ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ Η/Υ

ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΑΞΙΟΛΟΓΗΣΗ ΜΕΤΡΟΠΡΟΓΡΑΜΜΑΤΩΝ ΓΙΑ ΚΑΡΤΕΣ ΓΡΑΦΙΚΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΡΙΣΚΑΣ ΒΑΣΙΛΕΙΟΣ

Επιβλέποντες : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Xavier Martorell
Associate professor, UPC, Barcelona

Αθήνα, Σεπτέμβριος 2012

ΠΕΡΙΛΗΨΗ

Ο Παράλληλος προγραμματισμός είναι γνωστός από παλαιότερα. Πίσω στις δεκαετίες του '50 και του '60 χρησιμοποιήθηκε από τους ειδικους στα πλαίσια των εφαρμογών για υπερυπολογιστές για να επιταχύνει επιστημονικές και άλλες εφαρμογές. Απο τα μέσα της περασμένης δεκαετίας ήρθε ξανά στο προσκήνιο, όταν η Intel ακολούθησε την IBM και τη Sun Microsystems και ανακοίνωσε τον πρώτο διπύρηνο επεξεργαστή. Όσο η ανάγκη για περισσότερη υπολογιστική ισχύ εντείνεται, χρησιμοποιούμε όλο και περισσότερες και διαφορετικές υπολογιστικές μονάδες. Οι κάρτες γραφικών είναι το νέο ατού στον κόσμο της υψηλής υπολογιστικής ισχύς αφού έχουν για αρκετές εφαρμογές πολύ καλύτερη απόδοση από τους κλασικούς επεξεργαστές. Γι' αυτούς τους λόγους χρειαζόμαστε και νέα, πιο βολικά στη χρήση, προγραμματιστικά εργαλεία. Η NVIDIA λάνσαρε την CUDA, το προγραμματισικό της εργαλείο για τις κάρτες γραφικών της, κάνοντας τον προγραμματισμό για κάρτες γραφικών πιο έυκολο απο ποτέ. Στο supercomputing κέντρο της Βαρκελώνης (BSC) αναπτύσσουν από το 2007 το OmpSs ένα προγραμματιστικό μοντέλο για προγραμματισμό σε ετερογενείς πλατφόρμες (συνδυασμό πλατφόρμων κοινής μνήμης (shared memory) και καρτών γραφικών). Σε αυτή τη διπλωματική αναπτύξαμε ένα σετ από μετροπρογράμματα (Rodinia) με το OmpSs και συγκρίναμε απόδοση και ευκολία στον προγραμματισμό σε σχέση με υπάρχοντα εργαλεία (CUDA, OpenCL). Η διπλωματική εκπονήθηκε στο πολυτεχνείο της Βαρκελώνης (UPC) στα πλαίσια του προγράμματος Erasmus.

.....

ΠΡΙΣΚΑΣ ΒΑΣΙΛΕΙΟΣ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ Η/Υ

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω τον καθηγητή Νεκτάριο Κοζύρη και τον λέκτορα Γεώργιο Γκούμα που με ενθάρρυναν για την εκπόνηση της διπλωματικής στο πολυτεχνείο της Βαρκελώνης στα πλαίσια του προγράμματος Erasmus. Επίσης τον επιβλέποντα καθηγητή Xavier Martorell για τη συνεχή στήριξη κατα τη διάρκεια της εκπόνησης.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Multi-core architectures[10–15]

Parallel computing is old. You can trace the oldest parallel computers back to the late 1950's. In the 60's and 70's you could find supercomputers that used multiple processing elements and parallel software to achieve greater speed. But these were specialized machines for only the most critical (and best-funded) applications. For years, processor manufacturers consistently delivered increases in clock rates so that single-threaded code executed faster on newer processors with no modification. Moore's law proved to be true for more than half a century but the law cannot be sustained indefinitely. In 2003 Intel predicted the end would come between 2013 and 2018 and it is noted that transistors would eventually reach the limits of miniaturization at atomic levels. Parallel computation has recently become necessary to take full advantage of the law. The computing industry changed course in 2005 when Intel followed the lead of IBM and Sun Microsystems and announced that its high performance microprocessors would rely on multiple cores. An individual core is a distinct processing element and is basically the same as a CPU in an old single-core PC. When we integrate the cores onto a single integrated circuit die (CMP or chip multiprocessor) we get a multi-core chip. Today we have quad-core processors (with four cores) or octa-core processors (with 8 cores) but the tendency is to see the numbers to rise and in a few years we may easily have hundreds of cores in an ordinary PC. The type of connection between the different cores varies. Cores may or may not share caches and they may implement message passing or shared memory communication methods.

Multi-core processors are widely used for many applications including general-purpose, network and embedded. Multimedia and scientific applications are also heavily benefited from the use of many cores as well as computer graphics. Pretty much anything that can be threaded today can map efficiently to a multi-core architecture. For example downloading software while running an anti-virus program can be managed by two threads which in their turn can be managed independently by a dual-core processor.

1.1.1 Supercomputers

You can trace the first supercomputers back in the 60's. The first supercomputer, the Control Data Corporation (CDC) 6600 had only one CPU and was quite small. It cost 8 million dollars back then and operated at up to 40 MHz and having a peak performance of 3 million floating point operations per seconds (flops).
In comparison the IBM Sequoia a Blue Gene/Q supercomputer has a peak performance of 16.32 petaflops running on over 98000 nodes and containing on the whole 1572864 processor cores and 1.6 Petabytes of memory. Throughout their history, supercomputers remained the province of government-funded institutions. Due to the their role in nuclear weapons research their export was carefully controlled. Traditionally they were used for highly calculation-intensive tasks such as problems in climate research, quantum physics, weather forecasting and physical simulations (such as simulation of airplanes in wind tunnels). Traditionally the supercomputer meant a large number of processors are used in close proximity to each other to form a computer cluster. THe processors are connected by a local high-speed computer bus. In our days as communication networks become faster and faster new distributed technologies and ideas have emerged. Grid computing for example is a network where each computer's resources are shared with every other computer in the system. A grid computing system can be a collection of similar computers running on the same operating system. They can reach performance in the scale of PFLOPS comparable to the largest supercomputers. That's why the tendency for now is more to distributed platforms.

1.1.2 GPUs

Rendering graphics is an intensive task that, if pursued only by the CPU could cause performance to slow down. To offload this work we use the Graphics processing unit or simply GPU. A GPU is a dedicated processor optimized for accelerating graphics. The processor is specifically designed to perform floating-point calculations, which are fundamental to 3D graphics rendering and 2D picture drawing. The two main attributes of a GPU are the core clock frequency, which typically ranges from 250MHz to 4 GHz and the number of pipelines, which translate a 3D image characterized by vertices and lines into a 2D image formed bt pixels. That's the reason they were called graphics or video cards in the beginning. The term GPU was popularized by NVIDIA in 1999, who marketed the GeForce 256 as "the world's first GPU". GPUs are widely used in personal computers, game consoles and mobile phones.

(a) NVIDIA Tesla

FIGURE 1.1: Graphics Card

The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetics and memory bandwith that outpaces its CPU counterpart. The graphics chips started becoming increasingly programmable and computationally more and more powerful. In the late 90's computer scientists and

experts from various fields started using GPUs to accelerate a range of scientific applications which led to what we called GPGPU (General-purpose computing on graphics processing units). In the beginning one of the difficulties in programming GPGPU applications has been that despite their general purpose tasks having nothing to do with graphics, the applications still had to be programmed using graphics APIs like OpenGL and Cg. Although huge performance was achieved by scientists (over 100x compared to CPUs in some cases), this led to limited accessibility to the tremendous capabilities of GPUs.

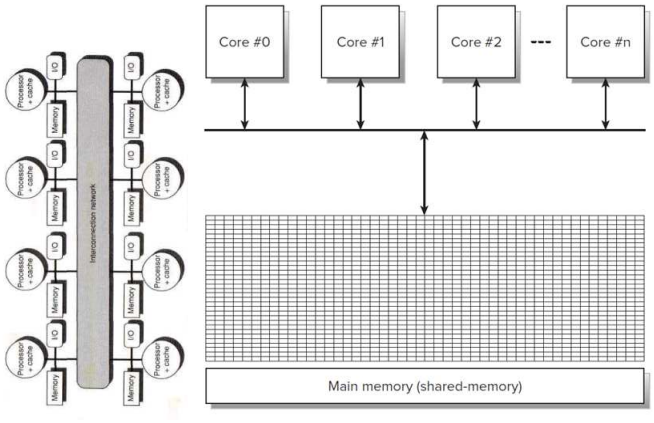
1.1.2.1 NVIDIA

NVIDIA, an American technology company founded in California in 1993, is the leading provider of GPUs in the global market. The GeForce brand dominated for about ten years in the technology market of Graphics Cards. In 2007 NVIDIA having recognized the potential of bringing this performance to the larger scientific commumity, invested in making the GPU fully and easily programmable and introduced CUDA. CUDA is a parallel computing platform that is accessible to software developers through variants of industry standard programming languages. Programmers use C with NVIDIA extensions to code algoritms for execution on the GPU. The GPU programming has recently been easier than ever. Third party wrappers are also available for Python, Perl, Fortran, Java, Ruby, Lua, Haskell, MATLAB. CUDA has revolutioned the GPU programming and is now widely deployed with thousands of GPU-accelerated applications and published research papers. Due to the architecture of the GPUs the programming units follow a single-instruction multiple-data (SIMD) programming model. For efficiency, the GPU processes many elements in parallel using the same instructions. Each element is independent from the other and elements cannot comminucate with each other. All GPU programs must be structured in this way: many parallel elements, each processed in parallel by a single program. For this reason CUDA is especially efficient to applications that are explicitly parallel in nature. These include Fast Video Transcoding, Video Enhancement, Oil and Natural Resource Exploration, Medical Imaging, Computational Sciences, Neural Networks, Gate-level VLSI Simulation, Fluid Dynamics and others. Recenlty it has been increasingly used also in computational finance. We will see in next chapters some programming details of CUDA.

1.1.3 Multi-core Processors

As mentioned in the beginning, the trend of increasing the clock speed of a processor to gain in performance has become a way of the past. Moore's law, that indicates that the number of transistors will double every 2 years is close to its end. Throughout the previous years frequency meant performance. But since the frequency reaches its peak we must now consider other aspects of the overall performance such as power consumption

and number of cores. Multiple processors seem to give an answer to the problems of single core processors, by increasing bandwith while decreasing power consumption. Multicore architectures vary greatly in different machines. First we have differences in communication and memory configuration. Below you can see the two main categories of memory communication.



(a) Message Passing (b) Shared Memory

FIGURE 1.2: Memory models

Another thing is if multicore processors should be homogenous (are all exactly the same) or heterogenous (that contains cores with different frequencies, cache sizes, functions). Homogenous architectures are off course easier to produce but it's not sure that they are the most efficient use of the existing technology. Heterogenous environments with specific functionality for each element are more complex but probably much more practical and efficient. Experts think that by 2017 a desktop chip could use 128 cores. As multi core architectures have now become mainstream and dominant in the market the important issue for the computing industry is to use the multicore processor to its full potential. If programmers do not write applications that take advantage of the parallel architecture there is no gain in performance. The member of Intel Shekhar Borkar stated in 2007 "The software has to aslo start following Moore's Law, software has to double the amount of parallelism that it can support every two years". In the end programmers have to learn how to write parallel programs that can be split up and run concurrently on multiple cores.

1.2 Parallel programming

As we saw above to exploit the parallel computer architecture programmers have to write applications that can be split up and run concurrently on multiple cores. Parallel programming has been used for many years, mainly in high-performance computing. But today, with the multi-core processors dominating the world IT market parallel programming is becoming mainstream. Parallel computer programs are more difficult to write and debug than the sequential ones. We have new software bugs introduced such as the race conditions and the communication and synchronization of the different tasks are matters that affect performance. The maximum theoritical speed-up of a program is known as the well known Amdahl's law. If P is the proportion of a program that can be made parallel and (1-P) is the proportion that cannot be parallelized the maximum speedup that we can have using N processors is

$$\frac{1}{(1-P)+\frac{P}{N}}$$

. But regardless of that achieving good performance in parallel programs, is a tricky matter and requires skill and experience. Regarding memory the two main models of parallel programming are shared and distributed memory. In shared memory the programs (meaning the corresponding CPUs) have access to the same shared memory. The shared memory model makes programming much easier as it does not require any communication between the the different tasks. On the other hand it introduces bugs such as race conditions and it can be used only to shared-memory hardware platforms. Distributed memory model refers to independent tasks with their own memory and data. The distributed model is used for distributed memory architectures (although it can also be used in shared-memory architectures), it gives much better scaling in parallel algorithms but it makes programming much more complicated. Data has to be distributed to be operated and communication between the tasks has to be explicit. Realizing the challenges of the recent hardware developments the software world has been very active in the evolution of parallel programming. Especially in the last decate new programming tools and APIs have been developed to make programming in parallel easier. The most popular tools are developed for the programming models that we saw above. In the next chapter we will see the programming tools that have been used for the development and for the evaluation of the parallel programs of the Rodinia Benchmark.

The tools are OpenMP, OpenCL, CUDA and OmpSs. We make a reference to MPI as well.

1.2.1 MPI

Correspoding to the distributed memory model the parallel programming protocol MPI has been introduced since 1994. MPI is language-dependent protocol which allows point-to-point and collective communication. MPI is the "de facto" industry standard for message massing and is the dominant model used today in high performance computing. Its main advantages that make it widely popular is that it offers high performance, scalability and portability. MPI has been implemented for almost every distributed memory architecture. In addition many versions of MPI are freely available to the public and is well documented. MPI has bindings for C/C++, Fortran, Python, OCaml. MPI library functions include, point-to-point send/receive operations, exchanching data between process pairs, sychronizing nodes (barrier), obtainining information about the network (number of processes, neighboring processes, current process ID) and many others. The MPI library contains about 500 functions. MPI has a sophisticated runtime system which:

- Launches the MPI application's processes. This role is shared between the runtime and the parallel machine scheduling mechanism.
- It makes the necessary connections between the processes depending on the network used and the hardware. It also distributes the information through an out-of-band messaging system.
- Controls the MPI processes: ensures that in case of a crush the entire environment is cleaned. Depending on the operating system termination signals are forwarded to the remaining processes.

1.2.2 OpenMP

Many vendors were delivering shared memory multiprocessors from the mid 80's till the mid 90's. They provided directive-based Fortran extensions to take advantage of the underlying

architecture. The problem in the field was the lack of standardisation in shared memory directives. Each vendor did its own thing. The OpenMP Fortran standard was released in October 1997 to cover the need of a standard protocol. OpenMP is the dominant API right now that supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran, covering most of the architectures and operating systems. OpenMP offers a simple and easy-to-program interface for developing parallel applcations from the standard desktop computer to supercomputers. In the pseudocode below we can see how easily we can parallelize a for loop. With a simple parallel for directive the loop is split up into equal portions and each thread executes each portion. Or in the next figure how we can support task parallism with each thread executing each separate task. The task directive was added in one of the next releases of OpenMP and it added significant flexibility to the ways OpenMP can be used. With the task directive we can parallelize recursive calls for example.

Program 1 OpenMP parallel for

```
#pragma omp parallel for shared(a)
for (i = 0; i<N; i++)
{
    a[i] = ...
}
```

Program 2 OpenMP parallel task

```
#pragma omp parallel
{
    #pragma omp task
    function1();

    #pragma omp task
    function2();
}
```

OpenMP offers many advantages as a platform of parallel programming.

- It's simple. It does not require communication between threads as MPI. code is portable.
- Data decompositions is handled automatically

But it shows some disadvantages also.

- Only runs in shared memory architectures.
- In some problems scalability is limited due to memory bound algorithms
- Cannot be used with GPUs.

OmpSs, the tool developed in the Barcelona Supercomputing Center is an extension of the OpenMP compiler and it tries to solve some of the issues we have with OpenMP.

1.2.3 Heterogenous Computing

Hardware accelerators are computer hardware used to perform some spesific task faster than on a general-purpose CPU. Examples of hardware accelerators are:

- The GPUs used for graphics processing.
- FPGAs (Field-programmable gate array) integrated circuits designed to be programmed by designers implementing logical functions.
- DSPs which are specialized microporcessors used in digital signal processing.

As they are giving dramatic performance gains and are becoming easier to program hardware accelerators have recently become popular. Due to the fact that they differ significantly from CPUs in architecture they require different programming models as well. For example in GPUs a new programming model was used to exploit the large scale parallelism they give. Programming heterogenous machines can be harsh since you have make the best use of the hardware's characteristics. This is still a responsibility of the programmer and it makes the code more complex. Fore example hardware specific code has to be included between normal application code which is executed in the CPU. Balancing the workload is also difficult as the performance of the processors is quite different.

1.2.3.1 OpenCL

Due to the increasing interest in heterogenous platforms there was a need for a unified programming model that could be used for cross-platform execution that would make the programmer's life easier. The framework OpenCL partially fulfiled that task. OpenCL includes a language for writing kernels (functions that execute on different devices) and it contains APIs to control these platforms. OpenCL is used mainly for executing code in CPUs and GPUs but some progress has been done for tools that translate OpenCL to run on FPGA devices. OpenCL is an open standard and it has been adopted by Intel, AMD, NVIDIA and ARM Holdings. And that is its main weapon against CUDA. CUDA is a tool specifically targeted to NVIDIA GPUs and that restricts its use. For example products like Photoshop can take a huge advantage of parallel processing in GPUs because that is what GPUs have been designed for. To deal with images. However, it is hard to introduce a product that will only get accelerated if the consumer has a specific brand of GPU. On the other hand OpenCL is a lower level API than CUDA and the code is much more complicated, a thing that makes its use quite restricted. In addition, despite the portability that OpenCL guarantees usually the code has to be changed when we use different heterogenous devices. That is why the implementation of an algorithm changes a lot in different architectures.

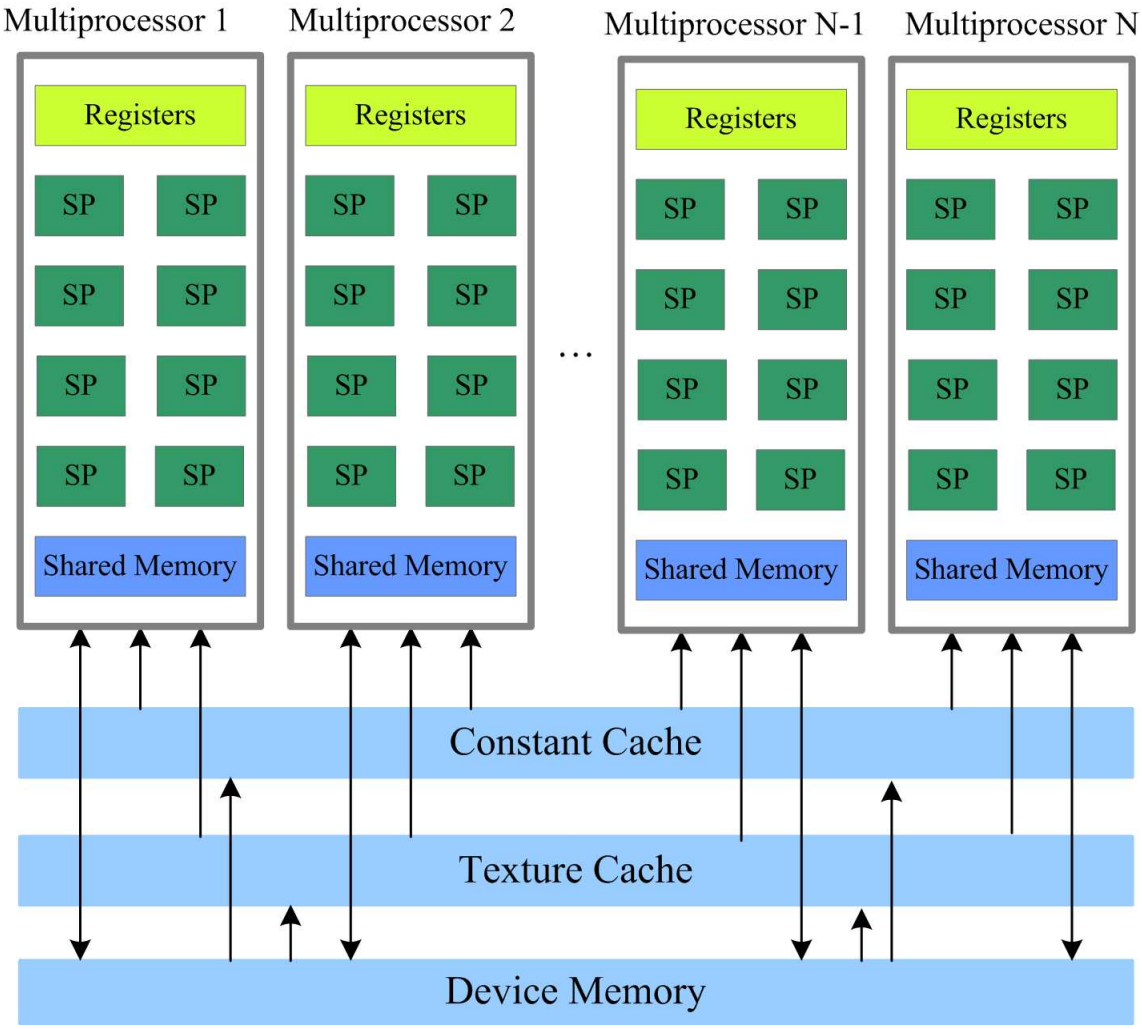
1.3 CUDA[6, 9, 10, 12]

We saw in the previous chapter how the high arithmetic throughtput from the early days of the GPUs led scientists to consider programming GPUs an attractive idea. But in the beginning of the GPU programming era the programmer has serious matters to solve. First of all he had limitations in where he could write results to memory, so algorithms that required to write to random memory locations were not able to run on the GPU. Secondly anyone who wanted to use a GPU had to learn OpenGL or DirectX since they were the only methods to communicate with the GPU back then. This meant that programmers had to execute computations calling OpenGL or DirectX functions and writing the computation parts in special - graphics programming languages. The latter was a serious bottleneck of the GPU programming expansion. In the crucial, for GPU computing, date of November 2006 NVIDIA announced the GeForce 8800 GTX the first GPU to be built with NVIDIA's CUDA Architecture. The architecture introduced several new components specially designed for general purpose GPU computing and aimed to make programming in the GPUs easier and more attractive even for non-specialists. In this chapter we will see in more detail the CUDA architecture and programming model.

1.3.1 CUDA architecture

The typical modern architecture of an NVIDIA card can be seen in the figure below. It is constructed by an array of many-threaded multiprocessors (SMs) which contain a number of stream processors (SPs). Two SMs form a building block although the number can vary from card to card. Each GPU multiprocessor has its own memory and texture filter(TF) and each pair of multiprocesors (building block) has a shared L1 cache. Each GPU has up to 8 GB of double data rate DRAM which we call the global memory. The memory of the GPU differs from the CPU memories in that it is basically a frame buffer memory used for graphics. The GPU memory is very-high bandwith off chip memory with slightly more latency than typical memories. For massively parallel applications the higher bandwidth compensates for the longer latency. The G80 graphic card, the first one with CUDA architecture, had 86.4 GB/s of memory bandwidth and a 8 GB/s communcation bandwidth with the CPU (4 GB/s from the system to the device and 4 GB/s the opposite direction). The communication bandwidth may seem like a

factor that could limit the performance but as it comparable to the CPU bus bandwidth it is not so important in the end. These numbers refer to G80 and as we will see later are bigger for the cards that we see today. The Tesla M2090 has 512 SPs (16 SMs, each with 32 SPs). Each SP has a multiply-add (MAD) unit. With 512 SPs it has a total processing power of 1.33 TFLOPs. Since each SPs has multiple threads a normal application that runs on a GPU could have a few thousands of threads executing simultaneously. We can see now why we can exploit the GPU for massively parallel applications.



(a) NVIDIA Tesla

FIGURE 1.3: Graphics Card

1.3.2 CUDA programming

In this section we will see some basic elements of CUDA programming - at least those which we have encountered in the Rodinia Benchmark suite. A CUDA program consists of one or more parts that are executed on the host (CPU) or on the device (GPU). The parts that require to be split and execute in parallel (the computationally more expensive parts) are implemented in the device code while the parts that exhibit little or no data parallelism are implemented in the host code. The host code is implemented in C/C++ and is compiled and run as a usual CPU process. The device code is written in an extended version of C/C++ where we can write data-parallel functions called kernels. It is compiled by nvcc, the special compiler designed and provided by NVIDIA.

1.3.2.1 Kernels

In CUDA the programmer can define C functions, called kernels, which are executed many times in parallel by different device threads. That means that CUDA follows the SIMD (Single Instruction Multiple Data) model. We write a single function that is executed multiple times by different threads. Below we can see how a simle kernel looks like. We have the global keyword which declares the function to be used in the device. The function takes the three vectors as arguments as in a regular C function. Each of the threads that execute the kernel have a unique ID that is given by the threadIdx variable. The example below shows the addition of two vectors. Each thread is responsible for one element of the array.

Program 3 Vector addition kernel

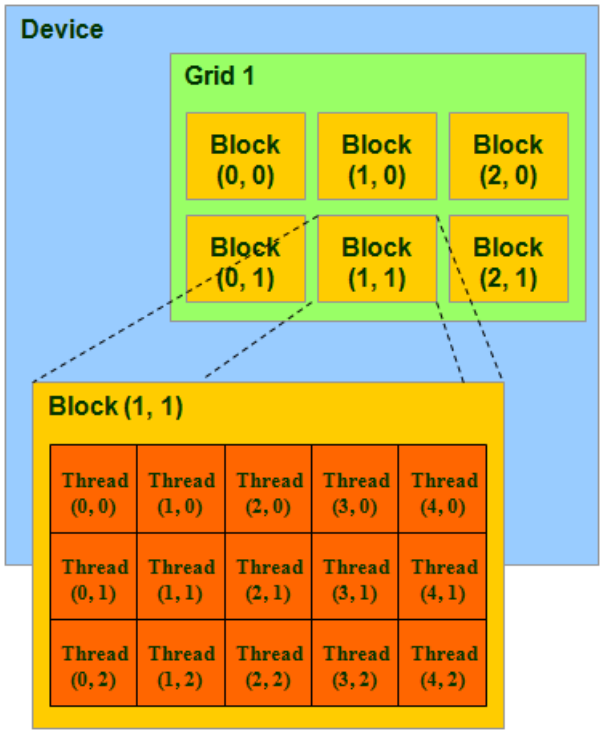
```
__global__ void AddVectors(float *A, float *B, float *C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];

}

void main()
{
    //kernel call
    AddVectors<<<N,1>>>(A,B,C);
}
```

The first attribute in the kernel call denotes the size of the grid. It says how many blocks will execute the kernel. The second contains the number of threads available for each block. Both can support three-component vectors and the maximum values are different for each GPU. We have to note that the numbers of blocks and threads can significantly outnumber the number of the actual GPU threads or blocks. The number is typically dictated by the size of the data processed and not by the number of processors.

The multiple blocks can be organized into one, two or a three-dimensional grid of thread blocks and the same applies for the threads within a block. Below we can see an example of two-dimensional matrices addition. Each block in a grid has a unique ID which can be accessed in the kernel by the blockIdx variable. The dimension of the thread block is accessible by the blockDim variable. In the example below the matrix has been split into a two dimensional grid of blocks and each block has its threads organized also in a two dimensional matrix.



(a) NVIDIA Tesla

FIGURE 1.4: Graphics Card

Program 4 Vector addition kernel

```
__global__ void AddMatrix(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if ((i < N) && (j < N))
        C[i][j] = A[i][j] + B[i][j];

}

int main()
{
    //kernel call
    dim3 dimBlock(10,10);
    dim3 dimGrid((N + dimBlock.x -1) / dimBlock.x,
     (N + dimBlock.y -1) / dimBlock.y);
    AddMatrix<<dimGrid,dimBlock<>>>(A,B,C)

}
```

1.3.2.2 Communication

Although it has been mentioned that explicit communication is not possible between threads there are two ways of implicit communication between threads of the same block.

- syncthreads: acts as a barrier for the threads in a block. Different threads must wait for all of them to finish executing until the point of syncthreads in order to proceed.
- Shared memory.

We will see some details of shared memory in the following chapter.

1.3.2.3 CUDA Memories

One of the common bottlenecks in parallel computing is the memory access. Slow memory performance will degrade the overall speed even if the computation itself is very fast. In the CUDA versions of the Rodinia benchmarks we often had optimizations

related to memory. The types of memory used in the benchmarks to accelerate memory accesses are:

- Constant Memory is a read only memory from kernels and is optimized for the case when all threads read from the same location. Constant memory provides one cycle of latency even though constant memory resides in the device memory. The cost scales linearly with the number of different addresses read by all threads. The constant cache is written by the host with cudaMemcpyToSymbol and is avaiable for all the kernel calls in the same application. Up to 64KB of data can be placed in the constant cache in current GPUs.

- Texture Memory: A texture unit is a component in modern GPUs to accelerate frequently performed operations such as mapping. Each on-chip texture unit has some internal memory taht buffers data from global memory. CUDA provides mechanisms so that programmers can exploit the capabilities of the texture memories that come with the GPU. The best performance will be achieved when threads of a warp read locations that are close together in a space context. When the memory references are close caching in texture memory can provide a large performance increase.

- Shared Memory: Shared memory is an on-chip memory that is much faster than the global memory space. Memory accessing on the shared memory can be as fast as accessing a register. However, if two addresses of a memory request fall in the same memory bank there is a bank confict and the access is serialized. To get maximum performance it is important to schedule the memory requests to minimize the bank conflicts.

1.4 OmpSs Environment[1, 2, 5]

This thesis was about developing and evaluating the Rodinia set of Benchmarks with OmpSs. In this chapter we will see the OmpSs programming model, Mercurium and Nanos++, the three of which comprise the environment of this project.

1.4.1 OmpSs programming model

OmpSs is a parallel programming model, developed by the BSC team, which covers the different homogenous and heterogenous architectures and might be extensible to future ones. It is an extension of the OpenMP model with a few changes: different execution model, some new constructs that have been added which were based on StarSs and without the parallel construct of OpenMP. We saw above that OpenCL has also been proposed as an open standard for programming in heterogenous architectures. However the low-level programming of OpenCL makes it complicated for the average programmer. OmpSs aims having the portability and low-level access of OpenCL but without its programming difficulties.

1.4.1.1 Execution Model

OmpSs is different from OpenMP in its execution model. OmpSs has a thread-pool model where all the threads exist from the beginning of the execution in constrast to the fork-join model of OpenMP. We still have one master thread which starts executing the code while the rest of the threads are available to execute tasks whenever they are called to. The master thread creates work for the rest with the regular OpenMP worksharing and task constructs.

As the team of threads are available from the start of the execution OmpSs makes the use of the parallel construct of OpenMP useless.

1.4.1.2 Extensions

OmpSs is extended with a set of clauses that can be added in the task construct to express data dependencies and to specify execution in heterogenous devices. Four new constructs are added that came from the StarSs model to speficy data dependencies: input, output, inout and inout-set. These four accept an expression that must evaluate to a set of lvalues. An lvalue is an expression that refers to a specific object.

- Input: If a task has an input clause that evaluates to a given lvalue, then the task will not run until a previous task with an output clause which has the same lvalue has finished its execution.

- Output: If a task has an output clause that evaluated to a given lvalue, then tha tsk will not run until a previous task with an input or output clause which has the same lvalue has finished its execution.
- Inout: If a task has an output clause that evaluated to a given lvalue, then it is as it had both an input and output clause that evaluated to the same lvalue.
- Inout-set: If a task has an inout-set clause that evaluated to a given lvalue, then it is as it had an inout clause that evaluated to the same lvalue but it will not create any dependencies with previously created tasks that have an inout-set clause with the same lvalue.

In order to support heterogenous computing we have a new construct: the target. We the target construct we can specify that a given task or worksharing construct can be run in a set of devices. The syntax is shown below:

Program 5 Vector addition kernel

```
#pragma omp target [clauses]
task construct | worksharing construct | function definition | function header
```

For the target construct we have the following clauses:

- device: We specify on which deviced will the construct be targeted (cell,cuda,smp). If no device is specified then the runtime system decides the target device.
- copy in: It is specified that a set of data may be needed to be transferred to the device.
- copy out: It is specified that a set of data may be needed to be transferred from the device to the host.
- copy inout: A combination of copy in and copy out.
- copy deps: It specifies that if the costruct has any dependence clauses then they also have copy semantics. (For example input will also mean copy in).
- implements: It specifies that the code is an implementation for the target devices of the function name given in the clause. These different implementations can be used instead of the original if the runtime considers it useful. (Note: the implements clause has not been implemented in the current version of OmpSs).

Program 6 Matrix Multiplication example

```
const int NB = 512;

#pragma omp task inout([NB*NB] C) input([NB*NB] A, [NB*NB] B)
void matmul_block(float * A,float * B,float * C){
// plain C kernel code for the SMP environment
}

#pragma omp target device(cell) copy_deps implements(matmul_block)
void matmul_block_cl(float * A,float * B,float * C){
// OpenCL kernel code
}

#pragma omp target device(cuda) copy_deps implements(matmul_block)
void matmul_block_gpu (float * A,float * B,float * C){
// CUDA kernel code
}

void matmul (int mDIM, int lDIM, int nDIM, float ** A, float ** B, float ** C){
     for(i = 0; i < mDIM; i++) {
         for (j = 0; j < nDIM; j++) {
             for (k = 0; k < lDIM; k++) {
                 matmul_block (A[i*lDIM+k],B[k*nDIM+j], C[i*nDIM+j]);
             }
         }
 }
 #pragma omp taskwait
```

1.4.2 Mercurium

Mercurium is a source-to-source compiler, developed in the Barcelona Supercomputing Centre and aims at fast prototyping. Currently it offers support for C/C++ and is used with the Nanos++ runtime system. It implements the OpenMP, OmpSs and StarSs programming models but it is quite extensible and it has been used to implement other programming models and compiler transformations such as Cell Superscalar (CellSs), Software Transactional Memory, Distributed Shared Memory, a few of them. The compiler recognizes the constructs and trasforms them into calls to the Nanos++ runtime system. The dataflow clauses are transformed into a set of expressions that will be evaluated when the program is executed. These expressions will generate addresses of memory that will be passed to the runtime system to build the task dependency graph. In addition, the compiler manages the code for different target devices. When the compiler generates the code for a task construct it has to look if there is a target directive

or an implement clause. If there is, the internal representation of the task is passed onto a device-specific "handler" for each non-SMP device. These "handlers" generate the device-specific data that go together with the task. They also generate an outline for the device which may be created in a separate file. This file is reintroduced in the compiler pipeline using a different compilation profile with different compilation tools (for example NVIDIA's nvcc for cuda devices). The binary output for the different files created is merged together into a single object file. This is done to make the compiler to keep its usual behaviour of creating one object file (to be compatible with other tools such as makefile). In the final linkage step the final binary is created.

1.4.3 Nanos

Nanos++ is an extensible and runtime library developed at BSC mainly aimed at providing support for the OmpSs parallel programming model, although it also supports in general task-based programming models such as OpenMP and Chapel. It is designed to deal with SMP architectures, GPUs and clusters of both GPUs and SMPs. The most important service of Nanos++ is to manage task parallelism with support for synchronization based on data-dependencies. Nanos is responsible for building the data-dependency graph. Each time the compiler creates a task it is submitted to the graph and it has its dependencies checked. In order to detect the dependencies the runtime keeps the addresses fo the arguments when dependency clauses exist. Nanos also provides support for efficiently keeping coherence across different address spaces, such as in GPUs. The Mercurium compiler is specialized in automatically generating the corresponding calls to the Nanos++ runtime.

Chapter 2

Goals and Methodology

As we saw before the existing programming models of heterogenous architectures expose the programmer in many low-level and hardware details making them quite complicated to use for non-experts. The OmpSs programming model was proposed to overcome these difficulties. However until now only a few applications were written in OmpSs. The goal of this project is:

- To show that OmpSs is a convenient tool to write applications for heterogenous environments and it makes parallel programming easier. As we will see further below OmpSs hides the memory management details of the GPU. The programmer states only the copies in and out of the device. We would also like to see if the automatic management of 2-GPUs that the runtime system provides is useful and what is its advantages.
- To show that the performance of the applications is better or at least similar with the existing models (with OpenCL and CUDA).

The OmpSs code that we developed was based on the CUDA versions that we already had. For each kernel call we had to specify the inputs and outputs of data for the GPU and make the necessary adjustments with the taskwait utility in order to keep the performance close to the CUDA version. The texture and constant memories that were used in the CUDA version had to be changed in global memory in OmpSs, because the compiler does not support them yet. This resulted in degration of performance in some cases. The CUDA kernels were optimized to take advantage of the characteristics of

the GPU, so we tried to keep them unchanged. In some cases such as the HeartWall example where the passing of pointer parameters is not possible in OmpSs we had to make some adjustments. In the following chapter these cases are explained.

Chapter 3

Design and Implementation

In this chapter we include pseudocode of the computation parts of each benchmark and the invocation of the CUDA kernels. The CUDA kernels themselves are not included.

3.1 Backprop

Back Propagation (BP), a neural network learning algorithm, is one of the most effective approaches to machine learning when processing image data. It trains the weights of connecting nodes on a layered neural network. The application is made of two phases: the Forward Phase, in which the activations are propagated from the input to the output layer, and the Backward Phase, in which the error between the observed and requested values in the output layer is propagated backwards to adjust the weights and bias values. In each phase, the processing of the nodes can be done in parallel.

```
GPU kernel call
 #pragma omp target device(cuda) copy_in( myiunits[0;elements_in1], \
                 input_weights_one_dim[0;elements_in2]) \
           copy_out  \
           ( partial_sum[0;elements_out])
 #pragma omp task firstprivate(in,hid,num_blocks)

  {
   dim3  grid( 1 , num_blocks);
   dim3  threads(32 , 32);
     printf("just before the kernel\n");
   bpnn_layerforward_CUDA<<< grid, threads >>>(myiunits ,  input_weights_one_dim, partial_sum,
      in, hid);
 }
 #pragma omp taskwait
```

```
//the layerforward and adjust_weights are done in the host because they are very short in
    computation time

  bpnn_layerforward(net->hidden_units, net->output_units, net->hidden_weights, hid, out);
  bpnn_output_error(net->output_delta, net->target, net->output_units, out, &out_err);
  bpnn_hidden_error(net->hidden_delta, hid, net->output_delta, out, net->hidden_weights, net->
    hidden_units, &hid_err);
  bpnn_adjust_weights(net->output_delta, out, net->hidden_units, hid, net->hidden_weights, net
    ->hidden_prev_weights);


//GPU kernel call
 #pragma omp target device(cuda) copy_in( \
            [elements_in1] myhidden, \
            [elements_in2] input_weights_prev_one_dim, \
            [elements_in3] input_weights_one_dim) \
            copy_out( \
             myiunits[0;elements_out1], \
             input_weights_one_dim[0;elements_out2])
  #pragma omp task firstprivate(num_blocks,in,hid)
  {
  dim3  grid( 1 , num_blocks);
  dim3  threads(32 , 32);

  bpnn_adjust_weights_cuda<<< grid, threads >>>(myhidden,hid, myiunits, in,
    input_weights_one_dim, input_weights_prev_one_dim);
  }
 #pragma omp taskwait
```

Listing 3.1: The final implementation of BackProp

In BackProp we create two tasks with a kernel invocation in each one. Both use the taskwait directive.

3.2 Heart Wall

The Heart Wall application tracks the movement of a mouse heart over a sequence of 104 609x590 ultrasound images to record response to the stimulus. In its initial stage, the program performs image processing operations on the first image to detect initial, partial shapes of inner and outer heart walls. These operations include: edge detection, SRAD despeckling, morphological transformation and dilation. The final stage of the computation is the Heart Wall Tracking.

```
#pragma omp target device(cuda) copy_in( common_in[0;1], common_input1[0;20],common_input2
    [0;20] ,common_input3[0;31], common_input4[0;31]     )
        copy_out(common_output1[0;elements_out12], common_output2[0;elements_out12],
    common_output3[0;elements_out34], common_output4[0;elements_out34] )
#pragma omp task
{
  dim3 grid1(1,1);
  dim3 threads1(1,1);
  init<<<grid1,threads1>>>(common_input1,common_input2,common_input3,common_input4,
    common_output1,common_output2,common_output3,common_output4,common_in);
```

```
}

#pragma omp taskwait noflush

   .
   .
   .

for(common_change.frame_no=0; common_change.frame_no<frames_processed; common_change.frame_no
    ++){

  frame = get_frame(frames,common_change.frame_no,0,cropped0,1);

  common_change_input = frame;
  common_change_input_elem = common.frame_elem;
  frameno_in = common_change.frame_no;

  #pragma omp target device(cuda) copy_in(common_change_input[0;common_change_input_elem])
  #pragma omp task firstprivate (num_threads,block_num,frameno_in)
  {
    dim3 blocks2(block_num,1);
    dim3 threads2(num_threads,1);

    // launch GPU kernel
    kernel<<<blocks2, threads2>>>(common_change_input,frameno_in);
  }
  #pragma omp taskwait

  free(frame);
  fflush(NULL);
}
```

Listing 3.2: The final implementation of HeartWall

In HeartWall we created a task for the Initialization Kernel and a task with the main kernel that is called repeatedly. The main task uses the taskwait directive.

The Heart Wall application had to show some interesting facts about programming in the OmpSs model. First, in the OmpSs model we cannot use the constant memory utility that we have in CUDA. The constant/texture memories are not accepted by the compiler although work is being done in this direction. As a result, everything was changed to global memory.

```
structure unique; /* the structure we define in the host side*/
__constant__ structure d_unique; /* the structure in the Device side*/


cudaMalloc((void **)&unique.mask_conv, mask_conv_elem);
cudaMalloc((void **)&unique.tmask, tmask_elem);


cudaMemcpyToSymbol(d_unique,&unique,sizeof(structure));
```

Listing 3.3: CUDA code

```
#pragma omp target device
statically_allocated_memory[10000000];

#pragma omp target device
__global__ initialize_kernel(int *array1, int *array2, ...)
{
  int pointer_to_memory = 0;
  d_unique.mask_conv = statically_allocated_memory[pointer_to_memory];
  pointer_to_memory += mask_conv_elem

  d_unique.tmask = statically_allocated_memory[pointer_to_memory];
  pointer_to_memory += tmask_elem;

  .
  .
  .
}
```

Listing 3.4: OmpSs code

Secondly in the OmpSs model someone cannot allocate memory in the GPU manually. That means that the data to and from the GPU has to copied in and out. In this particular benchmark in the cuda code there were not any memory copies in and out the device. Instead the allocated memory was referenced through a device C structure (d unique). The pointer that originally pointed to the device memory (unique) was passed in the device with a cudaMemcpyToSymbol as seen in the example. The reference of memory through the d unique structure made the kernel much easier to code and read. In the OmpSs model, however, pointers cannot be passed in the device. We also did not want to change the large kernel which constists of about 1500 lines of CUDA code.

To solve these problems we allocated statically a big array of data. We made the structure in the GPU and made the attributes of the structure point in different parts of the big allocated array. The OmpSs version was only slightly slower than the CUDA version.

3.3 CFD

CFD solver is an unstructured finite volume solver for the 3D Euler equations for compressible flow. We used the single precision version with redundant flux computation scheme, which has reduced memory latency and high arithmetic intensity. We also made the evaluation with two datasets of 97K and 0.2M.

```
#pragma omp target device(cuda)
#pragma omp task firstprivate(nvar_var)
{
  dim3 grid_h1(1), thred_h1(nvar_var);
  device_mem_init<<<grid_h1,thred_h1>>>(argument_1,argument_2,argument_3,argument_4,argument_5
     ,argument1, argument2, argument3,              argument4,argument5,argument6,argument7,
     argument8,argument9,argument10,argument11,argument12);
}
#pragma omp taskwait noflush
  .
  .
for(int i = 0; i < iterations; i++)
{
  #pragma omp target device(cuda) copy_in(old_variables[0;elements_in],variables[0;elements_in
     ])
  #pragma omp task firstprivate
  {
    dim3 Dg(nelr * NVAR / 128), Db(128);
    copy_kernel<<<Dg,Db>>>(old_variables,variables);
  }
  #pragma omp taskwait noflush

  compute_step_factor(nelr, variables, areas, step_factors);


  for(int j = 0; j < RK; j++)
  {
    #pragma omp target device(cuda) copy_in(normals[0;normals_in],variables[0;elements_in],
     elements_surrounding_elements[0;elements_surr], fluxes[0;elements_in])
    #pragma omp task firstprivate(nelr,bl_len)
    {
      dim3 Dg(nelr / bl_len), Db(bl_len);
      cuda_compute_flux<<<Dg,Db>>>(nelr, elements_surrounding_elements, normals, variables,
     fluxes);
    }
    #pragma omp taskwait noflush

    #pragma omp target device(cuda) copy_in(old_variables[0;elements_in],fluxes[0;elements_in
     ],step_factors[0;nelr] , variables[0;elements_in])
    #pragma omp task firstprivate(nelr,bl_len)
    {
      dim3 Dg(nelr / bl_len), Db(bl_len);
      cuda_time_step<<<Dg,Db>>>(j, nelr, old_variables, variables, step_factors, fluxes);
    }
    #pragma omp taskwait noflush

  }

}
```

Listing 3.5: The final implementation of CFD

In CFD we created a task with the initialization of the device memory kernel, a task with kernel that copies the data from old variables to variables (which is done by CudaMemCpy from Device to Device in the), and two tasks for the main computation kernels. All use the taskwait directive with the noflush utility.

3.4 LUD

LUD is an algorithm to decompose a matrix as a product of a lower triangular matrix and an upper triangular matrix. The decomposition is done in parallel. For LUD we used two datasets. One with a matrix of 512 x 512. And the second with 2K x 2K elements.

```
 for (i=0; i < matrix_dim-BLOCK_SIZE; i += BLOCK_SIZE)
{
      #pragma omp target device(cuda) copy_in(m[0;elements_in])
      #pragma omp task firstprivate(block_size,matrix_dim,i)
  {
     dim3 grid1(1);
     dim3 thread1(block_size);
          lud_diagonal<<<grid1,thread1>>>(m, matrix_dim, i);
  }
  #pragma omp taskwait noflush


       #pragma omp target device(cuda) copy_in(m[0;elements_in])
       #pragma omp task firstprivate(block_size,matrix_dim,i)
  {
    dim3 grid2((matrix_dim-i)/block_size-1);
    dim3 thread2(block_size * 2);
              lud_perimeter<<<grid2,thread2>>>(m, matrix_dim, i);
  }
  #pragma omp taskwait noflush

  #pragma omp target device(cuda) copy_in(m[0;elements_in])
       #pragma omp task firstprivate(block_size,matrix_dim,i)
  {
         dim3 dimGrid((matrix_dim-i)/block_size-1, (matrix_dim-i)/block_size-1);
    dim3 dimBlock(block_size, block_size);
    lud_internal<<<dimGrid, dimBlock>>>(m, matrix_dim, i);
  }
  #pragma omp taskwait noflush
 }

       #pragma omp target device(cuda) copy_in(m[0;elements_in])
       #pragma omp task firstprivate(block_size,matrix_dim,i)
  {
    dim3 grid3(1);
    dim3 thread3(block_size);
          lud_diagonal<<<grid3,thread3>>>(m, matrix_dim, i);
  }
  #pragma omp taskwait noflush

  #pragma omp target device(cuda) copy_out(m[0;elements_in])
      #pragma omp task firstprivate(block_size,matrix_dim,i)
  {
    dim3 grid4(1);
    dim3 thread4(1);
    blank_kernel<<<grid4,thread4>>>(m);
  }
  #pragma omp taskwait
```

Listing 3.6: The final implementation of LUD

In LUD we created four tasks with the computation kernels, all with the taskwait directive and the noflush utility. A final task was created for the data to be sent to the host.

3.5 Hotspot

HotSpot is a 2D transient thermal modeling kernel which computes the final stage of a grid of cells when given the initial conditions (temperature and power dissipation per cell). The application iteratively updates the temperature values in all cells in parallel, and usually stops after a given number of iterations. For HotSpot we used two datasets of 512 x 512 (256KB) and 1024 x 1024 (1M) cells.

```
for (t = 0; t < total_iterations; t+=num_iterations) {
            int temp = src;
            src = dst;
            dst = temp;
      if (src == 0)
    {
    #pragma omp target device(cuda) copy_in(MatrixTemp0[0;size],MatrixTemp1[0;size],
     MatrixPower[0;size])
          #pragma omp task firstprivate(blockCols,blockRows)
    {
        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
             dim3 dimGrid(blockCols, blockRows);
                calculate_temp<<<dimGrid, dimBlock>>>(MIN(num_iterations, total_iterations-t),
      MatrixPower,MatrixTemp0,MatrixTemp1,\
       col,row,borderCols, borderRows, Cap,Rx,Ry,Rz,step,time_elapsed);
    }
    #pragma omp taskwait noflush
    }
       else
    {
     #pragma omp target device(cuda) copy_in(MatrixTemp0[0;size],MatrixTemp1[0;size],
     MatrixPower[0;size])
           #pragma omp task firstprivate(blockCols,blockRows)
    {
        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
             dim3 dimGrid(blockCols, blockRows);
                calculate_temp<<<dimGrid, dimBlock>>>(MIN(num_iterations, total_iterations-t),
      MatrixPower,MatrixTemp1,MatrixTemp0,\
       col,row,borderCols, borderRows, Cap,Rx,Ry,Rz,step,time_elapsed);
    }
      #pragma omp taskwait noflush

    }

  }
```

Listing 3.7: The final implementation of Hotspot

In Hotspot we used two tasks which call the two comutation kernels. Both with the taskwait directive and noflush utility.

3.6 Needleman-Wunsch

Needleman-Wunsch (NW)is a dynamic programming algorithm for sequence align-ment, which builds up the best alignment by using optimal alignments of smaller subsequences. It consists of three steps: initialization of the score matrix, calculation of scores, and deducing the alignment from the score matrix. The second step is parallelized.

```
for( int i = 1 ; i <= block_width ; i++){

  #pragma omp target device(cuda) copy_in(referrence[0;size],input_itemsets[0;size],
     matrix_cuda_out[0;size])
  #pragma omp task firstprivate(max_cols, penalty, i, block_width)
  {
    dim3 dimGrid;
    dim3 dimBlock(BLOCK_SIZE, 1);
    dimGrid.x = i;
    dimGrid.y = 1;
    needle_cuda_shared_1<<<dimGrid, dimBlock>>>(referrence, input_itemsets, matrix_cuda_out
                                          ,max_cols, penalty, i, block_width);
  }
  #pragma omp taskwait noflush

}
      //process bottom-right matrix
for( int i = block_width - 1  ; i >= 1 ; i--){

  #pragma omp target device(cuda) copy_in(referrence[0;size],input_itemsets[0;size],
     matrix_cuda_out[0;size])
  #pragma omp task firstprivate(max_cols, penalty, i, block_width)
  {
    dim3 dimGrid1;
    dim3 dimBlock1(BLOCK_SIZE, 1);
    dimGrid1.x = i;
    dimGrid1.y = 1;
    needle_cuda_shared_2<<<dimGrid1, dimBlock1>>>(referrence, input_itemsets, matrix_cuda_out
                                          ,max_cols, penalty, i, block_width);
  }
  #pragma omp taskwait noflush
  }
```

Listing 3.8: The final implementation of Needleman-Wunsch

In Needleman-Wunsch we created two tasks which invoke the two main computation kernels. Both the taskwait noflush directive.

3.7 BFS

Bfs is a fundamental graph search algorithm. We are testing its parallel implementation on two graph datasets consisting of 64K and 1M nodes.

```
do
{
  *d_over = false;
  #pragma omp target device(cuda) copy_in(d_over[0;1])
  #pragma omp task
  {
    dim3 grid1(1);
    dim3 threads1(1);
    input_kernel1<<<grid1,threads1>>>( d_over);
  }
  #pragma omp taskwait noflush


  #pragma omp target device(cuda) copy_inout(h_graph_nodes[0;no_of_nodes],h_graph_edges[0;
     edge_list_size],h_graph_mask[0;no_of_nodes], h_updating_graph_mask[0;no_of_nodes],
     h_graph_visited[0;no_of_nodes],h_cost[0;no_of_nodes]) //copy_out(h_graph_mask[0;
     no_of_nodes],h_cost[0;no_of_nodes],h_updating_graph_mask[0;no_of_nodes])
  #pragma omp task firstprivate(no_of_nodes,num_of_blocks,num_of_threads_per_block)
  {
      dim3  grid( num_of_blocks, 1, 1);
      dim3  threads( num_of_threads_per_block, 1, 1);
      Kernel<<< grid, threads,0 >>>( h_graph_nodes, h_graph_edges, h_graph_mask,
     h_updating_graph_mask, h_graph_visited, h_cost, no_of_nodes);
  }
  #pragma omp taskwait noflush

  #pragma omp target device(cuda) copy_inout(h_graph_mask[0;no_of_nodes],
     h_updating_graph_mask[0;no_of_nodes],h_graph_visited[0;no_of_nodes],d_over[0;1])
  #pragma omp task firstprivate(no_of_nodes,num_of_blocks,num_of_threads_per_block)
  {
    dim3  grid_2( num_of_blocks, 1, 1);
    dim3  threads_2( num_of_threads_per_block, 1, 1);
    Kernel2<<< grid_2, threads_2,0 >>>( h_graph_mask, h_updating_graph_mask, h_graph_visited,
     d_over, no_of_nodes);
  }
  #pragma omp taskwait noflush

  #pragma omp target device(smp) copy_inout(d_over[0;1])
  #pragma omp task shared(stop)
  {
    stop = *d_over;
  }
  #pragma omp taskwait noflush

  }
  while( stop );
```

Listing 3.9: The final implementation of BFS

In BFS we created one task for the data to be sent to the device (Input Kernel), two tasks with the main computation kernels, which both have the taskwait noflush directive, and finally, one task with the device(smp) directive which is used for the dover variable to be sent from the device to the host without flushing the memory.

3.7.1 Optimization

The first OmpSs version of the BFS was significantly slower. The 64K data set run at 25 mSecs (almost 6 times slower) and the 1 MB dataset at 294 mSecs (12.5 times slower). That's why as presented in the code below a stop variable had to be extracted in every repetition to see if any change has been made and if the whole of the graph has been searched. In the first version we made an output kernel with a taskwait directive in order to have the stop variable extracted. This version had significant overhead because the taskwait directive made the memory flush. As a result in each repetition the data had to be copied in again and again with the copy in directive.

```
#pragma omp target device(cuda) copy_inout(d_stop[0;1])
#pragma omp task
{
  output_kernel<<<grid,threads>>>(d_stop)

}
#pragma omp taskwait
```

Listing 3.10: Slow version

The solution we proposed was a pragma directive call with the device(smp) attribute which allowed us to extract the stop variable in the host without having the GPU memory flushed.

```
#pragma omp target device(smp) copy_inout(d_stop[0;1])
#pragma omp task shared(stop)
{
  stop = *d_stop;
}
#pragma omp taskwait noflush
```

Listing 3.11: Fast version

3.8 Kmeans

K-means (KM) is a clustering algorithm that uses the mean based data partitioning method. It contains dense linear algebra calculations and it has a lot of data parallelism to be exploited.

```
do {
        delta = 0.0;
  delta = (float) kmeansCuda(feature,nfeatures,npoints,nclusters,membership,clusters,
    new_centers_len,new_centers,c);
  for (i=0; i<nclusters; i++) {
    for (j=0; j<nfeatures; j++) {
```

```
      if (new_centers_len[i] > 0)
        nlusters[i][j] = new_centers[i][j] / new_centers_len[i];  /* take average i.e. sum/n
    */
        new_centers[i][j] = 0.0;  /* set back to 0 */
      }
      new_centers_len[i] = 0;      /* set back to 0 */
    }
    c++;
    } while ((delta > threshold) && (loop++ < 500));


float kmeansCuda ()
{
  #pragma omp target device(cuda) copy_inout(membership_new[0;npoints],t_features[0;
     elements_in2],t_features_flipped[0;elements_in2],feature_d   [0;elements_in2],
     feature_flipped_d[0;elements_in2])
  #pragma omp task firstprivate(num_blocks,num_threads,npoints,nfeatures)
{

   dim3 grid3(num_of_blocks,1);
  dim3 block3(num_of_threads,1);
  copy_kernel_nfeatures<<<grid3,block3>>>(membership_new,t_features,t_features_flipped,
     feature_d,feature_flipped_d);

}
#pragma omp taskwait noflush

  #pragma omp target device(cuda) copy_in(feature_d[0;elements_in2],membership_new[0;npoints],
     clusters_d[0;elements_in1],t_features[0;elements_in2],t_features_flipped[0;elements_in2],
     t_clusters[0;elements_in1]) copy_out(membership_new[0;npoints])
  #pragma omp task firstprivate(num_blocks_perdim,num_threads_perdim,nfeatures,npoints,
     nclusters)
  {
     dim3  grid( num_blocks_perdim, num_blocks_perdim );
       dim3  threads( num_threads_perdim*num_threads_perdim );

  /* execute the kernel */
     kmeansPoint<<< grid, threads >>>( feature_d,nfeatures,npoints, nclusters,membership_new,
     clusters_d, t_features,t_features_flipped,t_clusters);

  }
#pragma omp taskwait


}
```

Listing 3.12: The final implementation of Kmeans

In Kmeans we created on task for the data to be copied in the device and one task which invlokes the main computation kernel and used the taskwait directive. The memory flushes after the call of the kernel.

3.9 SRAD

SRAD (Speckle Reducing Anisotropic Diffusion) is a diffusion method used in ultrasonic and radar imaging applications. SRAD is iterative; in each iteration, computing and updating of the whole image are performed in parallel.

```
for (iter=0; iter<niter; iter++){

#pragma omp target device(cuda) copy_in(image[0;Ne]) copy_out(d_sums[0;Ne],d_sums2[0;Ne])
    #pragma omp task firstprivate(Ne)
    {
      dim3 blocks21;
      dime3 threads21.
      prepare<<<blocks21, threads21>>>(Ne,image,d_sums,d_sums2);

    }
    #pragma omp taskwait noflush


  while(blocks2x != 0)
  {
    block_num_x1 = blocks2x;

    #pragma omp target device(cuda) copy_out(d_sums[0;Ne],d_sums2[0;Ne])
    #pragma omp task firstprivate(Ne,no,mul,block_num_x1)
    {

      reduce<<<blocks22, threads22>>>(Ne,no,mul,d_sums, d_sums2);

    }
    #pragma omp taskwait noflush

    #pragma omp target device(smp) copy_in(d_sums[0;Ne],d_sums2[0;Ne])
    #pragma omp task shared(total,total2)
    {
      total = d_sums[0];
      total2 = d_sums2[0];
    }

  #pragma omp target device(cuda) copy_in(iN[0;Nr],iS[0;Nr],jE[0;Nc],jW[0;Nc],image[0;Ne],d_dN
     [0;Ne],d_dS[0;Ne],d_dW[0;Ne],d_dE[0;Ne],d_c[0;Ne]) \
        copy_out(d_dN[0;Ne],d_dS[0;Ne],d_dW[0;Ne],d_dE[0;Ne],d_c[0;Ne])
  #pragma omp task firstprivate(Ne,Nc,Nr,q0sqr,lambda,blocks_x)
  {
    dim3 blocks23;
    dim3 threads23;
    srad<<<blocks23, threads23>>>( lambda,      Nr,Nc,Ne,iN,iS,jE,jW,d_dN,d_dS,d_dW,d_dE,d_c,
     image);

  }
  #pragma omp taskwait noflush

  #pragma omp target device(cuda) copy_inout(iN[0;Nr],iS[0;Nr],jE[0;Nc],jW[0;Nc],d_dN[0;Ne],
     d_dS[0;Ne],d_dW[0;Ne],d_dE[0;Ne],d_c[0;Ne],image[0;Ne])
        //copy_out(image[0;Ne])
  #pragma omp task firstprivate(Ne,Nc,Nr,lambda,blocks_x)
  {
    dim3 blocks24;
    dim3 threads24;
    srad2<<<blocks24, threads24>>>( lambda, Nr, Nc, Ne, iN, iS, jE, jW, d_dN, d_dS, d_dW, d_dE
     , d-c,  image);
```

```
  }
  #pragma omp taskwait noflush
}

}
```

LISTING 3.13: The final implementation of SRAD

In SRAD we created five tasks. Four of them call the kernels and use the directive taskwait with noflush and one is implemented for the host and allows the transfers of the results of the reduction kernel (sums[0] and sums2[0]) from the device to the host without the memory being flushed.

3.9.1 Optimization

The first OmpSs vesrion that we constructed was quite slow. It gave a speedup of about 2 compared to the serial version. That was because a taskwait directive after a reduce CUDA operation made the memory to flush and involved extra copy ins and outs. We avoided the memory flush with a pragma smp directive as shown in the code below. As

Program 7 Slow version

```
#pragma omp target device(cuda) copy_out(d_sums[0;1],d_sums2[0;1])
#pragma omp task
{
  reduce<<<blocks,threads(d_sums,d_sums2)>>>
}
#pragma omp taskwait
total = d_sums[0];
total2 = d_sums2[0];
```

the only thing we wish as a memory output is only the first element of the array we did it as shown below, to avoid the memory flush:

Program 8 Fast version

```
#pragma omp target device(cuda) copy_out(d_sums[0;El],d_sums2[0;El])
#pragma omp task
{
  reduce<<<blocks,threads(d_sums,d_sums2)>>>
}
#pragma omp taskwait noflush

#pragma omp target device(smp) copy_in(d_sums[0;El],d_sums2[0;El])
#pragma omp task  shared(total,total2)
{
  total = d_sums[0];
  total2 = d_sums2[0];
}
#pragma omp taskwait noflush
```

3.10 Particle Filter

Particle Filter (PF) is a probabilistic model for tracking objects in a noisy environment using a given set of particle samples . The application has several parallel stages, and implicit synchronization between stages is required.

```
.
.
.
#pragma omp target device(cuda) copy_in(arrayX[0;Nparticles],arrayY[0;Nparticles],CDF[0;
    Nparticles],u[0;Nparticles],xj[0;Nparticles],yj[0;Nparticles]) copy_out(xj[0;Nparticles],
    yj[0;Nparticles])
#pragma omp task firstprivate(num_blocks, threads_per_block, Nparticles)
{
  kernel <<< num_blocks, threads_per_block >>> (arrayX, arrayY, CDF, u, xj, yj, Nparticles);
}
#pragma omp taskwait
.
.
.
```

Listing 3.14: The final implementation of Particle FIlter

In Particle Filter we created a single task which calls the computation kernel. It uses the taskwait directive. The memory flushes afte the invocation.

3.11 PathFinder

PathFinder is a dynamic programming algorithm to finds the shortest path of a 2D grid, row by row, by choosing the smallest accumulated weights. In each iteration, the shortest path calculation is parallelized.

```
for (int t = 0; t < rows-1; t+=pyramid_height) {
            temp = src;
            src = dst;
            dst = temp;


  if (src == 0)
  {

  #pragma omp target device(cuda) copy_in(gpu1[0;cols],gpuWall[0;wall_size]) copy_out(gpu2[0;
    cols])
  #pragma omp task firstprivate(cols,rows,borderCols,pyramid_height,t,block_size,blockCols,min
    )
  {

    dim3 dimBlock(block_size);
   dim3 dimGrid(blockCols);
          dynproc_kernel<<<dimGrid, dimBlock>>>(min, gpuWall, gpu1, gpu2, cols,rows, t,
    borderCols);
  }
```

```
  #pragma omp taskwait noflush
  }
  else
  {
    #pragma omp target device(cuda) copy_in(gpu2[0;cols],gpuWall[0;wall_size]) copy_out(gpu1
     [0;cols])
    #pragma omp task firstprivate(cols,rows,borderCols,pyramid_height,t,block_size,blockCols,
     min)
    {

      dim3 dimBlock1(block_size);
      dim3 dimGrid1(blockCols);
              dynproc_kernel<<<dimGrid1, dimBlock1>>>(min,gpuWall, gpu2, gpu1,cols,rows, t,
     borderCols);
  }
  #pragma omp taskwait noflush

  }
}
```

Listing 3.15: The final implementation of PathFinder

In PathFinder we created two tasks which invoke the computation kernels. The taskwait directive was used with noflush for the memory.

3.12 Gaussian Elimination

In linear algebra, Gaussian elimination is an algorithm for solving systems of linear equations. It can also be used to find the rank of a matrix, to calculate the determinant of a matrix, and to calculate the inverse of an invertible square matrix. Elementary row operations are used to reduce a matrix to what is called triangular form. Gaussian Elimination computes result row by row, solving for all of the variables in a linear system. The algorithm must synchronize between iterations, but the values calculated in each iteration can be computed in parallel.

```
for (t=0; t<(Size-1); t++) {

  #pragma omp target device(cuda) copy_in(m[0;Size*Size],a[0;Size*Size])
  #pragma omp task firstprivate(Size,t,block_size,grid_size)
  {
  dim3 dimBlock(block_size);
  dim3 dimGrid(grid_size);
  Fan1<<<dimGrid,dimBlock>>>(m,a,Size,t);
  }
  #pragma omp taskwait noflush

  #pragma omp target device(cuda) copy_in(m[0;Size*Size],a[0;Size*Size],b[0;Size])
  #pragma omp task firstprivate(blockSize2d,gridSize2d,Size,t)
  {
    dim3 dimBlockXY(blockSize2d,blockSize2d);
    dim3 dimGridXY(gridSize2d,gridSize2d);
    Fan2<<<dimGridXY,dimBlockXY>>>(m,a,b,Size,Size-t,t);
```

```
  }
  #pragma omp taskwait noflush

  }
```

Listing 3.16: The final implementation of Gaussian Elimination

In Gaussian Elimination we created two tasks which invoke the computation kernels. The taskwait directive was used with the noflush utility.

3.13 Nearest Neighbors

NN (Nearest Neighbor) finds the k-nearest neighbors from an unstructured data set. The sequential NN algorithm reads in one record at a time, calculates the Euclidean distance from the target latitude and longitude, and evaluates the k nearest neighbors. The parallel versions read in many records at a time, execute the distance calculation on multiple threads, and the master thread updates the list of nearest neighbors.

```
#pragma omp target device(cuda) copy_in(sandbox[0;mem_in]) copy_out(z[0;mem_out])
#pragma omp task firstprivate(x2,y2,block_size)
{

  dim3 dimBlock(block_size);
  dim3 dimGrid( (REC_WINDOW/dimBlock.x) + (!(REC_WINDOWdimBlock.x)?0:1) );
  euclid<<<dimGrid,dimBlock>>>(sandbox, x2, y2, z, REC_WINDOW, REC_LENGTH, LATITUDE_POS);

}
#pragma omp taskwait
```

Listing 3.17: The final implementation of Nearest Neighbors

In Nearest Neighbors we created on task with the invocation of the computation kernel with the taskwait directive. The memory flushes after the call.

3.14 Streamcluster

For a stream of input points, Streacluster a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances (SSQ) metric.

```
#pragma omp target device(cuda) copy_deps
   #pragma omp task firstprivate (num_blocks_x,num_blocks_y,dim,num,x,K) inout(work_mem_h[0;
     work_mem_in],switch_membership[0;num],center_table[0;num],coord_h[0;coord_in],point_p[0;
     num])
```

```
  {
  dim3 grid_size(num_blocks_x, num_blocks_y,1);
  dim3 threads(THREADS_PER_BLOCK);
//   size_t smSize = dim * sizeof(float);
  pgain_kernel<<< grid_size, threads>>> (
     num,dim,x,point_p,K,coord_h,work_mem_h,center_table,switch_membership);
}
#pragma omp taskwait

// num: dimension of point coordinates
// dim: point to open a center at
// x: data point array
// point_p: number of centers
// coord_h: array of point coordinates
// work_mem_h:  cost and lower field array
// center_table:  center index table
// switch membership:  changes in membership
```

Listing 3.18: The final implementation of StreamCluster

The OmpSs performance of Streamcluster was problematic. We explain why in the next chapter.

Chapter 4

Performance Analysis of Rodinia Benchmark

4.1 BSC

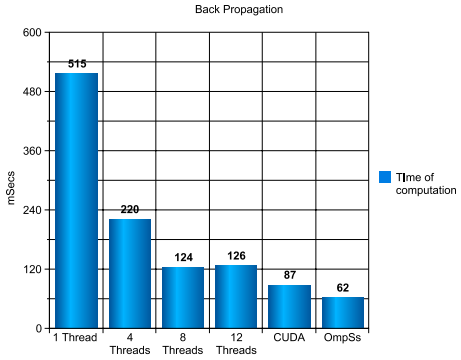
The experiments were done in MinoTauro, the cluster in the Barcelona Supercomputing Centre. We used one compute node which consists of two 6-Core Intel chips, E5649 at 2.53 GHz and two GPUs of NVIDIA Tesla M2090 of the Fermi architecure. The Intel chips have a max memory size of 288GB (with 12MB Cache). The GPU has 512 CUDA cores with 665 GFlops peak double preicision floating point performance. Its global memory size is 6 GB. MinoTauro has made it to both the Top500 and Green500 lists in recent years.

4.2 Back Propagation[1]

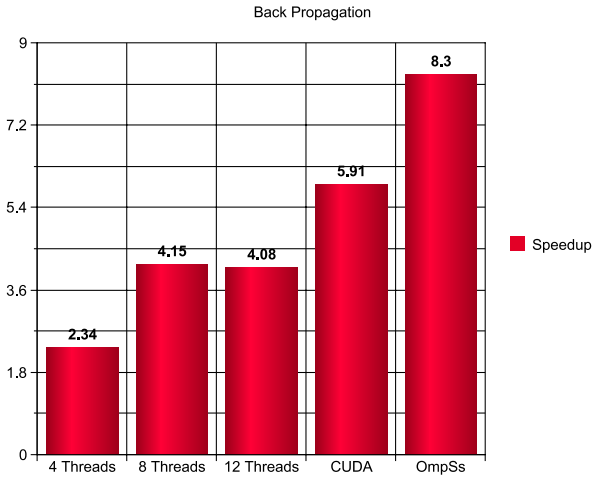
For Back Propagation we used two datasets of 1 MB and 16 MB nodes.

GPU TRANSFER STATISTICS - 1MB	
Total input transfers	208MB
Total output transfers	4.125MB

[1]In the GPU statistics that we show the Device inputs and outputs measure in fact the Copy In and Copy Outs that take place in the tasks. In benchmarks that we used the noflush directive those do not necessarily coincide with the actual inputs and outputs. However we include them for reasons of estimate of the memory transfers that take place in each benchmark.



(a) Time of computation

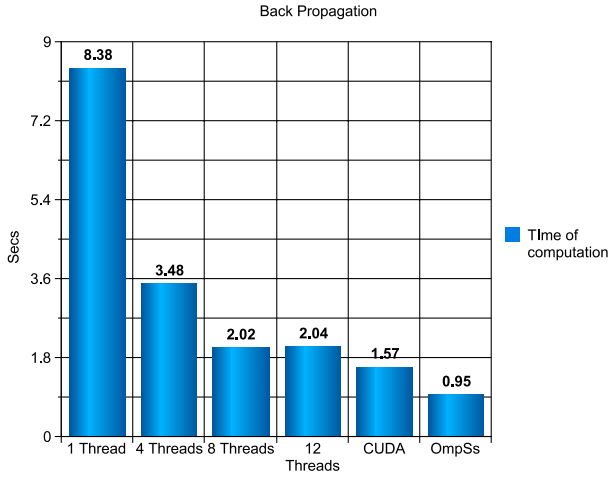


(b) Speedup

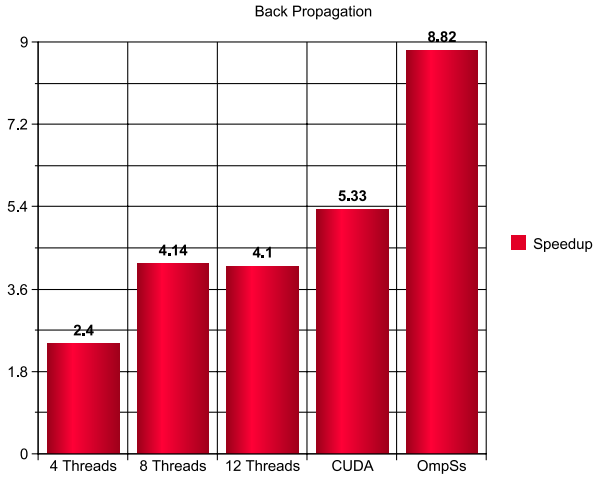
FIGURE 4.1: Size 1 MB

The time of the computation is the time needed to compute the bpnn train function which is the heart of the computation. As mentioned above it involves two main phases which in the CUDA and OmpSs versions correspond to two CUDA kernel calls. The OpenMP version does not scale well in this application and we have a peak performace with 8 threads and a speedup of about 4. As we can see from the charts above in this benchmark the OmpSs version is the fastest with a speedup of 8. Comparing with the CUDA version they can be assumed similar. We note that the OpenCL version produced a run time error during the execution of the kernel and was not evaluated.

GPU TRANSFER STATISTICS 16MB	
Total input transfers	3.25GB
Total output transfers	66MB



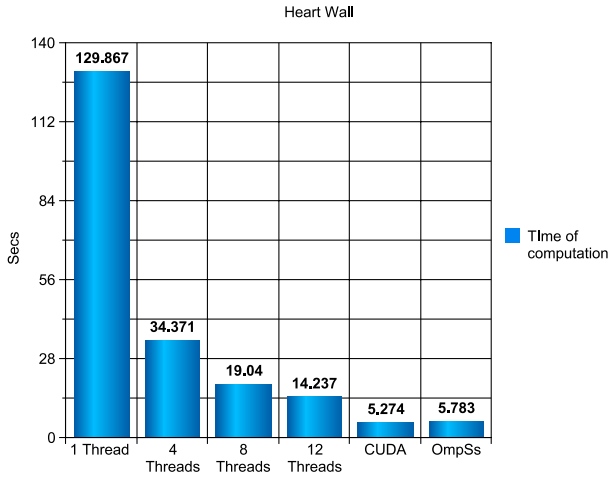
(a) Time of computation



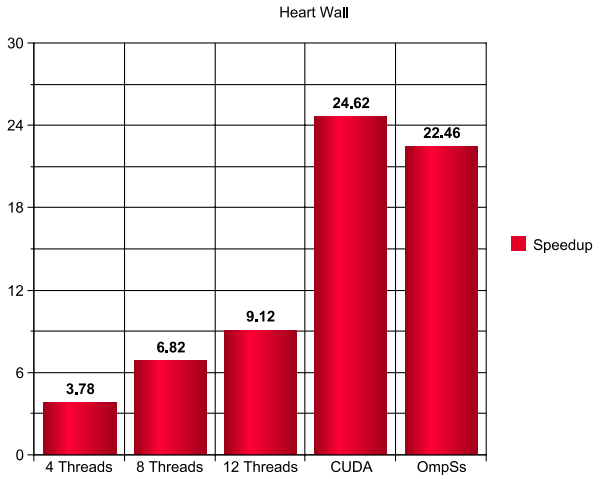
(b) Speedup

FIGURE 4.2: Size 16 MB

4.3 Heart Wall



(a) Time of computation



(b) Speedup

FIGURE 4.3: Heart Wall

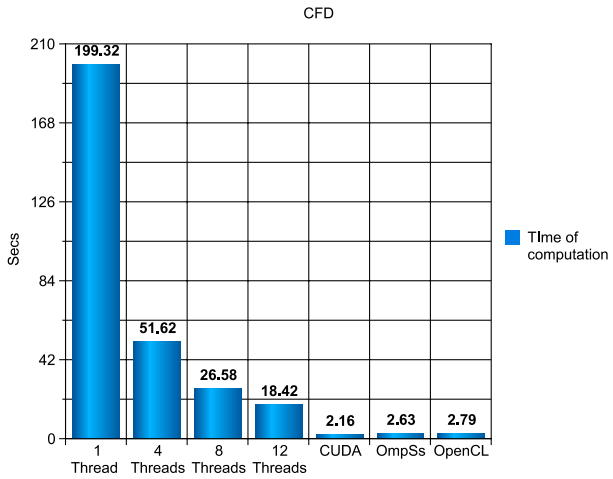
GPU TRANSFER STATISTICS	
Total input transfers	192.63MB
Total output transfers	62.15KB

Only two stages of the application, SRAD and Tracking, have enough parallelism and significant contribution to the overall run time to justify optimization efforts. SRAD is also a part of the Rodinia suite and is examined afterwards separately. The separation of these two parts of the application is done for reasons of clarity of the analysis. In this application we analyse the results of the Heart Wall tracking parallelization. However , in the near future, the applications are planed to be unified.

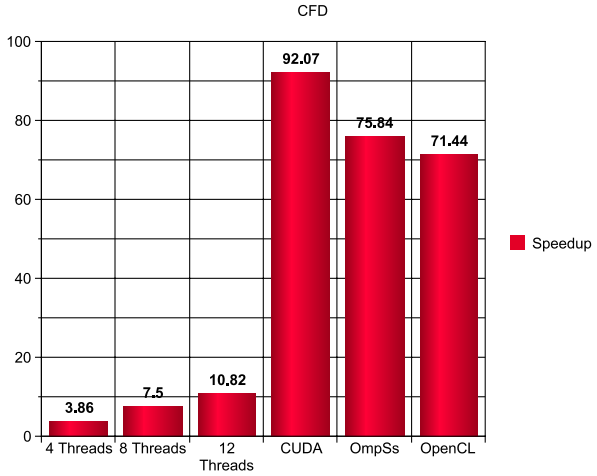
The OpenMP version of Heart Wall shows good scaling until 8 threads. Then, there is a decline. In this application we can actually notice the huge performance advantage we can get with GPU's. The CUDA version has a speedup of 24.62 compared to the serial version. In HeartWall OmpSs performs very similar to CUDA. The inputs and outputs and the heart of the computation is the same in the two occasions. In fact the OmpSs kernel is slightly changed to be adapted the fact that we cannot pass pointers in the OmpSs kernels as input. The computation itself is the same. Further explanation can be seen in the HeartWall section of the previous chapter.

4.4 CFD

We made the evaluation with two datasets of 97K and 0.2M.



(a) Time of computation



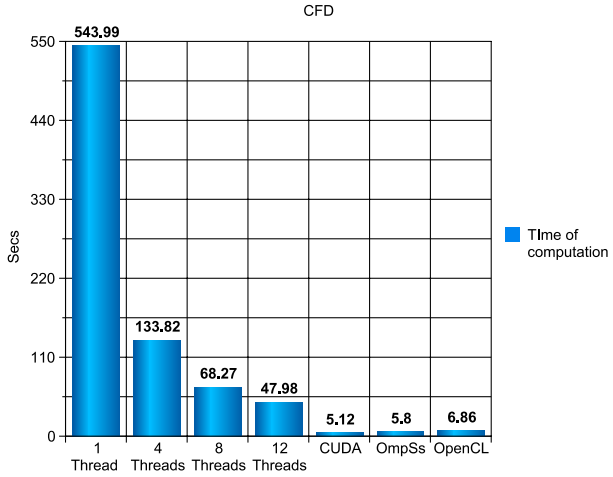
(b) Speedup

FIGURE 4.4: CFD - 97K

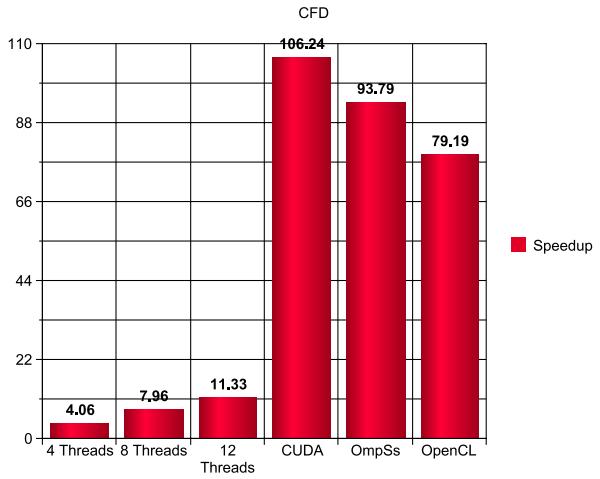
GPU TRANSFER STATISTICS - 97K	
Total input transfers	11.85MB
Total output transfers	2.22MB

The OpenMP version of CFD scales until the 12 - thread execution. In [3] you can see its behaviour with more cores available. The OmpSs version shows similar performance with the CUDA version. In the OmpSs version we had to construct a new initialization kernel which makes all the nessecary transactions of data between the host and the GPU. The CUDA kernels were called with a noflush directive as we did not want any transactions

between the calls. We made also a new copy kernel to copy the data between two arrays in the GPU as there is no something similar as the CudaMemCpyDeviceToDevice. We inevitably had also some extra host initializations due to to the fact that we couldn't use a CudaMalloc. As a result we can have a memory GPU initialization only if we have a corresponding host memory initialization as mentioned above in the heartwall example.



(a) Time of computation



(b) Speedup

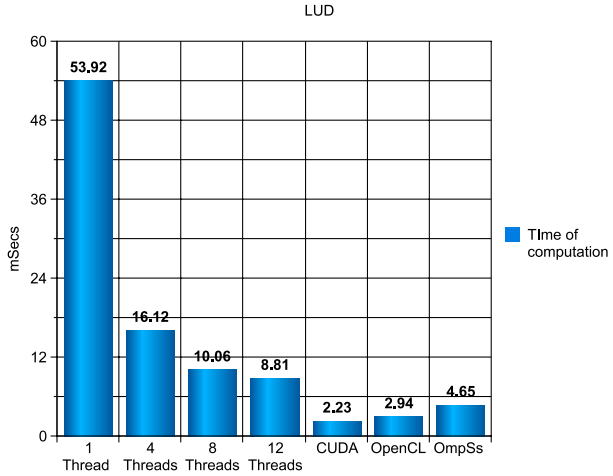
FIGURE 4.5: CFD - 0.2M

GPU TRANSFER STATISTICS - 0.2MB	
Total input transfers	28.4MB
Total output transfers	5.32MB

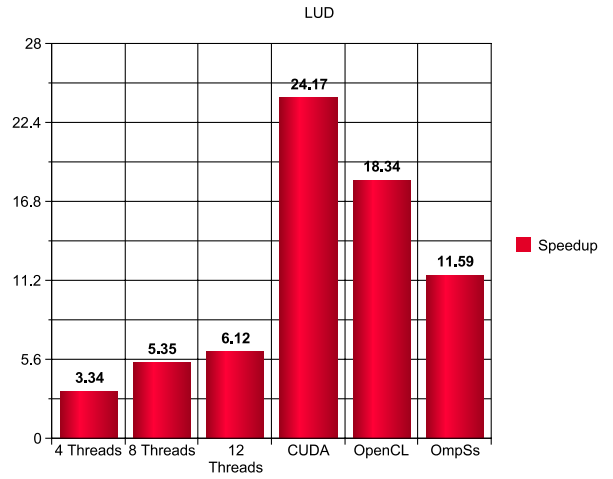
CFD is another good example of how the GPU processing power can sometimes lead to massive performance gains.

4.5 LU Decomposition

For LUD we used three datasets. One with a matrix of 256 x 256.The second with 512 x 512 And the third with 2K x 2K elements.



(a) Time of computation

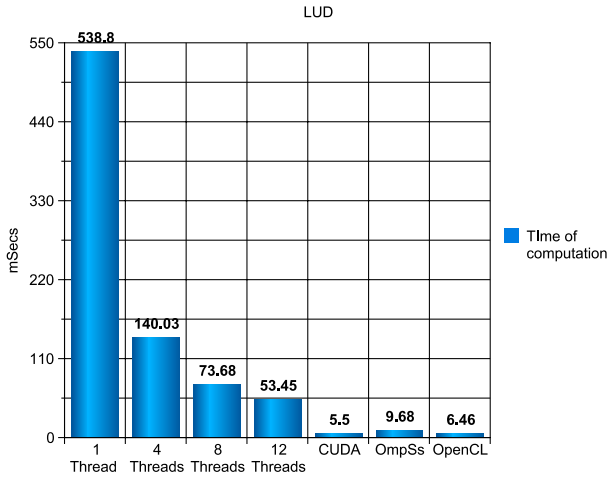


(b) Speedup

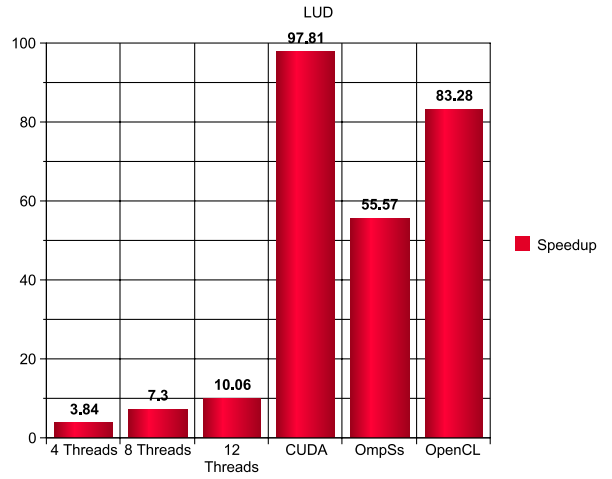
FIGURE 4.6: LUD - 256 X 256

GPU TRANSFER STATISTICS 256 x 256	
Total input transfers	256KB
Total output transfers	256KB

GPU TRANSFER STATISTICS - 512 x 512	
Total input transfers	1MB
Total output transfers	1MB



(a) Time of computation

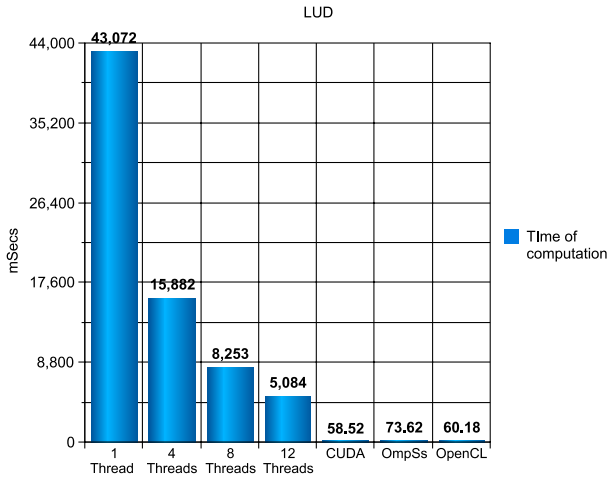


(b) Speedup

FIGURE 4.7: LUD - 512 x 512

GPU TRANSFER STATISTICS 2K x 2K	
Total input transfers	16MB
Total output transfers	16MB

The OmpSs version is slightly slower than the CUDA and OpenCL versions. The computation involves repeated kernel calls in a loop and they are constructed with the taskwait noflush directive. The OpenMP shows scaling till the 12 - thread execution. We can see from the third test case that for bigger matrices the OpenMP version is quite slow and the GPU versions offer a huge advantage in computation.

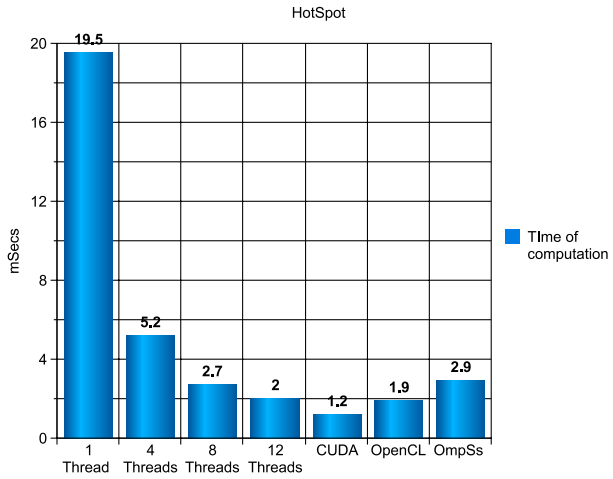


(a) Time of computation

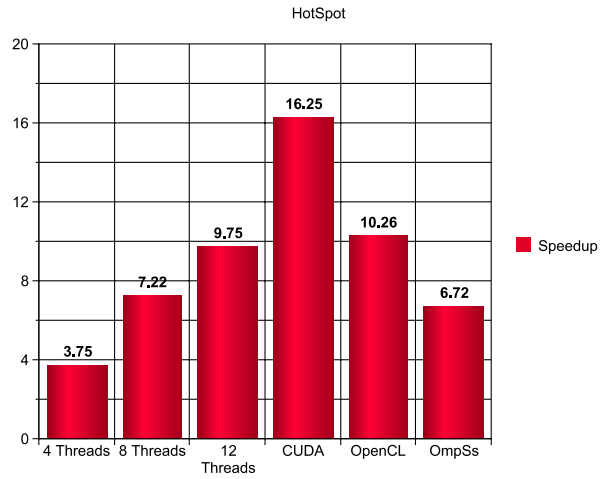
FIGURE 4.8: LUD - 2K x 2K

4.6 HotSpot

For HotSpot we used two datasets of 512 x 512 (256KB) and 1024 x 1024 (1M) cells.



(a) Time of computation



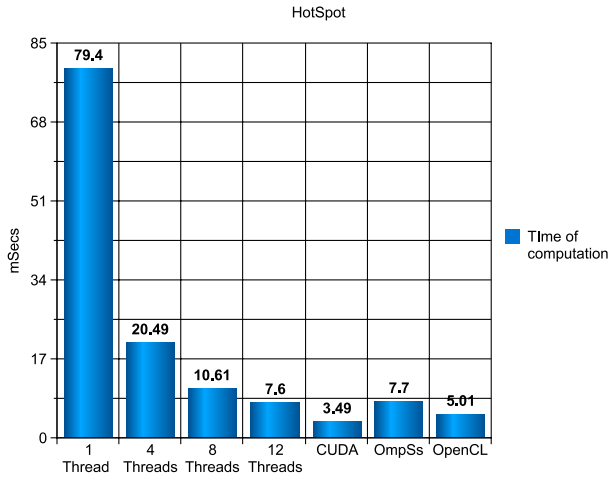
(b) Speedup

FIGURE 4.9: HotSpot 512 x 512

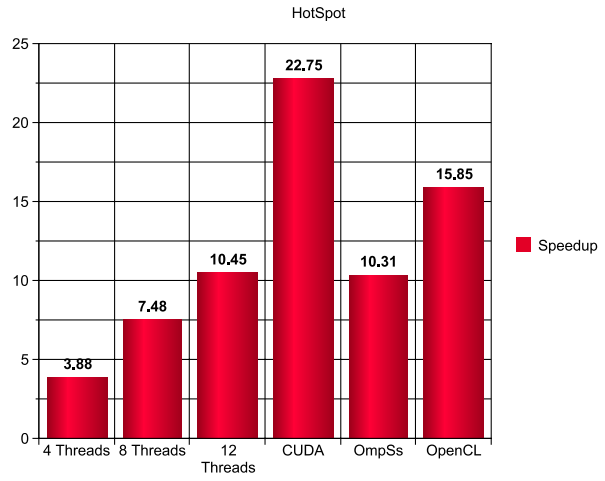
GPU TRANSFER STATISTICS 512 x 512	
Total input transfers	3MB
Total output transfers	1MB

The OpenMP version scales until 12 threads. The OmpSs version is 2 times slower than the CUDA version. The OpenCL performace is between the previous two. In the OmpSs version we make repeated calls in the CUDA kernel with the noflush directive and we constructed one extra output kernel for data to be transfered to the Host from the Device. The output kernel execution is partly responsible for the small delay of

the OmpSs version although the execution time is too small (a few ms) to allow safe deductions.



(a) Time of computation



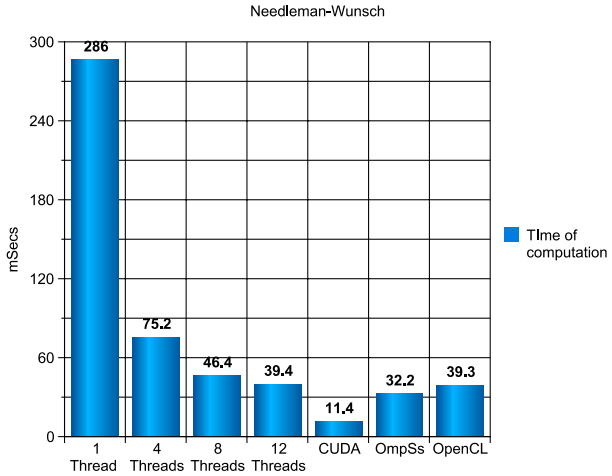
(b) Speedup

FIGURE 4.10: HotSpot 1024 x 1024

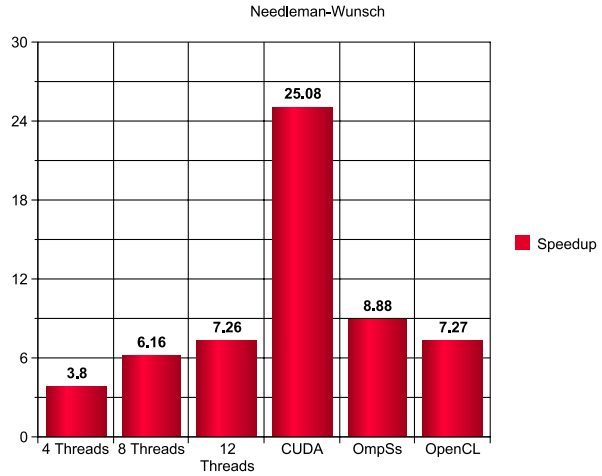
GPU TRANSFER STATISTICS 1024 x 1024	
Total input transfers	12MB
Total output transfers	4MB

4.7 Needleman-Wunsch

For Needleman-Wunsch we used two 2D matrix datasets of 2K x 2K and 4K x 4K.



(a) Time of computation

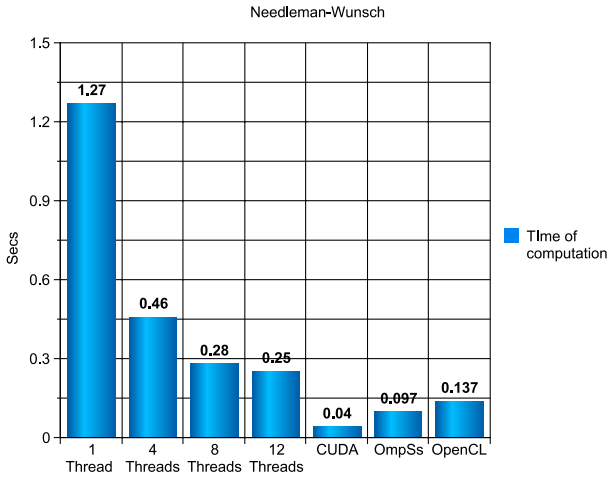


(b) Speedup

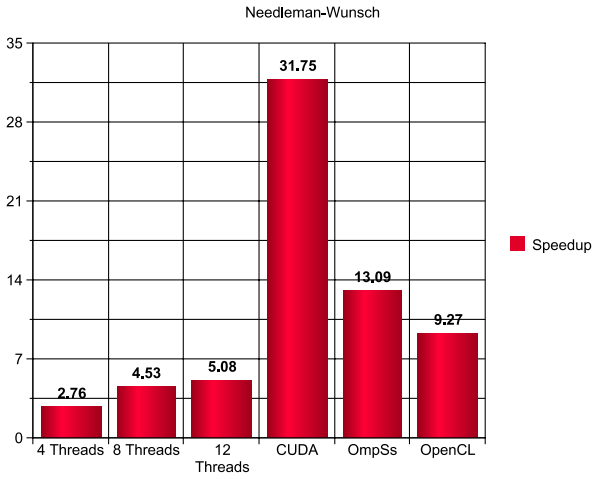
FIGURE 4.11: NW - 2K x 2K

GPU TRANSFER STATISTICS 2048 x 2048	
Total input transfers	48.04MB
Total output transfers	16.01MB

The OpenMP version has a relatively poor performance in the NW algorithm. According to[3] the performance gained by parallelization is diminished by the overhead of frequent memory accesses and memory latency. The OmpSs version is slower than the CUDA version and closer to performance to the OpenCL one. We also have here repeated calls of the CUDA kernel in a for loop and we used the noflush directive.



(a) Time of computation



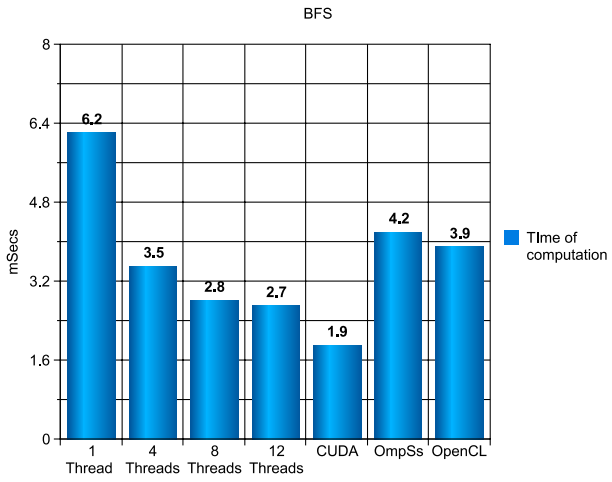
(b) Speedup

FIGURE 4.12: NW -4K x 4K

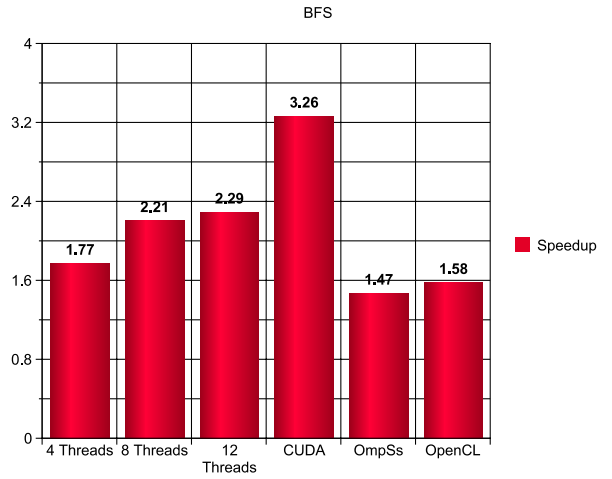
GPU TRANSFER STATISTICS 4096 x 4096	
Total input transfers	192.09MB
Total output transfers	64.03MB

4.8 BFS

We are testing the parallel implementation of BFS with two graph datasets consisting of 64K and 1M nodes.



(a) Time of computation

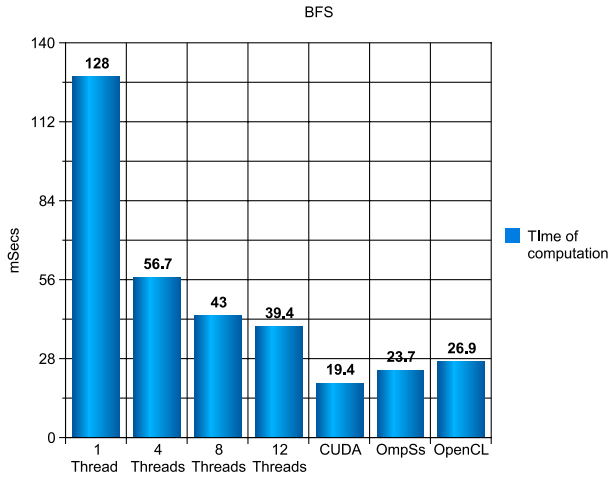


(b) Speedup

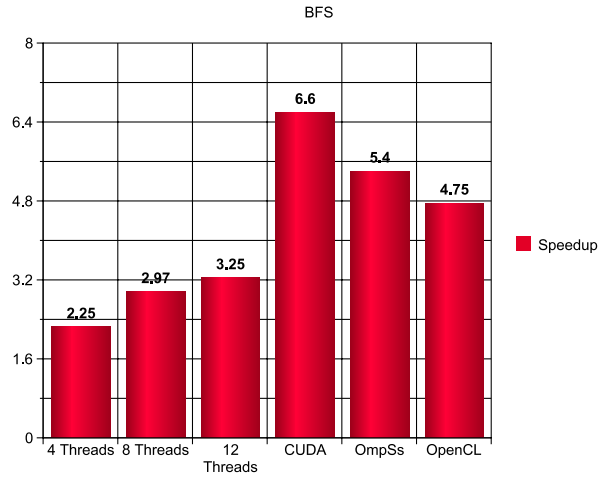
FIGURE 4.13: BFS - 64K

GPU TRANSFER STATISTICS 64K	
Total input transfers	26.81MB
Total output transfers	26.81MB

The OpenMP version of the BFS algorithms has poor performance. It has about a speedup of about 3 for 12 threads. According to after the OpenMP version shows its peak performance with 8 or 12 threads and then it remains stable.



(a) Time of computation



(b) Speedup

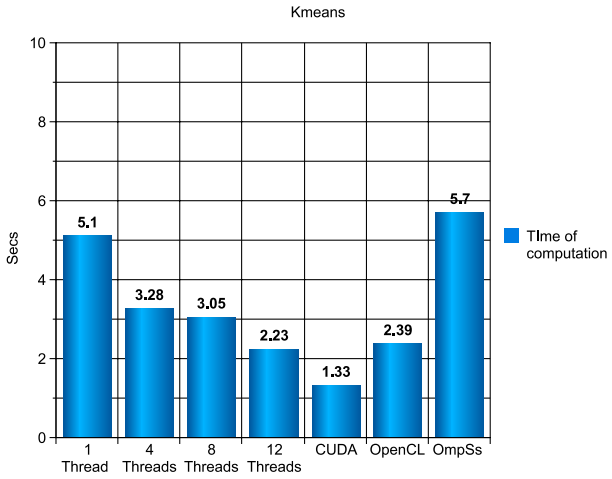
FIGURE 4.14: BFS - 1M

GPU TRANSFER STATISTICS 1M	
Total input transfers	483.5MB
Total output transfers	483.5MB

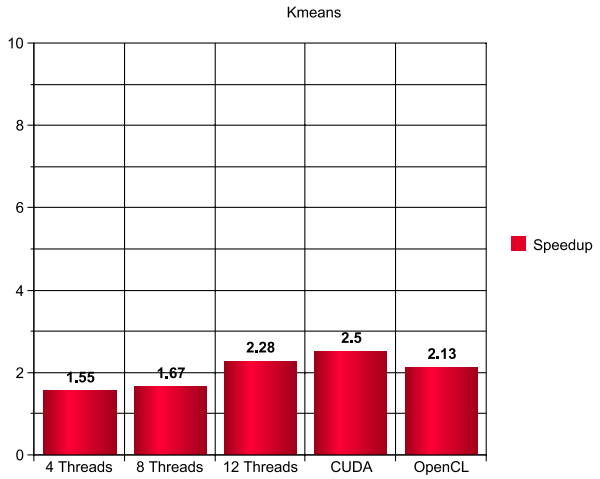
The OmpSs shows similar performance to the CUDA version (slightly worse) and is slightly faster than the OpenCL version.

4.9 K-means

For K - means we used two datasets of 482K, 800K objects respectively.



(a) Time of computation



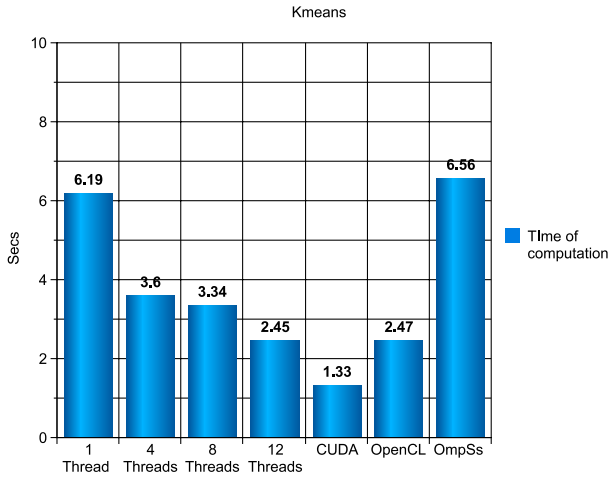
(b) Speedup

FIGURE 4.15: Kmeans - 482K

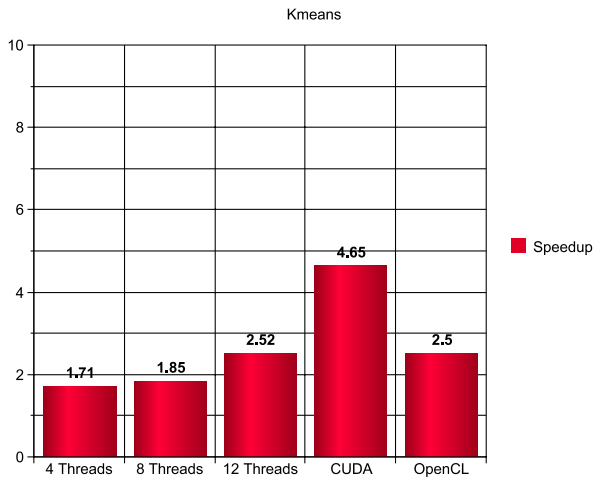
GPU TRANSFER STATISTICS 482K	
Total input transfers	1.32GB
Total output transfers	1.32GB

The K-means OpenMP version scales but not quite good though. It achieves a maximum of 2.5 speedup at 12 threads. According to [3] it achieves a maximum performance at 24 threads and then it remains almost steady due to cache misses and thread switch overhead. The Kmeans is one of the benchmarks that OmpSs gives a poor performance. We assume that's partly because of the loss of the optimizations of the special memories of CUDA. In the CUDA version we had the constant memory utility which gave significant performance gains. In the OmpSs the compiler does not accept yet Constant

memories or textures, although in new versions of OmpSs these might be included. The OmpSs version does not give any performance advantage.



(a) Time of computation

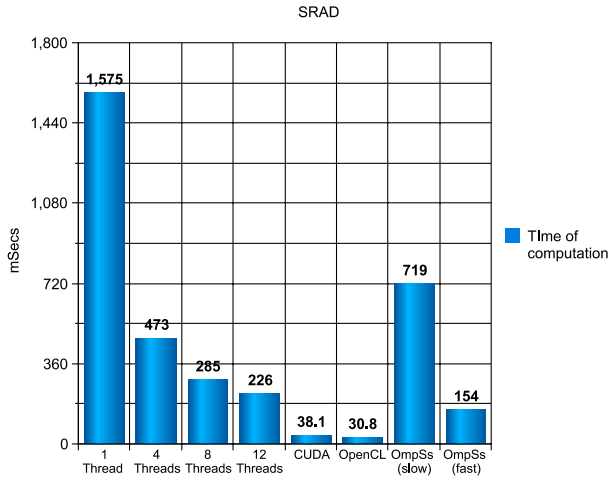


(b) Speedup

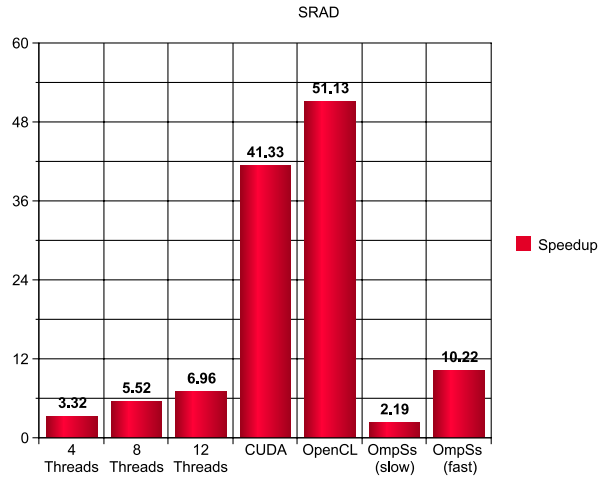
FIGURE 4.16: Kmeans - 800K

GPU TRANSFER STATISTICS 800K	
Total input transfers	370.52MB
Total output transfers	370.52MB

4.10 SRAD



(a) Time of computation



(b) Speedup

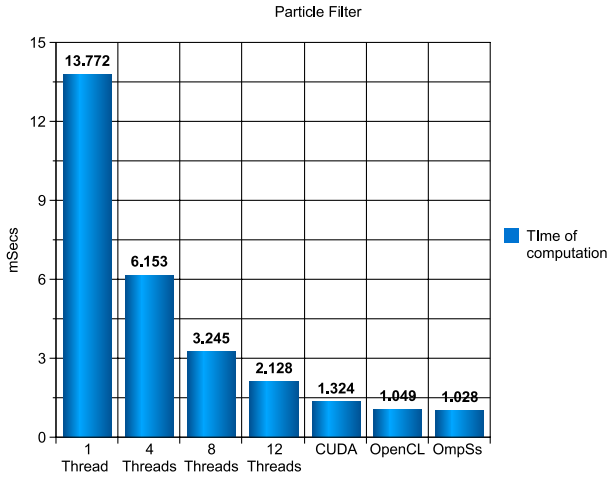
FIGURE 4.17: SRAD

GPU TRANSFER STATISTICS	
Total input transfers	572.84MB
Total output transfers	878.66MB

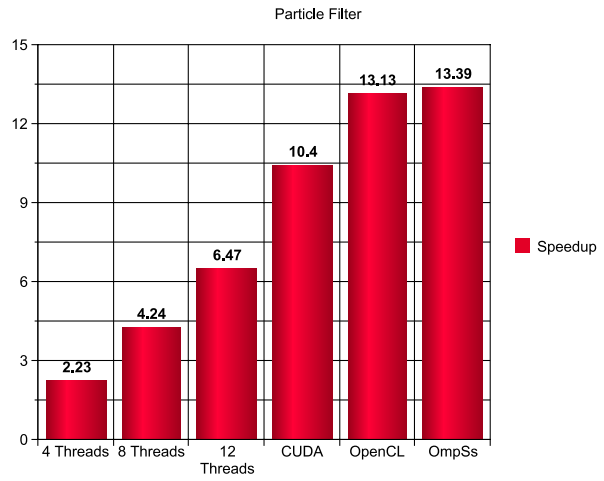
SRAD scales until 12 threads and according to [3] continues to decrease in execution time till 20 threads. Then it remains stable. The CUDA and OpenCL versions show big performance advantages to the serial one with the CUDA up to 50 speedup. In Chapter three the slow and fast versions of SRAD are explained.

4.11 Particle Filter

For Particle Filter, we used two input data sets of 50000 (50K) and 100000 (100K) particle samples.



(a) Time of computation



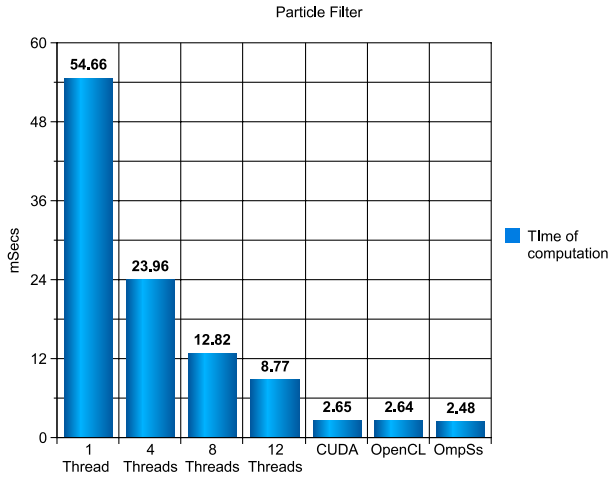
(b) Speedup

FIGURE 4.18: 5 X $10^4(50K)$

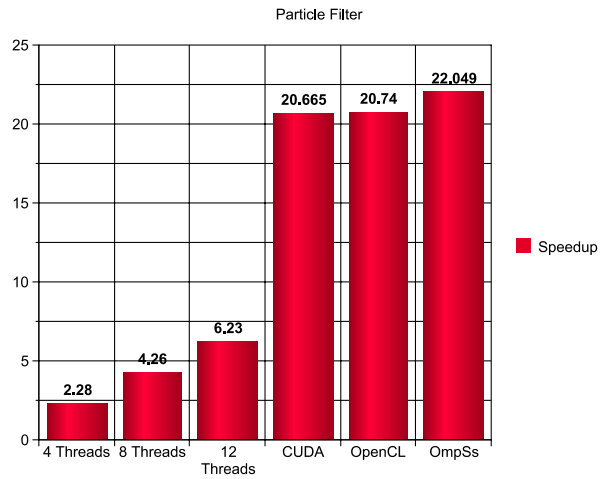
GPU TRANSFER STATISTICS	
Total input transfers	20.59MB
Total output transfers	20.59MB

The time evaluation was done in the main Particle Filter function which is the computation part. The OpenMP version shows increasing performance until 12 threads although the Speedup scaling is not linear. The OpenCL, CUDA and OmpSs versions show similar performance with a speedup of about 13 and 20 for the two datasets. The

OmpSs version is slightly faster then the other two. As the computation involved only one kernel call we used one pragma task call with a taskwait.



(a) Time of computation



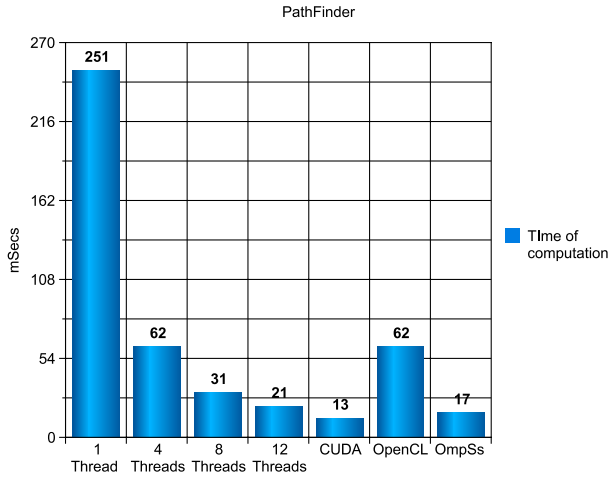
(b) Speedup

FIGURE 4.19: $10^5(100K)$

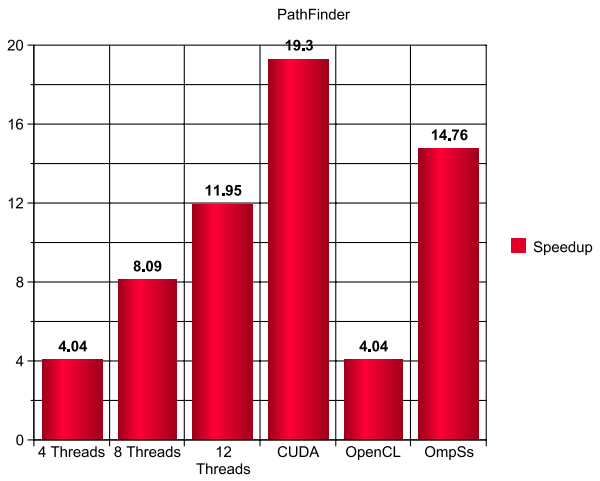
GPU TRANSFER STATISTICS	
Total input transfers	41.19MB
Total output transfers	41.19MB

4.12 PathFinder

In our experiments we use two grids with widths of 100K, 200K.



(a) Time of computation



(b) Speedup

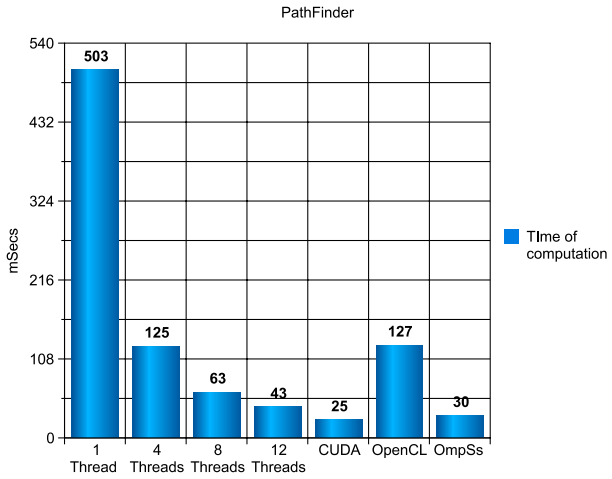
FIGURE 4.20: 100K

GPU TRANSFER STATISTICS - 100K	
Total input transfers	76.29MB
Total output transfers	1.52MB

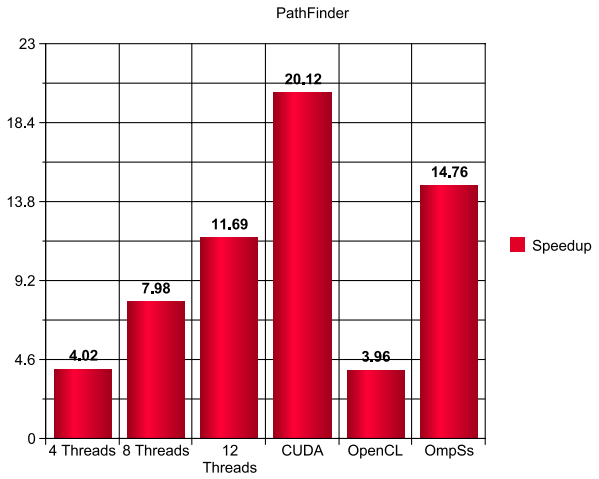
The evaluation was done in the calc path function which makes the main part of the computation. Data transfers in the GPU are included in the time measured. The OpenMP version shows good performance with a perfectly linear scaling. We note that the OpenCL version of PathFinder is significantly slower then OmpSs and CUDA

versions (it's closer to the 4-thread OpenMP version). The CUDA and OmpSs versions show similar performance for both the two datasets.

The computation consists of a main for loop and repeated calls to the GPU kernel. In the OmpSs version we made a call to the kernel with the noflush directive as we do not want any memory flush (or any memory transactions) between the calls. We also made an extra output kernel which is used with a simple taskwait for the final memory transaction from the Device to the Host.



(a) Time of computation



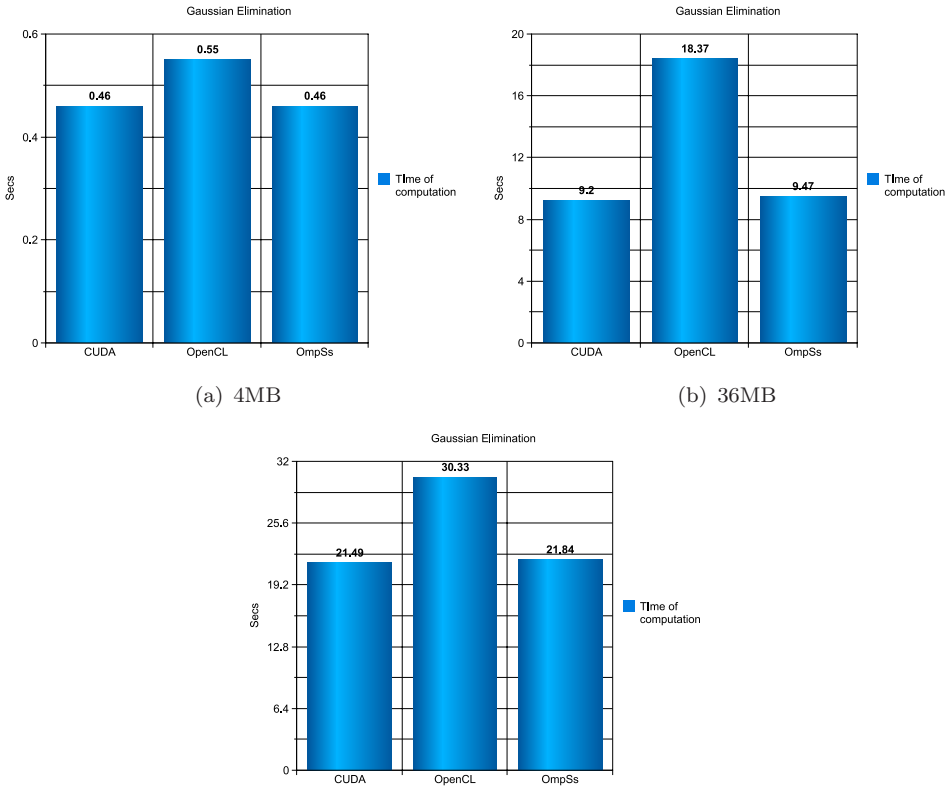
(b) Speedup

FIGURE 4.21: 200K

GPU TRANSFER STATISTICS - 200K	
Total input transfers	152.58MB
Total output transfers	3.05MB

4.13 Gaussian Elimination

We used three datasets of 4MB, 36MB, 64MB.



(a) 4MB

(b) 36MB

(c) 64MB

FIGURE 4.22: Time of Computation

GPU TRANSFER STATISTICS - 4MB	
Total input transfers	8MB
Total output transfers	8MB

GPU TRANSFER STATISTICS - 36MB	
Total input transfers	72MB
Total output transfers	72MB

GPU TRANSFER STATISTICS - 64MB	
Total input transfers	128MB
Total output transfers	128MB

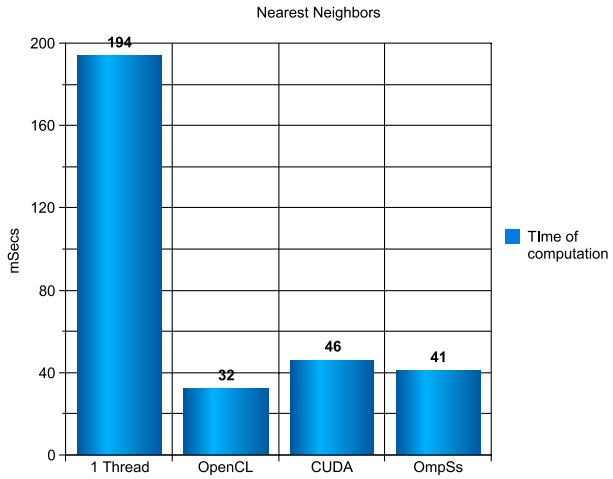
In the Gaussian elimination benchmark we have not an openMP version available. As a result we did not have a serial version to compare with. In the above diagrams we can see the pure time of computation involved in the three versions. The evaluation was done in the main ForwardSub function which contains the heart of the computation. In the OmpSs version we have two kernel calls with the noflush directive as we did not wish any memory transactions between the Device and the Host before or during the

computation. We finally constructed one final output kernel for the final result to be transfered to the host.

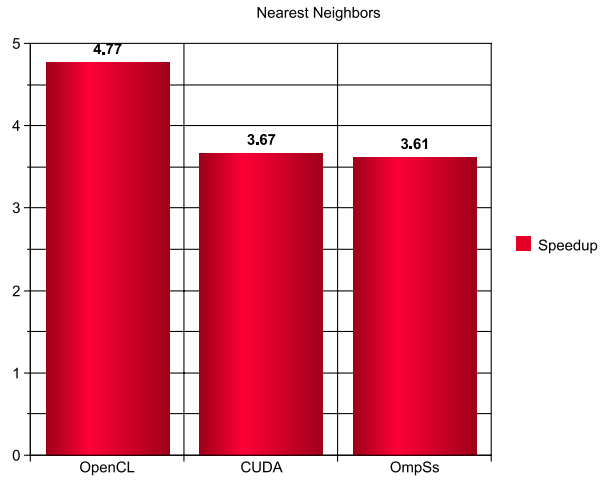
The OmpSs and the CUDA version show exatcly the same performance while the OpenCL version is quite slower. In the dataset of 32MB, for exapmple, the OpenCL version takes double time to compute.

4.14 Nearest Neighbors

For the evaluation we used two datasets with 342080 and 684160 records.



(a) Time of computation

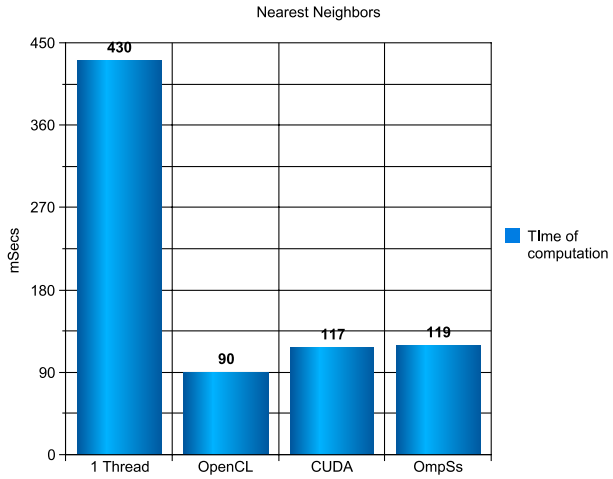


(b) Speedup

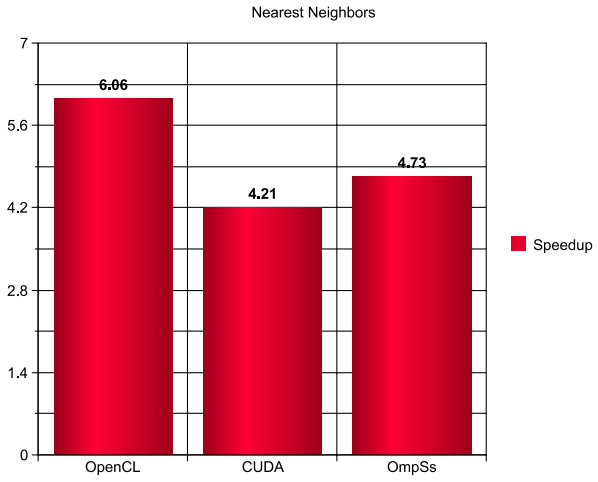
FIGURE 4.23: 342080 records

GPU TRANSFER STATISTICS - 342080	
Total input transfers	22.43MB
Total output transfers	1.83MB

The OmpSs version of Nearest Neighbor inlcudes one kernel call which is called repeatedly in the iterations involved in the computation. The call includes one copy in and one copy out utility with a taskwait. As there is a taskwait the 2 - GPU version of OmpSs gives almost the same (slightly worse) performance.



(a) Time of computation



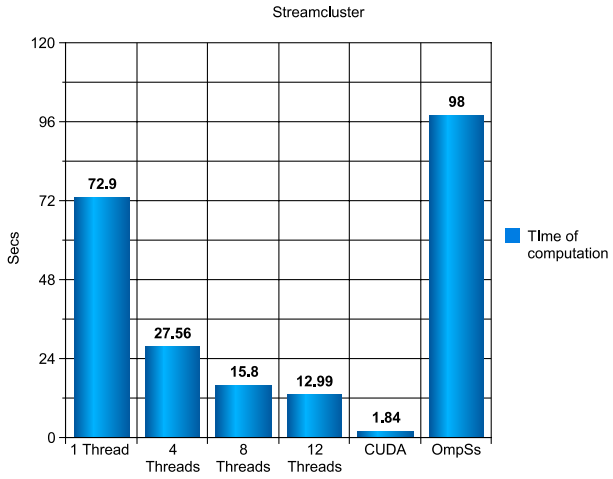
(b) Speedup

FIGURE 4.24: 684160 records

GPU TRANSFER STATISTICS - 684160	
Total input transfers	44.86MB
Total output transfers	3.66MB

The CUDA and OmpSs versions show similar performance while the OpenCL version is slightly faster in this example. These GPU versions give a performance gain of about 3-4.

4.15 Streamcluster



(a) Time of computation

FIGURE 4.25: Streamcluster

The performance of Streamcluster was quite problematic. The OmpSs version was actually slower than the serial version. In the CUDA code below we see that the big stream of data coord is copied once in the Device (in the first loop-call of the kernel).

Program 9 CUDA code

```
if (loop == 0)
{
  cudaMemcpy(coord_d, coord_h, num*dim*sizeof(float),cudaMemcpyHostToDevice);
 ...
}

pgain_kernel<<<Grid,Threads>>>(Arguments);
```

In OmpSs, however, as we have to state explicitly every time the copies in the Device the copy in has to be done every single time. In addition in this example the memory has to be flushed in the end. The repeated copy ins cause a significant slow down in performance. We note that if the taskwait on utility is enabled in the Mercurium compiler we may have control of which parts of the memory we want to be flushed. So this is probably going to solve the problem

Program 10 OmpSs code

```
#pragma omp task copy_in(coord_h[0;coord_elems])
{
  pgain_kernel<<<Grid,Threads>>>(Arguments);
}
#pragma omp task
```

Chapter 5

Conclusion and Future Work

As we saw in the analysis of the benchmarks OmpSs shows similar performance with the existing programming models. In the general case, it is slightly slower than CUDA and slightly faster than the OpenCL versions that we had available. In addition it is easier and more convenient for the programmer to use. The examples we had were not quite big in size but in bigger projects the automatic memory management of the device that is handled by the runtime system can save a lot of work and reduce the amount of errors and memory leaks. The degradation in performance that we had in some cases was caused by some utilities that the Mercurium compiler does not yet support. For example it does not support the constant and texture memories and the taskwait on utility. Especially the last one has caused the last benchmark (streamcluster) to be slower than the serial version. Finally we saw from the benchmarks that despite the convinience that OmpSs aspires to show with automatic 2-GPU execution this is not always an advantage. The benchmarks executed in the same time (or even slower) with 2-GPUs and it needs a re-design of the algorithms to gain extra performance.

OmpSs is a programming model that targets to make parallel programming for heterogeneous platforms easier. But, although it hides some memory management details and it offers more convenience it still needs expertise in parallel architectures and programming to write applications. As long as you still have to write the GPU kernels (in CUDA or OpenCL) it takes time and skill to build applications that will run correctly and fast to the GPU. In conclusion it is a programming model that is moving towards the goal

of making programming in heterogeneous platforms easier for the programmer, but the GPU part of the implementation is still tricky for the average user.

Future Work includes:

- Development with the OpenCL Kernels. OpenCL kernels are not yet supported by Mercurium.
- Development of versions for 2 GPUS. The versions that we developed did not give any performace gain for 2 GPUS or they gave wrong results. The development of 2-GPU versions needs changes in the original algorithm as the memory is not shared anymore and we have a distributed memory-like system. The Nanos runtime manages automatically the 2-GPU environment with all inputs, outputs and taskwaits and the programmer does not have to specify each time the device that has to execute each task.
- Development of fast versions for streamcluster becnhmark. Although with the taskwait on utlity that is expected to be enabled soon, the problem can be solved easily.

Bibliography

[1] Alejandro Duran, Eduard Ayguade, Rosa Badia, Jesus Labarta, Luis Martinell, Xavier Martorell, Judit Planas. OmpSs: A proposal for programming heterogenous multi-core architectures. Barcelona SuperComputing Center (BSC), Universitat Politecnica de Catalunya (UPC), Instituto de Investigacion en Inteligencia Artificial (IIIA), December 2010.

[2] Eduard Ayguade, Rosa M. Badia1, Pieter Bellens, Javier Bueno1, Alejandro Duran, Yoav Etsion, Montse Farreras, Roger Ferrer, Jesus Labarta, Vladimir Marjanovic, Lluis Martinell, Xavier Martorell, Josep M. Perez, Judit Planas, Alex Ramirez, Xavier Teruel, Ioanna Tsalouchidou and Mateo Valero. Hybrid/Heterogenous Programming with OmpSs and its software/hardware implications. Barcelona SuperComputing Center (BSC), Universitat Politecnica de Catalunya (UPC), Instituto de Investigacion en Inteligencia Artificial (IIIA).

[3] Jie Shen, Ana Lucia Varbanescu. A Detailed Performance Analysis of the OpenMP Rodinia Benchmark. Parallel and Distributed Systems Group, Delft University of Technology, 2011. URL http://www.pds.ewi.tudelft.nl/fileadmin/pds/reports/2011/PDS-2011-011.pdf.

[4] Jie Shen, Jianbin Fang, Ana Lucia Varbanescu and Kenk Sips. OpenCL vs OpenMP: A Programmability Debate Department of Computer Science, Delft University of Technology, Department of Computer Systems, VU University Amsterdam.

[5] Ioanna N. Tsalouchidou. OpenMP extensions to support dependent work distributions. National Technical University of Athens, Diploma Thesis, July 2011.

[6] NVIDIA group. *NVIDIA CUDA Programming Guide*, August 2009.

[7] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G, Szafaryn, Liang Wang, Kevin Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. Department of Computer Science, University of Virginia. http://www.cs.virginia.edu/~skadron/Papers/rodinia_iiswc10.pdf.

[8] Rodinia Benchmark. Department of Computer Science, University of Virginia https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page.

[9] Jason Sanders, Edward Kandrot. *CUDA by example*, pages 21-35, 95-137, 2011.

[10] David B. Kirk, Wen-mei W. Hwu. *Programming Massively Parallel Processors*, pages 1 - 34, 2010.

[11] Wikipedia. http://en.wikipedia.org/wiki/Heterogeneous_computing.

[12] Wikipedia. http://en.wikipedia.org/wiki/GPU.

[13] Wikipedia. http://en.wikipedia.org/wiki/GPGPU.

[14] Wikipedia. http://en.wikipedia.org/wiki/Multicore.

[15] Wikipedia. http://en.wikipedia.org/wiki/Parallel_programming.