MHz	2511 MHz	4.62 %
Histogram	2100 MHz	2245 MHz	6.90 %
2d convolution	2500 MHz	2518 MHz	0.72 %
3d stencil	2600 MHz	2519 MHz	3.12 %
Dense matrix multiplication	2400 MHz	2502 MHz	4.25 %
Sparse matrix vector multiplication	2500 MHz	2517 MHz	0.68 %
Vector operation	2400 MHz	2505 MHz	4.38 %
N-body	2600 MHz	2519 MHz	3.12 %
FFT	2100 MHz	2235 MHz	6.43 %
SparseLU	2600 MHz	2574 MHz	1.00 %
NQueens	2500 MHz	2500 MHz	0.00 %
Strassen	2500 MHz	2500 MHz	0.00 %
Black Scholes	2600 MHz	2520 MHz	3.08 %
Fibonacci	2600 MHz	2509 MHz	3.50 %
Quick Sort	2400 MHz	2518 MHz	4.92 %
Unstructured 3d stencil	2200 MHz	2263 MHz	2.86 %



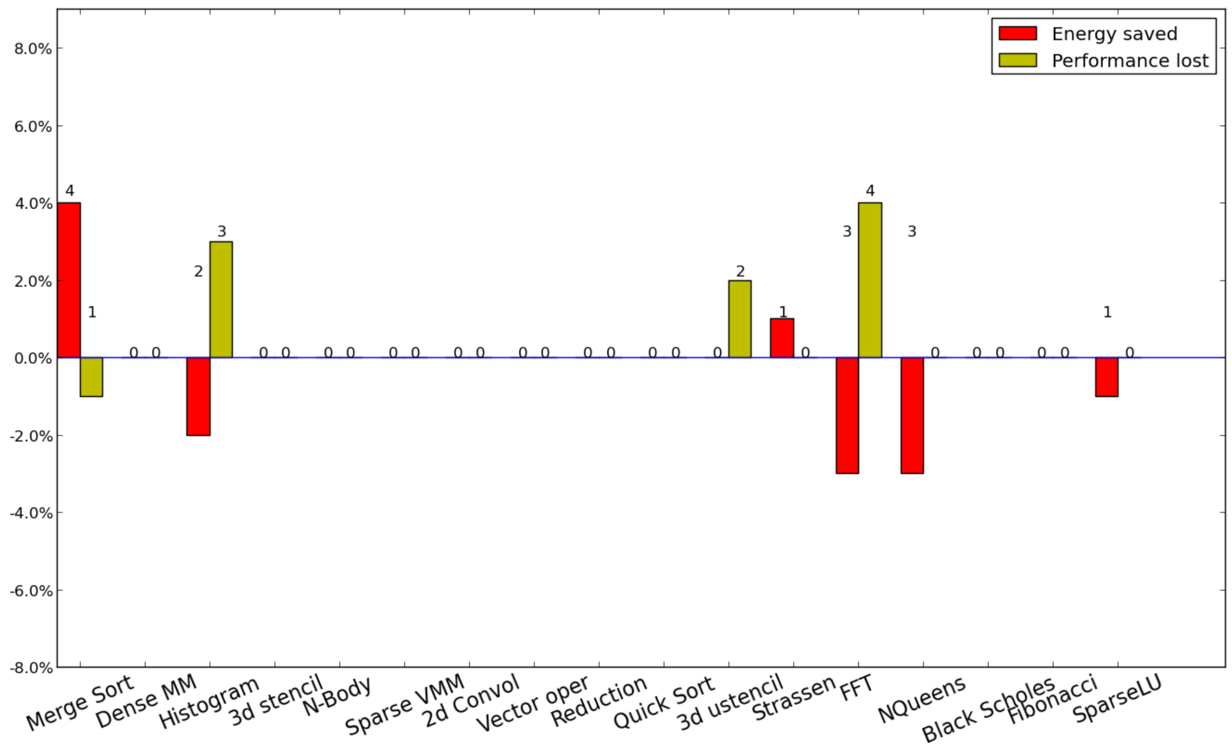
0.0.49 Results for 6 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2500 MHz	2592 MHz	3.68 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2200 MHz	2350 MHz	6.82 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2500 MHz	2600 MHz	4.00 %
Dense matrix multiplication	2500 MHz	2600 MHz	4.00 %
Sparse matrix vector multiplication	2500 MHz	2600 MHz	4.00 %
Vector operation	2400 MHz	2600 MHz	8.33 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2300 MHz	2331 MHz	1.35 %
SparseLU	2500 MHz	2600 MHz	4.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2597 MHz	0.12 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2598 MHz	0.08 %
Unstructured 3d stencil	2200 MHz	2353 MHz	6.95 %



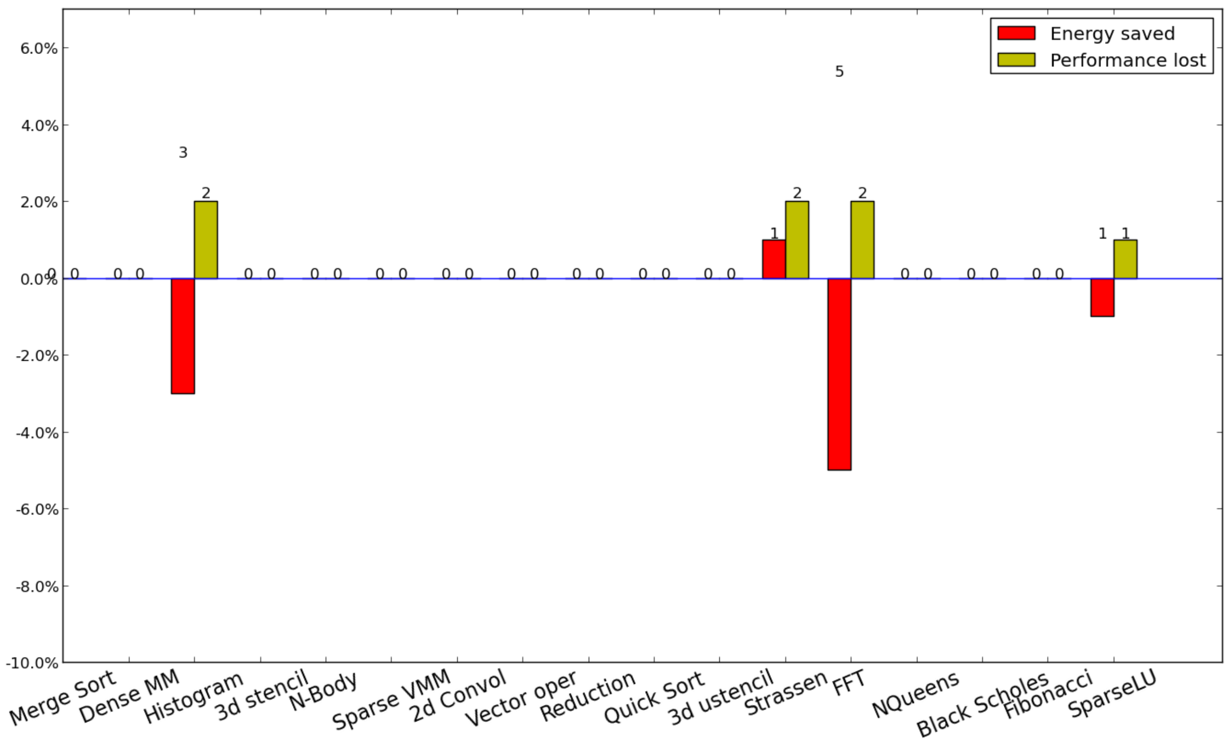
0.0.50 Results for 5 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2592 MHz	0.31 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2200 MHz	2518 MHz	14.45 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2515 MHz	4.79 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2500 MHz	2597 MHz	3.88 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2200 MHz	2600 MHz	18.18 %
Unstructured 3d stencil	2200 MHz	2513 MHz	14.23 %



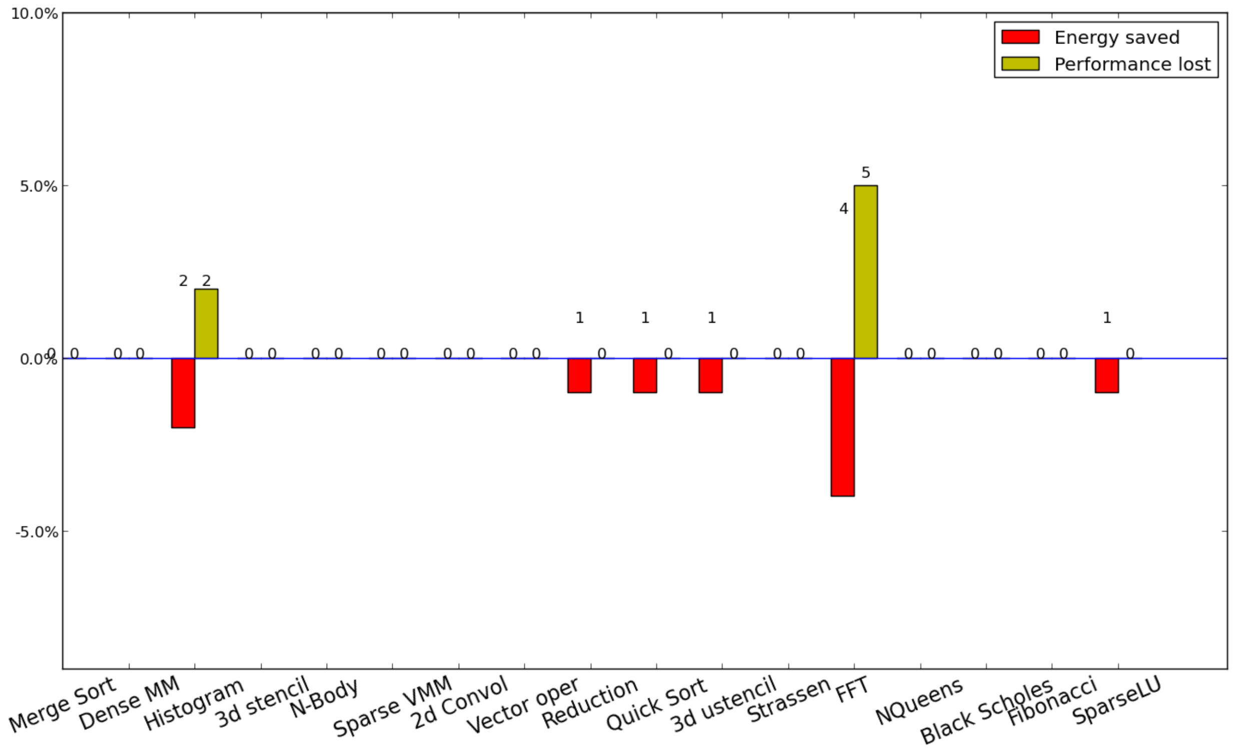
0.0.51 Results for 4 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2500 MHz	2600 MHz	4.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2500 MHz	2600 MHz	4.00 %
Quick Sort	2400 MHz	2600 MHz	8.33 %
Unstructured 3d stencil	2200 MHz	2600 MHz	18.18 %



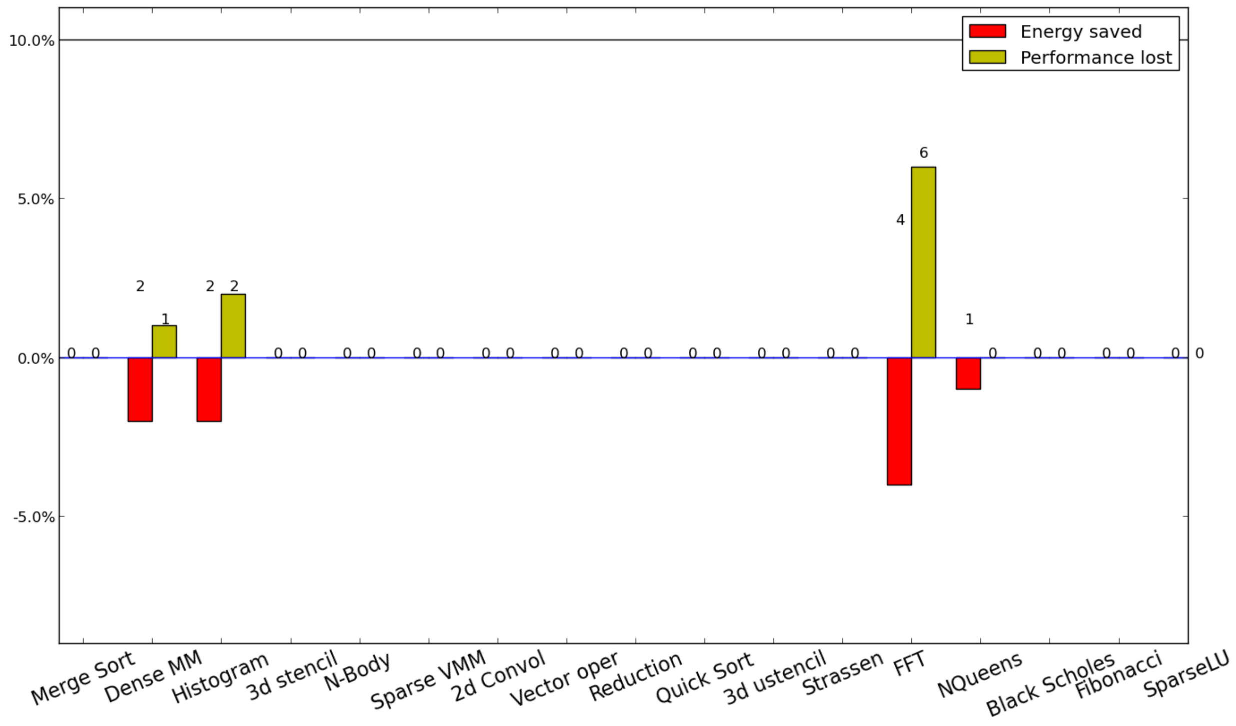
0.0.52 Results for 3 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2100 MHz	2600 MHz	23.81 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2200 MHz	2600 MHz	18.18 %



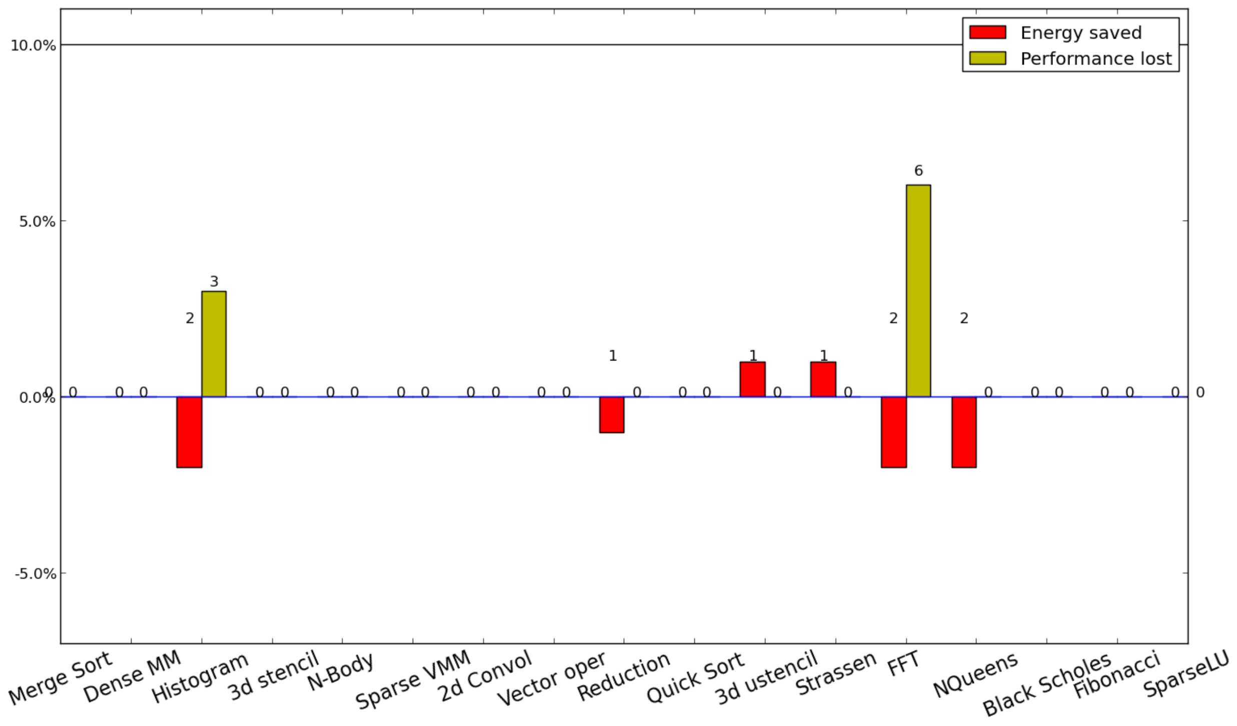
0.0.53 Results for 2 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2200 MHz	2600 MHz	18.18 %



0.0.54 Results for 1 thread

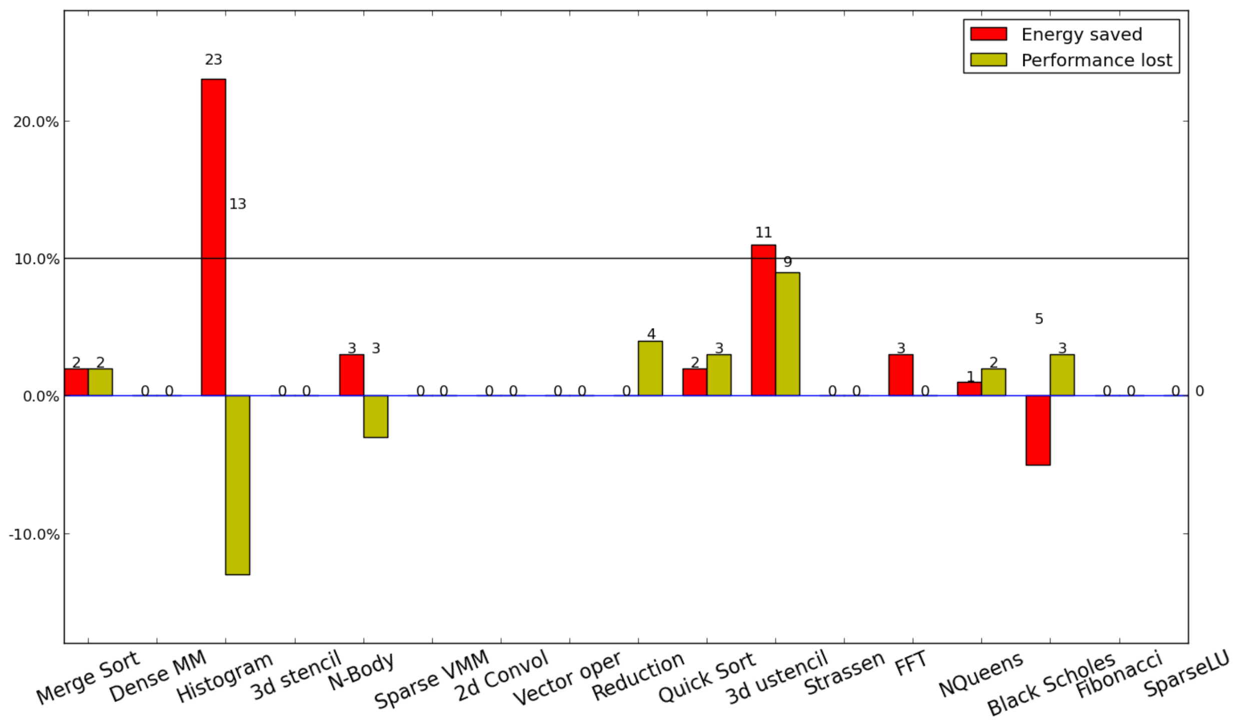
Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2500 MHz	2600 MHz	4.00 %



Results for the metric Energy Delay Product under the constraint that performance cannot suffer by more than 10%

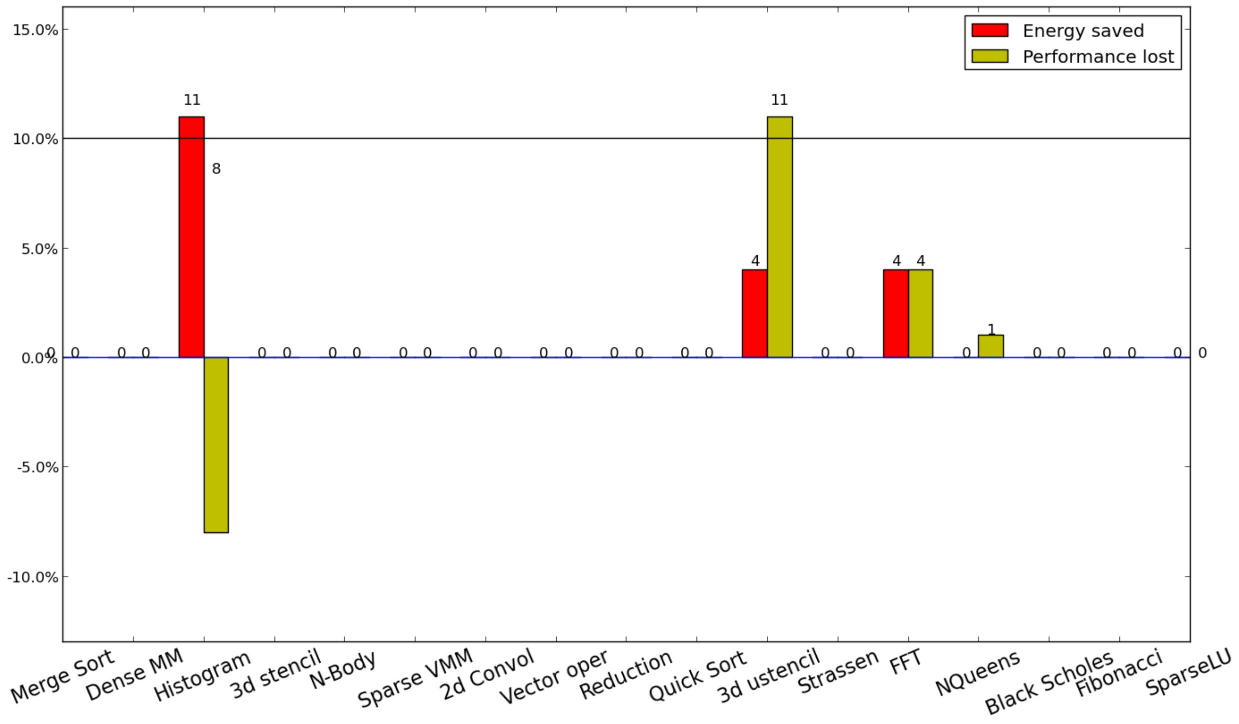
0.0.55 Results for 16 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2493 MHz	4.12 %
Reduction	2600 MHz	2505 MHz	3.65 %
Histogram	2000 MHz	2250 MHz	12.50 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2500 MHz	2598 MHz	3.92 %
FFT	2500 MHz	2363 MHz	5.48 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2549 MHz	1.96 %
Strassen	2400 MHz	2600 MHz	8.33 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2462 MHz	5.31 %
Unstructured 3d stencil	2100 MHz	2138 MHz	1.81 %



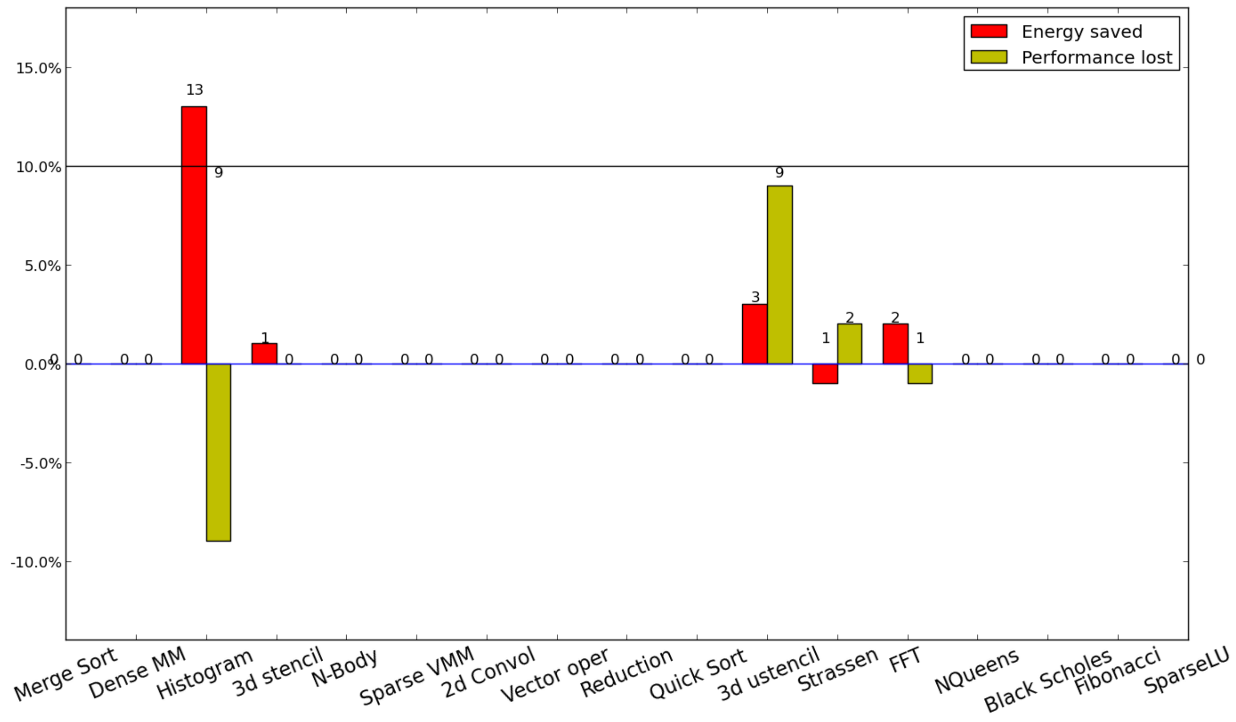
0.0.56 Results for 15 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	1900 MHz	2314 MHz	21.79 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2330 MHz	2.92 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2555 MHz	1.73 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2100 MHz	2600 MHz	23.81 %
Quick Sort	2500 MHz	2600 MHz	4.00 %
Unstructured 3d stencil	2400 MHz	2275 MHz	5.21 %



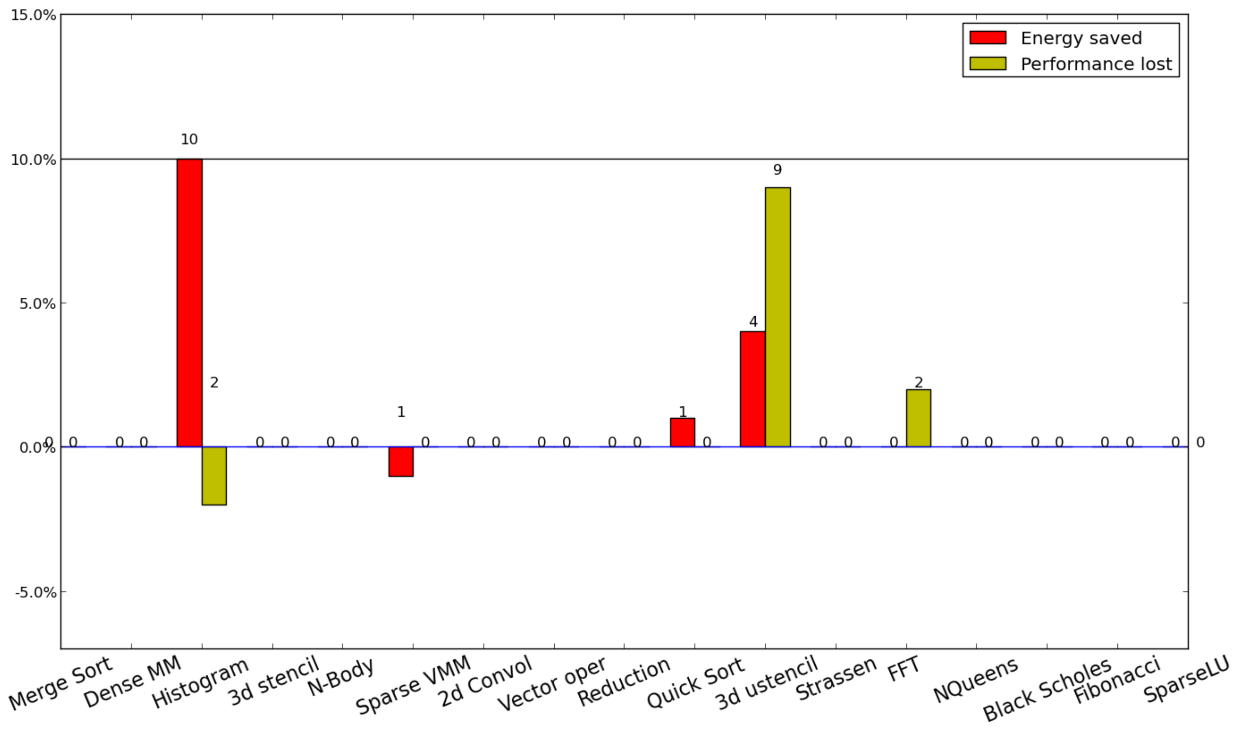
0.0.57 Results for 14 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2100 MHz	2412 MHz	14.86 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2395 MHz	0.21 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2594 MHz	0.23 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2200 MHz	2600 MHz	18.18 %
Quick Sort	2500 MHz	2600 MHz	4.00 %
Unstructured 3d stencil	2200 MHz	2314 MHz	5.18 %



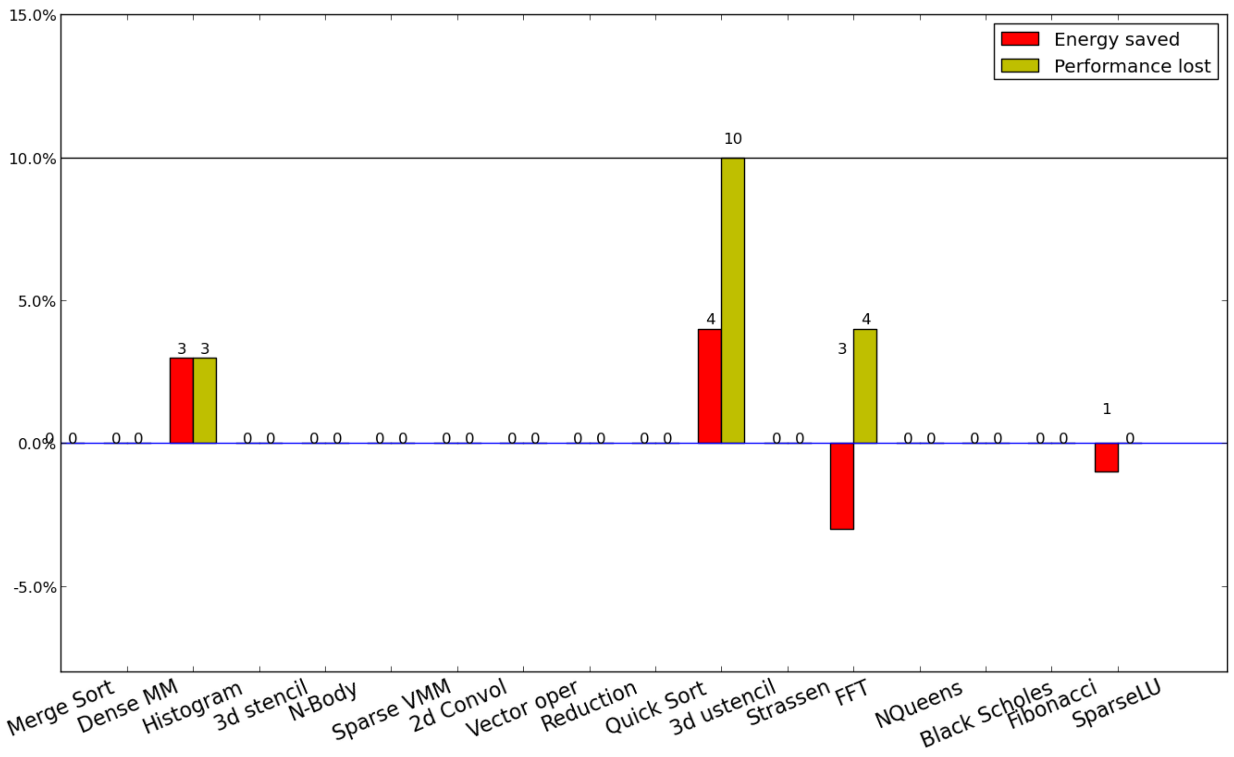
0.0.58 Results for 13 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2200 MHz	2418 MHz	9.91 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2400 MHz	7.69 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2572 MHz	1.08 %
Unstructured 3d stencil	2300 MHz	2266 MHz	1.48 %



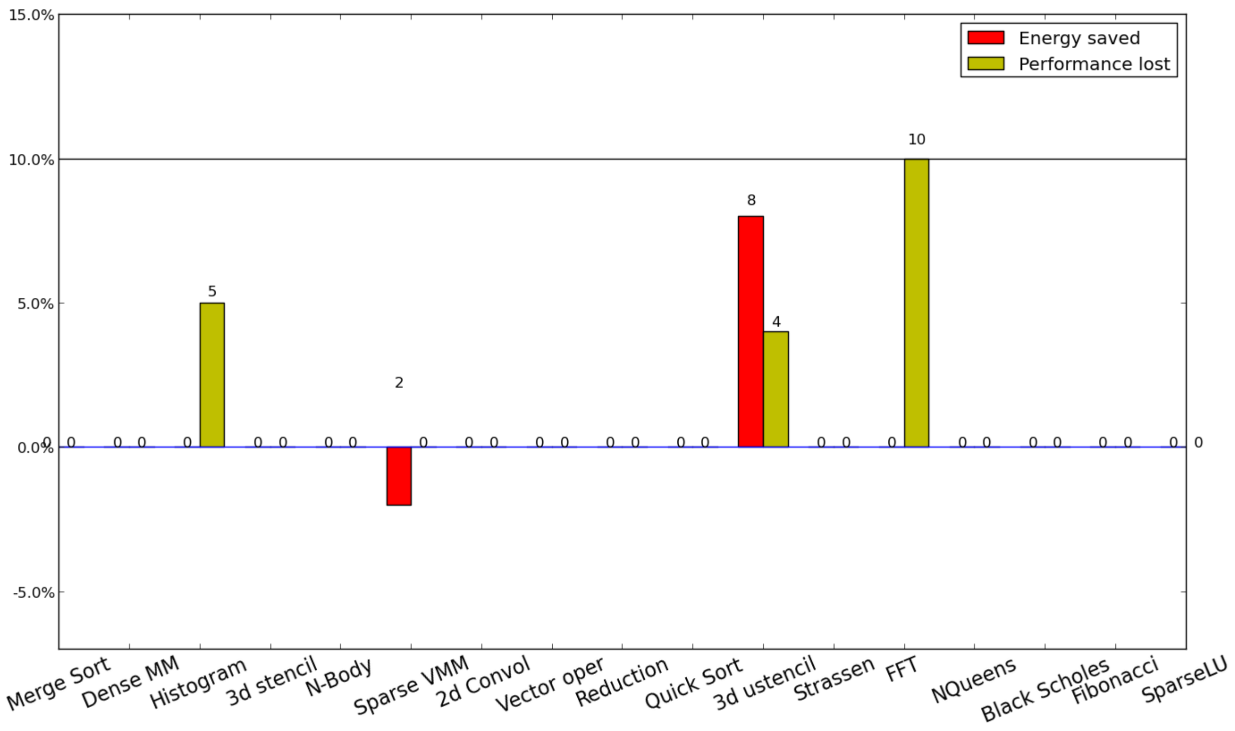
0.0.59 Results for 12 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2500 MHz	2435 MHz	2.60 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2400 MHz	7.69 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2300 MHz	2600 MHz	13.04 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2400 MHz	2253 MHz	6.12 %



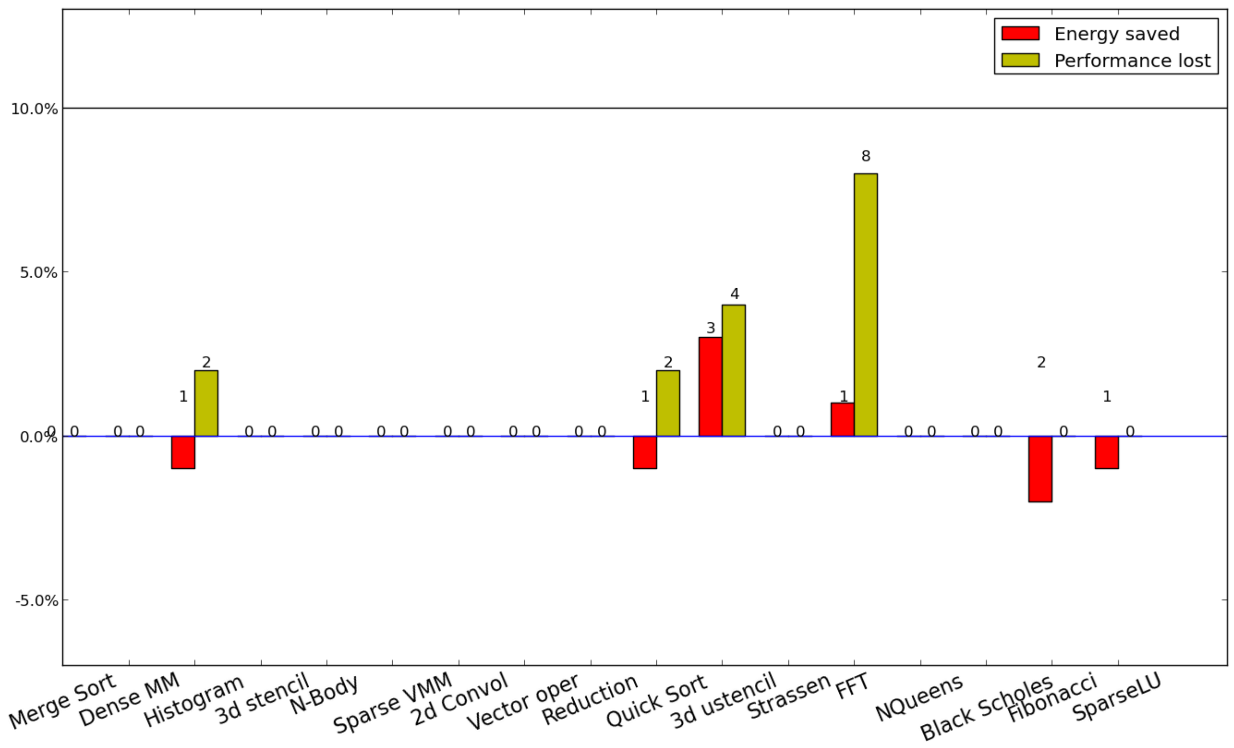
0.0.60 Results for 11 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2400 MHz	2457 MHz	2.38 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2500 MHz	2298 MHz	8.08 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2500 MHz	2600 MHz	4.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2100 MHz	2301 MHz	9.57 %



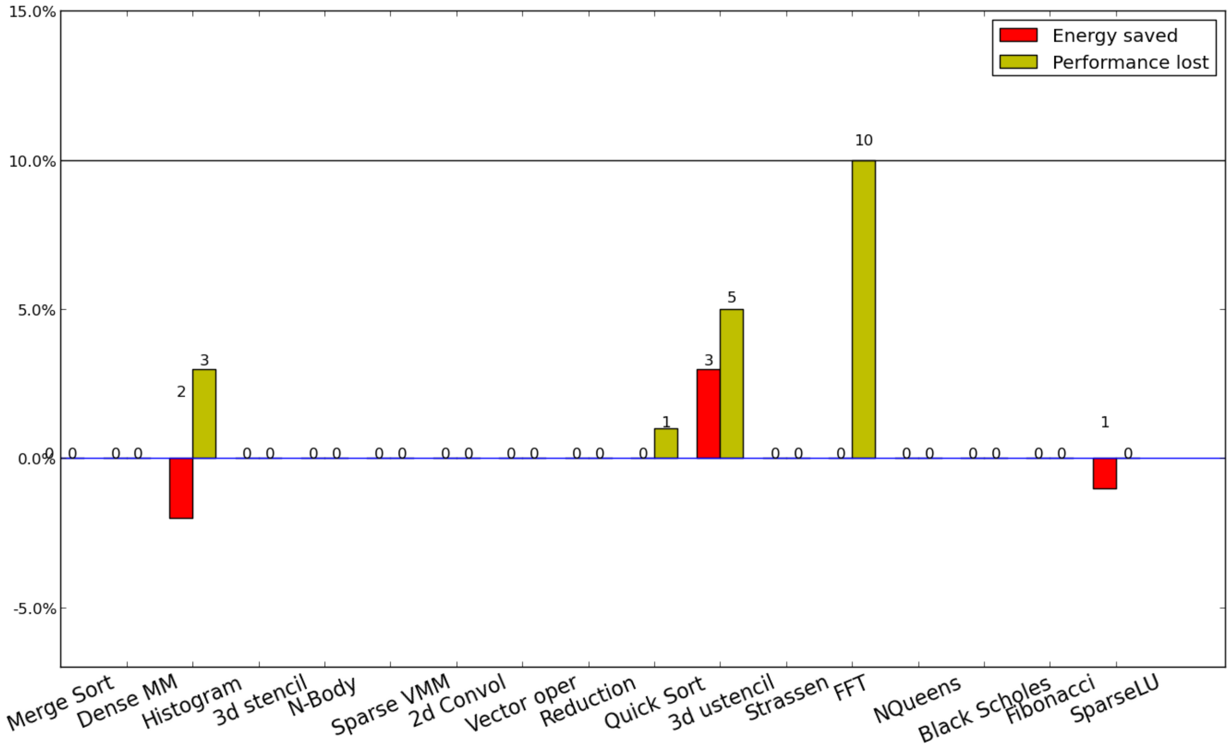
0.0.61 Results for 10 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2200 MHz	2438 MHz	10.82 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2341 MHz	2.46 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2200 MHz	2383 MHz	8.32 %



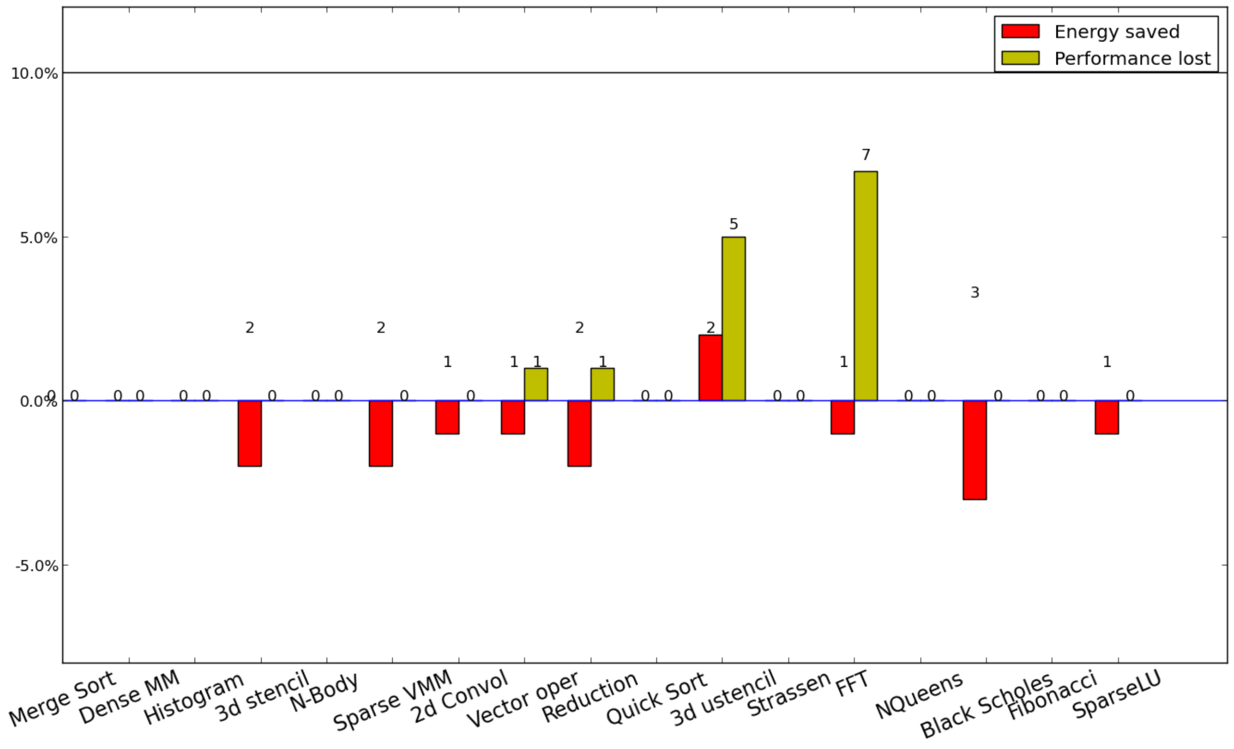
0.0.62 Results for 9 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2500 MHz	2395 MHz	4.20 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2500 MHz	2340 MHz	6.40 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2400 MHz	2600 MHz	8.33 %
Fibonacci	2400 MHz	2600 MHz	8.33 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2400 MHz	2336 MHz	2.67 %



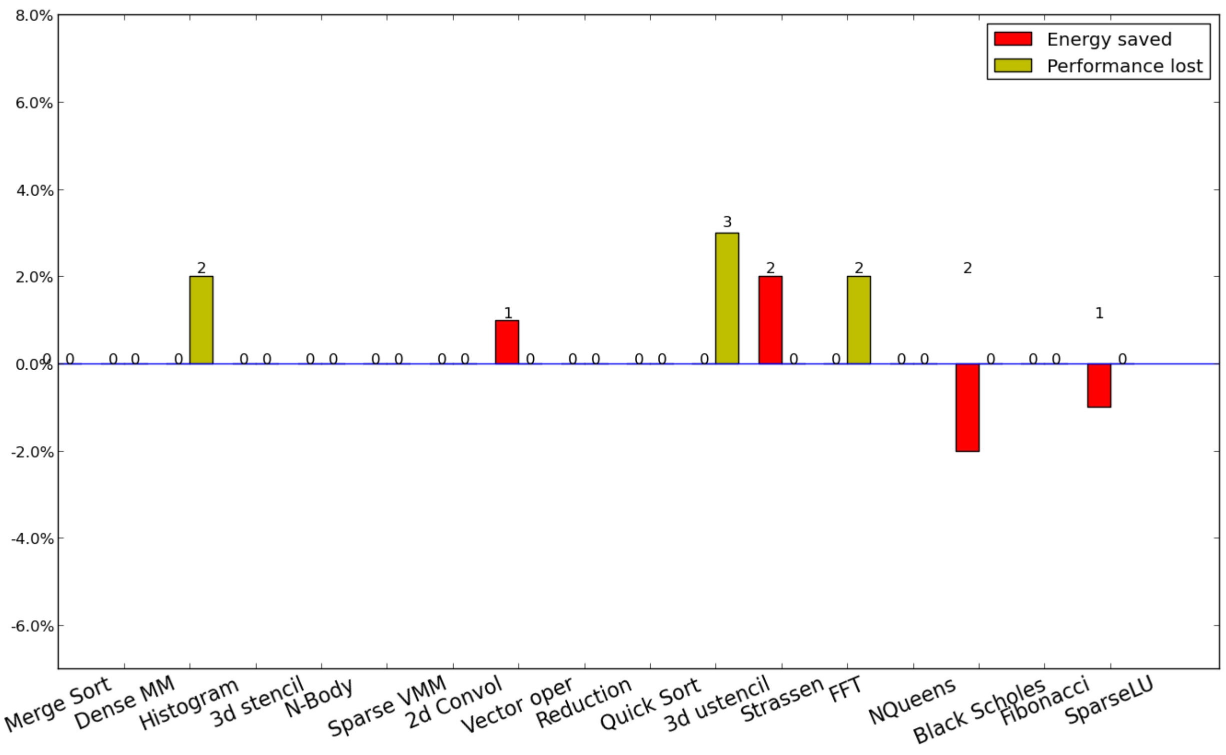
0.0.63 Results for 8 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2300 MHz	2428 MHz	5.57 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2500 MHz	2422 MHz	3.12 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2600 MHz	2400 MHz	7.69 %



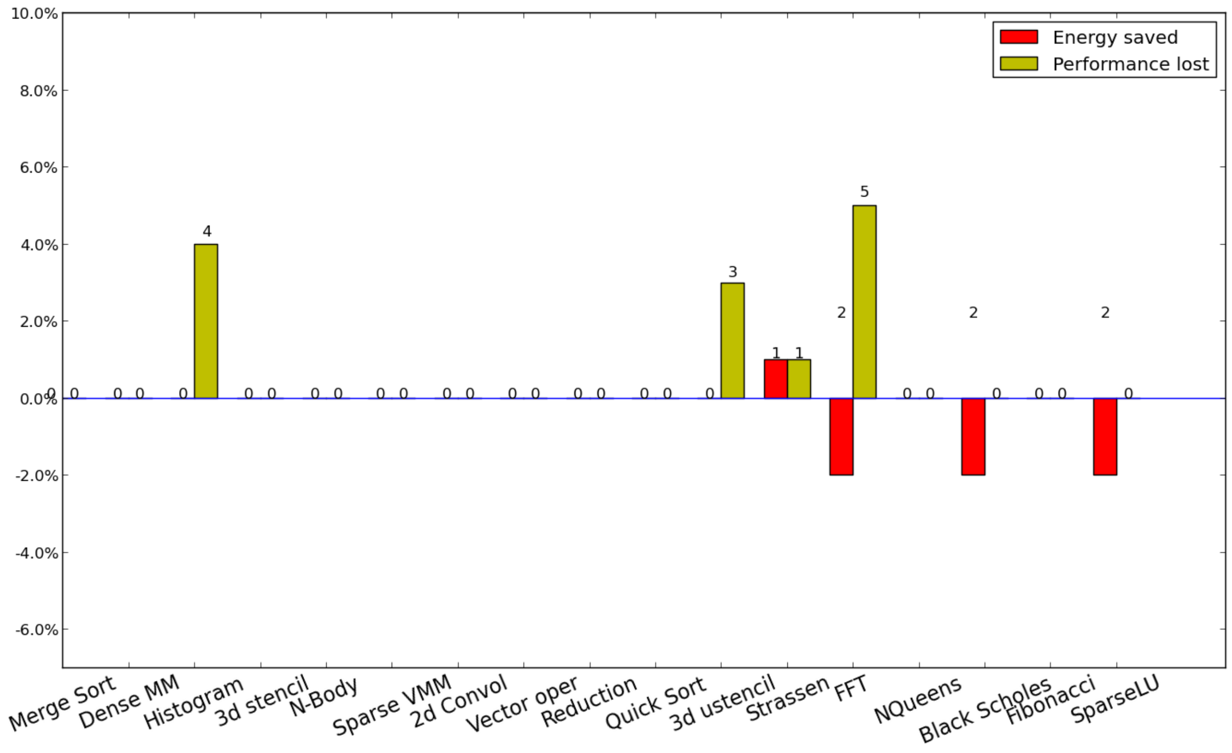
0.0.64 Results for 7 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2400 MHz	2430 MHz	1.25 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2412 MHz	0.50 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2400 MHz	2600 MHz	8.33 %
Unstructured 3d stencil	2600 MHz	2437 MHz	6.27 %



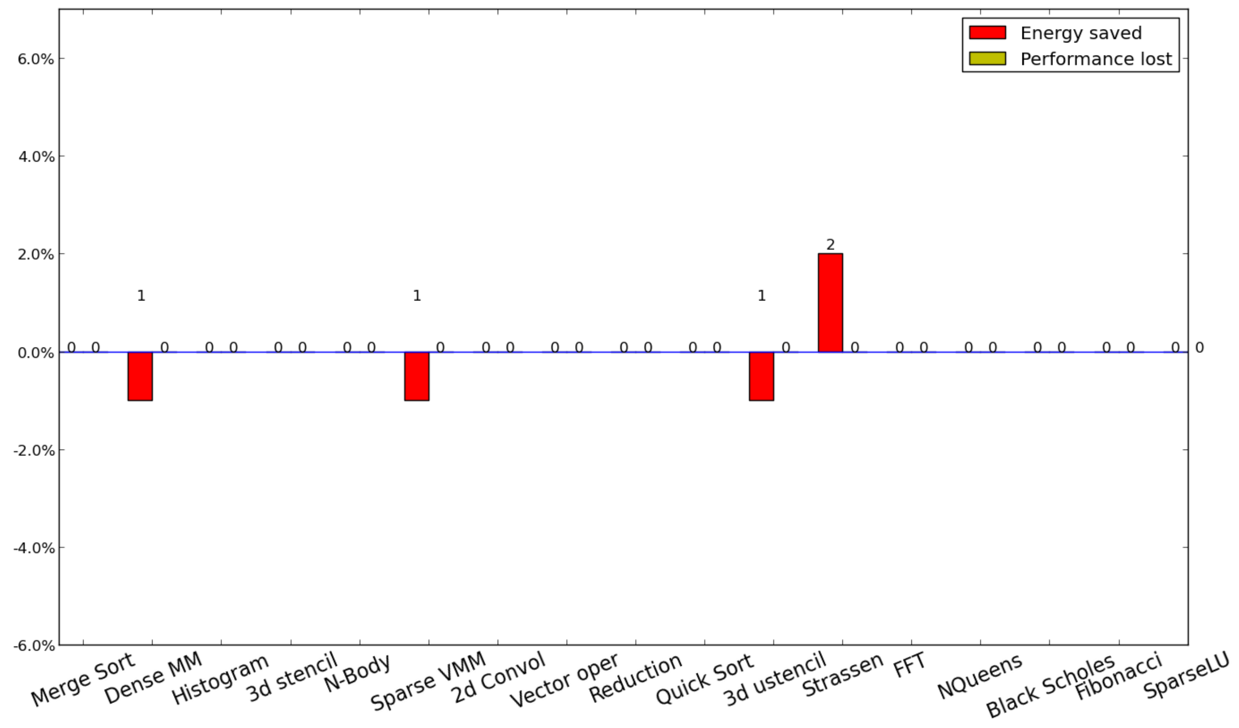
0.0.65 Results for 6 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2433 MHz	6.42 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2417 MHz	7.04 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2500 MHz	2435 MHz	2.60 %



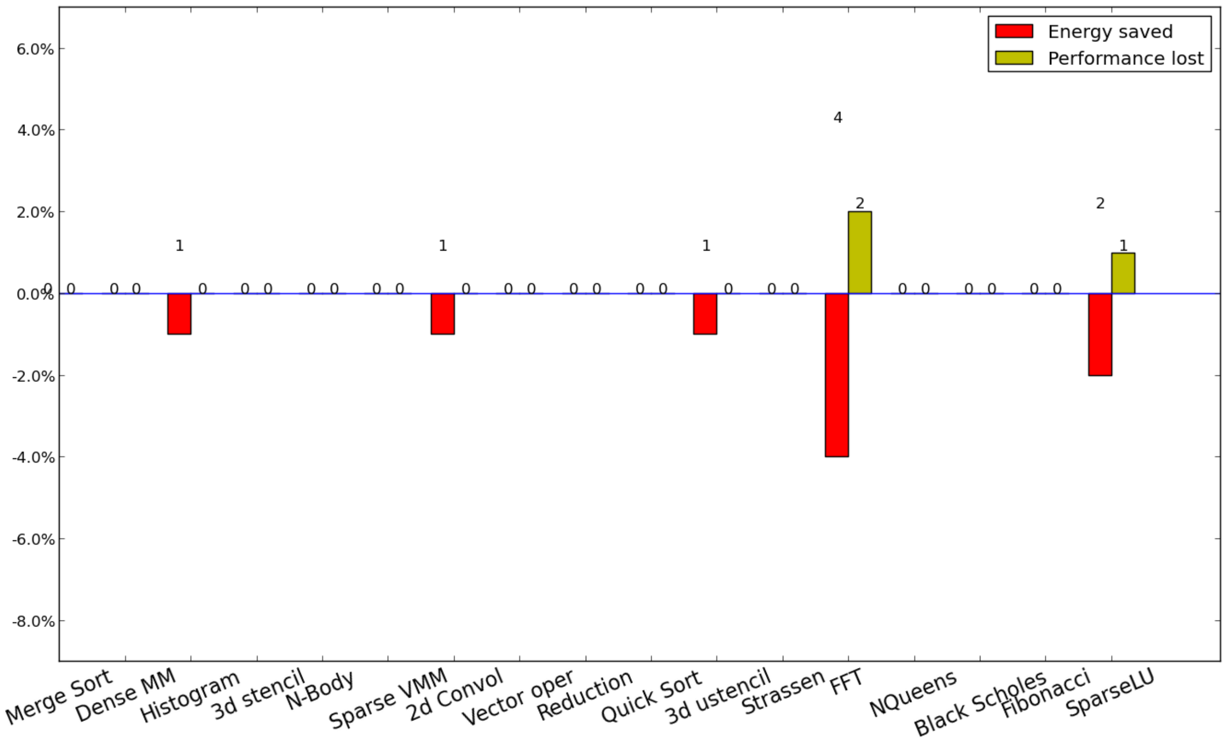
0.0.66 Results for 5 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2500 MHz	2600 MHz	4.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2400 MHz	2600 MHz	8.33 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2200 MHz	2600 MHz	18.18 %
Unstructured 3d stencil	2600 MHz	2600 MHz	0.00 %



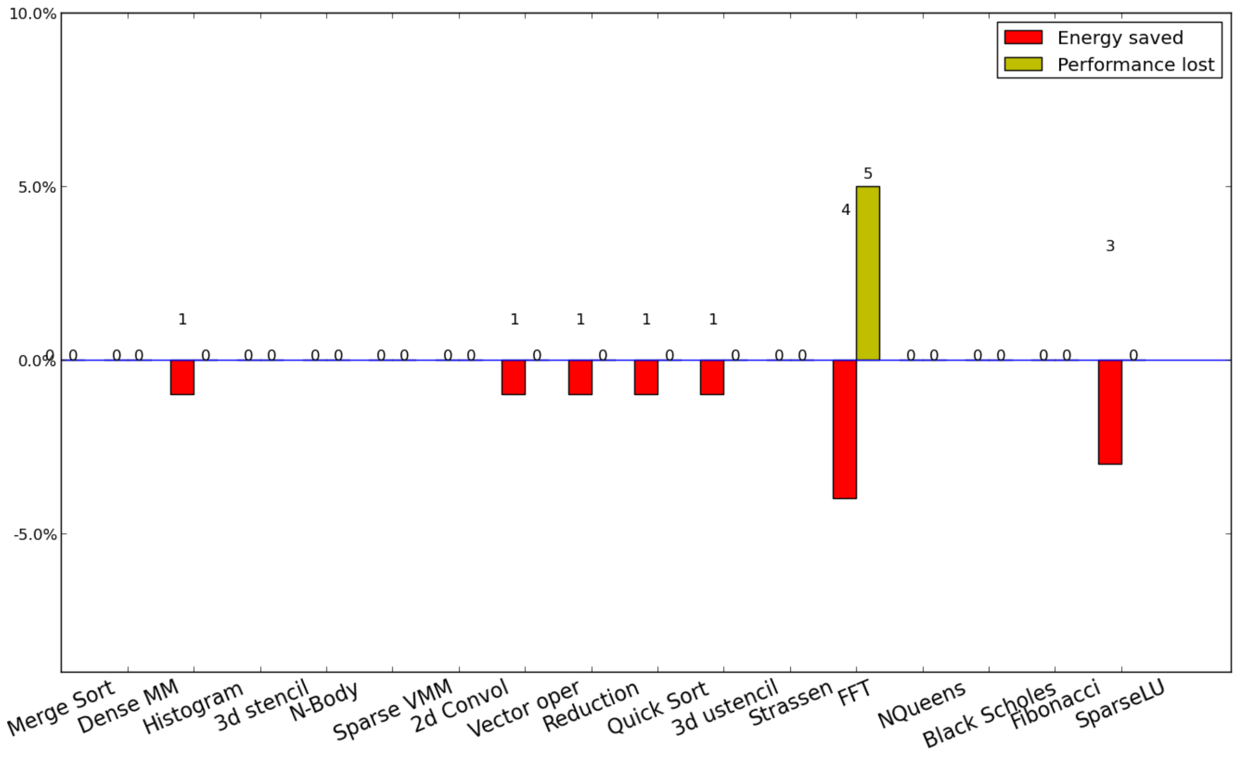
0.0.67 Results for 4 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2500 MHz	2600 MHz	4.00 %
Quick Sort	2400 MHz	2600 MHz	8.33 %
Unstructured 3d stencil	2600 MHz	2600 MHz	0.00 %



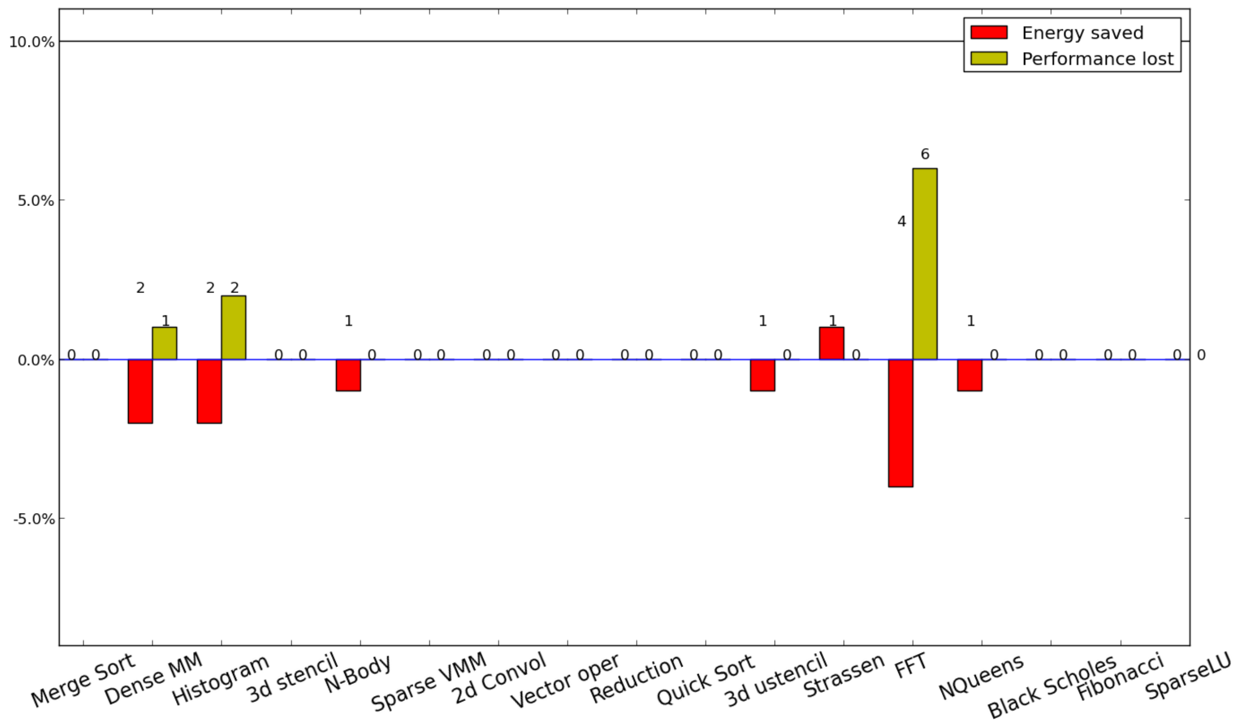
0.0.68 Results for 3 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2100 MHz	2600 MHz	23.81 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2600 MHz	2600 MHz	0.00 %



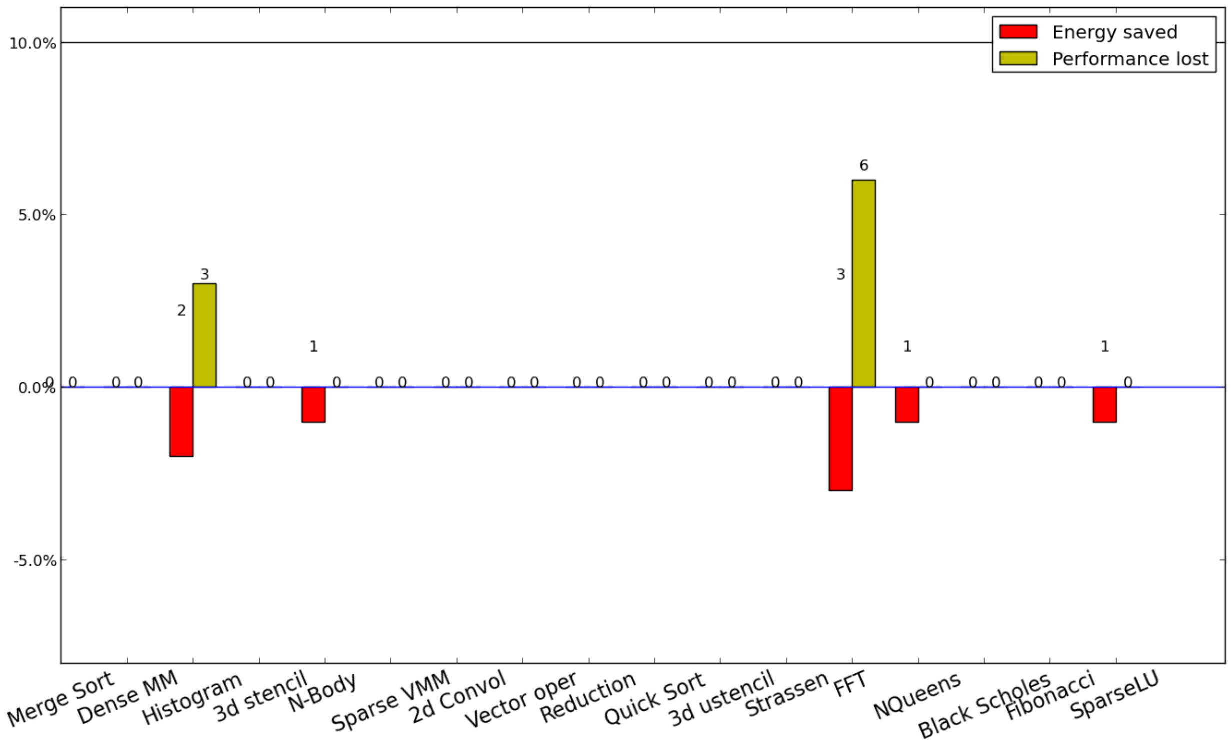
0.0.69 Results for 2 threads

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2600 MHz	2600 MHz	0.00 %



0.0.70 Results for 1 thread

Kernel	Optimal Frequency	Predicted Frequency	Error
Merge Sort	2600 MHz	2600 MHz	0.00 %
Reduction	2600 MHz	2600 MHz	0.00 %
Histogram	2600 MHz	2600 MHz	0.00 %
2d convolution	2600 MHz	2600 MHz	0.00 %
3d stencil	2600 MHz	2600 MHz	0.00 %
Dense matrix multiplication	2600 MHz	2600 MHz	0.00 %
Sparse matrix vector multiplication	2600 MHz	2600 MHz	0.00 %
Vector operation	2600 MHz	2600 MHz	0.00 %
N-body	2600 MHz	2600 MHz	0.00 %
FFT	2600 MHz	2600 MHz	0.00 %
SparseLU	2600 MHz	2600 MHz	0.00 %
NQueens	2600 MHz	2600 MHz	0.00 %
Strassen	2600 MHz	2600 MHz	0.00 %
Black Scholes	2600 MHz	2600 MHz	0.00 %
Fibonacci	2600 MHz	2600 MHz	0.00 %
Quick Sort	2600 MHz	2600 MHz	0.00 %
Unstructured 3d stencil	2600 MHz	2600 MHz	0.00 %


Energy saved
Performance lost
Merge Sort
Dense MM
Histogram
3d stencil
N-Body
Sparse VMM
2d Convol
Vector oper
Reduction
Quick Sort
3d ustencil
Strassen
FFT
NQueens
Black Scholes
Fibonacci
SparseLU