UNIVERSITY OF THESSALY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Master Thesis

"Study of the effect of optimizations on OpenCL code when executed on different heterogeneous architectures"

Authors

Theocharidis Konstantinos

Kalogirou Christos

Supervisor

Antonopoulos Christos, Assistant Professor

Committee Members

Bellas Nikolaos, Associate Professor

Potamianos Gerasimos, Associate Professor

Volos, July, 2013

Πανεπιστήμιο Θεσσαλίας

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Μεταπτυχιακή Διπλωματική Εργασία

"Μελέτη επίπτωσης βελτιστοποιήσεων σε κώδικα OpenCL κατά την εκτέλεση σε διαφορετικές ετερογενείς αρχιτεκτονικές"

Συγγραφείς

Θεοχαρίδης Κωνσταντίνος

Καλογήρου Χρήστος

Επιβλέπων Καθηγητής

Αντωνόπουλος Χρήστος, Επίκουρος Καθηγητής

Μέλη Επιτροπής

Μπέλλας Νικόλαος, Αναπληρωτής Καθηγητής

Ποταμιάνος Γεράσιμος, Αναπληρωτής Καθηγητής

Βόλος, Ιούλιος, 2013

Ευχαριστίες

Θα θέλαμε να ευχαριστήσουμε τους καθηγητές μας, κύριο Αντωνόπουλο Χρήστο, κύριο Μπέλλα Νικόλαο και κύριο Ποταμιάνο Γεράσιμο για τη στενή συνεργασία τους και τη βοήθειά τους. Επίσης θέλουμε να ευχαριστήσουμε τις οικογένειές μας και τους φίλους μας για τη στήριξή τους.

Περίληψη

Η παρούσα διπλωματική εργασία μελετά την επίδραση διαφόρων βελτιστοποιήσεων σε κώδικα OpenCL πάνω σε διαφορετικές ετερογενείς αρχιτεκτονικές. Το ενδιαφέρον της επιστημονικής κοινότητας προς τις τελευταίες προέκυψε με τους περιορισμούς που έθεσε η κατανάλωση ισχύος και η παραγωγή θερμότητας, με την παύση της ισχύος του νόμου του Moore.

Αρχικά, ο παράλληλος προγραμματισμός επιτυγχάνονταν μέσω multi-core επεξεργαστών, many-core καρτών γραφικών, αλλά και άλλων επιταχυντών, όπως τα FPGAs. Το πρόβλημα που δημιουργήθηκε όμως ήταν ότι κάθε μία από αυτές τις αρχιτεκτονικές είχε το δικό της προγραμματιστικό μοντέλο και ο κώδικας χρειαζόταν να ξαναγραφεί για να είναι μεταφέρσιμος ανάμεσα στις αρχιτεκτονικές. Το γεγονός αυτό οδήγησε στην δημιουργία των ετερογενών συστημάτων και για αυτό το 2009, η Khronos έβγαλε το μοντέλο προγραμματισμού OpenCL. Το OpenCL εγγυάται πως ο ίδιος κώδικάς τρέχει ορθά σε όλες τις αρχιτεκτονικές, αλλά δεν μπορεί να εγγυηθεί τίποτα για την απόδοσή του. Συνεπώς, άρχισαν να διερευνούνται διάφορες τεχνικές βελτιστοποίησης οι οποίες να είναι ικανές να προσαρμόσουν την απόδοση μιας αρχιτεκτονικής σε αυτήν μιας άλλης.

Η προσπάθεια αυτή αποτέλεσε και το κίνητρο της εργασίας μας, στην οποία μελετούμε διάφορες βελτιστοποιήσεις και διάφορα υπολογιστικά πρότυπα εφαρμογών για να βρούμε συγκεκριμένα, ποιες τεχνικές βελτιστοποίησης ευνοούν ποια υπολογιστικά πρότυπα και μέσω ποιων αρχιτεκτονικών γίνεται κάτι τέτοιο. Ιδιαίτερη σημασία έχει το γεγονός ότι πέρα από τα πειραματικά αποτελέσματα που παρατίθενται, γίνεται ιδιαίτερη προσπάθεια στο να αιτιολογηθεί ο λόγος που αυτά προέκυψαν.

Οι τεχνικές βελτιστοποίησης οι οποίες εξετάστηκαν ήταν οι α) *Geometry*, b) *Vectorization*, c) *Loops* και d) *Branches* και οι εφαρμογές πάνω στις οποίες εξετάστηκαν ήταν οι *lud*, *crc*, *needle*, *srad* και *bfs*. Οι εφαρμογές αυτές αποτελούν μέρος των 13 Dwarfs του Berkeley. Οι αρχιτεκτονικές οι οποίες χρησιμοποιήθηκαν ήταν ένας Intel Xeon E5645, μία NVIDIA GeForce GTX480 και μία AMD Cayman.

Abstract

This master thesis studies the effect of different optimizations on OpenCL code when executed on different heterogeneous architectures. The interest of scientific community towards the latter arose with the saturation of Moore's law. This happened as frequency scaling began to reach its limits due to physical constraints such as power consumption and heat generation.

Firstly, the parallel programming achieved via multi-core CPUs, many-core GPUs and other accelerator devices, like FPGAs. However, the problem was that each one of such architectures had its private programming model and the code needed to be rewritten so as to be portable among architectures. This fact led to the genesis of heterogeneous systems and for this reason, in 2009, the Khronos consortium introduced OpenCL, a programming standard which guarantees functional, but not performance portability among different architectures. Therefore, developers began to search for optimizations, whose implementation, can adapt the performance of one accelerator device to the same levels of another.

This searching stimulated our interest for this project, through which we evaluate various optimizations and computational patterns in order to reveal which specific optimization techniques benefit which certain computational patterns and on which certain architectures this happens. It is very important that the underlying cause-result relation, for optimizations and parametric choices improving or degrading the performance on a particular architecture, is also sought out.

The optimization techniques that examined were the a) *Geometry*, b) *Vectorization*, c) *Loops* and d) *Branches* and the applications on which they are applied were a subset of the 13 Berkeley dwarfs. These were the applications *lud*, *crc*, *needle*, *srad* and *bfs*. Last but not least, the architectural devices that used were an Intel Xeon E5645 CPU, an NVIDIA GeForce GTX480 GPU and an AMD Cayman GPU.

Table of contents

Chapter 1

Introduction

According to Moore's Law, the number of transistors on a chip roughly doubles every two years. This law has stayed valid over the years by cramming more and more transistors into the same core. As frequency scaling began to reach its limits due to physical constraints such as power consumption and heat generation, the area of chip, known as “Dark Silicon” [13], which can’t be powered at the same time with the others due to power limitations, started to increase dramatically. It is characteristic that studies reveal that the amount of dark silicon in 22nm technology is around 20% and it predicts it will be more than 50% at 8nm. Therefore, the question is, what is the point in scaling down and increasing the number of transistors per chip if we can’t use them? This question led the scientific community to focus on the idea of heterogeneous systems and parallel computing.

The many-core CPU system, such as that of Intel i7 processor technology and the many-core architectures that support general purpose programming on GPUs via appropriate programming models, such as the CUDA (Compute Unified Device Architecture) programming model of NVIDIA, are two fundamental paradigms that support parallelism. However, the interest for better performance and parallelism has now led to heterogeneous computing, involving CPUs and other highly parallel multi-core architectures, like GPUs and FPGAs. The aim of heterogeneous computing is to overcome the problem of dark silicon by exploiting with the best manner the accelerator devices that participate in the computation process. This can be done by assigning each task to the accelerator device that can execute it faster and more power efficiently in comparison with the others. This can be derived from the fact that some devices are more able to execute certain computational patterns than others and the verification of this observation is one of the motivations of this thesis.

Additionally, the arising problems of using different programming models on different architectures pose also for heterogeneous systems. Rewriting the code for each device limits the opportunities for remapping the code on a different architecture, therefore, in 2009, the Khronos [1] consortium introduced OpenCL [2], a programming standard which supports programs that execute across heterogeneous platforms including CPUs, GPUs, FPGAs and other accelerators. The key merit of OpenCL is that it allows programmers to write code which is at least functionally independent of the underlying architecture. It provides easy-to-use abstractions and a broad set of programming APIs which are based on the C language.

However, OpenCL guarantees functional portability but not performance portability. Though OpenCL compliant, each architecture is designed according to specifications decided by its manufacturer. This creates the problem of the same program exhibiting unpredictably different performance on hardware with similar technical capabilities. The hardware can be from different manufacturers or even different generations of the same model. Functional portability is necessary

but ensuring performance portability is also essential from a developer's point of view.

If an application written in generic, platform independent OpenCL code, is not fast enough to be usable on a platform, then developers will prefer to program the application in the platform's native language or resort to platform-specific optimizations in OpenCL, which however, may limit both the functional and mainly the performance portability of the code. Nevertheless, the application might have to be written more than once for the application to work optimally on multiple platforms. As this adds to undue overhead for the developers, a more feasible solution will be to optimize the application individually for each platform. This method could be applied for exploring the OpenCL optimization space. Since programs are guaranteed to be portable, developers could tune an OpenCL program which was meant for one architecture and make it optimized for another without losing correctness.

Therefore, the goal of this thesis is to find out if there are specific standard optimizations which are good for specific computational patterns and architectures. Such optimizations could be applied automatically (maybe), alleviating the programmer from the burden of platform-specific optimizations and enhancing code portability. This can be achieved by taking a subset of the 13 OpenCL dwarfs of Berkeley [4] and apply on them a set of optimizations while they are executing on different architectures. Finally, an evaluation and analysis of these results aids in understanding one the respective architecture in more detail and also helps in interpreting which form of computational pattern is more suitable to be optimized for which accelerator device.

1.1 Contributions

The main contribution of this thesis is identifying which optimization parameters prove beneficial or not for the Intel CPU, AMD GPU and NVIDIA GPU devices that used, taking into account the different computational and communicational patterns on which they are tested. Furthermore, one more contribution is the finding of which of the afore-mentioned optimizations can keep the performance portable among which devices.

Moreover, another significant contribution is that the underlying cause-result relation for optimizations and parametric choices improving or degrading the performance on a particular architecture is also sought out. The results of the experiments are analyzed to provide a deeper understanding of the underlying architecture and to realize the suitability of specific optimizations applying on specific computational patterns and architectural devices.

Additionally, along with the afore-mentioned contributions, further gains from this thesis are that the programmers working with OpenCL will have a better understanding of how to develop programs with optimal performance and that the identified parameters can be later incorporated into a compiler which will then automatically apply the specific optimizations based on the target architecture.

1.2 Thesis Outline

This thesis is organized into seven chapters including this chapter. The organization is as follows:

- ***Chapter 2*****:** Presents the hardware architectures that used for the experiments and analyzes the most fundamental architectural parts of them.
- ***Chapter 3*****:** Gives a background perspective of the concepts and terminologies used throughout this thesis. Parallel computing and OpenCL are some of the concepts which are discussed.
- ***Chapter 4*****:** Presents the 13 Dwarfs and explains in brief their characteristics discussing the scientific areas on which they are applied.
- ***Chapter 5*****:** Discusses the related work. Prior work in exploration of optimization space, benchmarks used, etc., are presented in comparison to the work done in this thesis.
- ***Chapter 6*****:** Presents the experimental results and analyzes the HW/SW interactions on each architecture resulting to the positive and negative effect of optimizations.
- ***Chapter 7*****:** Concludes this thesis by presenting in detail the conclusions and take-home points and discussing directions of future work.

Chapter 2

Hardware Architectures

This chapter discusses in brief the hardware was used for the experiments. We focus on the characteristics that are more relevant to our study. They were a CPU and two GPUs, an Intel Xeon E5645, a NVIDIA GeForce GTX480 and an AMD Cayman.

2.1 Intel Xeon E5645

The CPU was used for the experiments belongs to Intel Xeon 5600 family and more specifically its name is Intel Xeon E5645 [9], [10]. Its architecture codename is Westmere and is built in 32nm. The following *Figure 2.1* shows how this processor is organized.

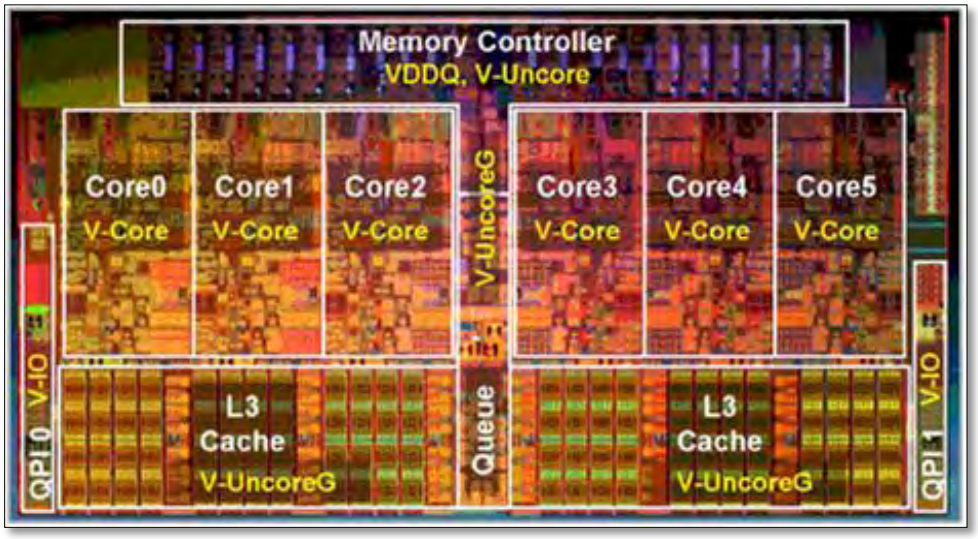


Figure 2.1: *The die of the Intel Xeon E5645* [9]

The package contains six cores and each core can run two threads simultaneously if possible, using the Hyperthreading Technology. This means that a maximum number of twelve threads running simultaneously are feasible, if the two threads of a core do not have to share common resources of this core. Otherwise the maximum number of threads is six, one in each core. The standard frequency of the processor is at 2,4GHz but under heavy workload can reach the frequency of 2.67GHz. Also the package contains a Memory Controller with max memory capacity support of 288GB and max bandwidth of 32GB/s.

Like all modern processors, the E5645 has a cache memory with different levels. There are 3 levels of cache. The first level is separate for each core with a capacity of 32KB. The second one is also separate for each core with a capacity of 256KB. The third and last one is the biggest reaching the capacity of 12MB. As it is shown in the picture, is a big part of the chipset and is common for all processors. Its position on the die is crucial and is designed next to the cores for low latency access.

This processor like his predecessors embeds some instruction sets known as SSE. It provides the latest packages the SSE4.1 and SSE4.2 which are single instruction multiple data (SIMD) instructions. The SSE4.1 adds instructions that improve compiler vectorization and provides a hint that can improve memory throughput when reading from uncacheable WC memory type. The SSE4.2 provides a rich set of string and text processing capabilities that traditionally required many opcodes.

2.2 NVIDIA GeForce GTX480

Unlike a central processor, a graphic card processor has a totally different architecture. The graphic card that is presented is the NVIDIA GeForce GTX480 [11] using the Fermi architecture.

The block diagram of Fermi is shown in *Figure 2.2*. Fermi's lithography is in 40nm. A Fermi GPU contains 15 streaming multiprocessors of 32 CUDA cores each. Each multiprocessor can manage up to 1.536 threads meaning that we can have 23.040 threads in flight (1.536 threads * 15 multiprocessors). These threads cannot run concurrently but it is useful to have so many threads in order to use another warp for computations when the current warp is accessing the memory. We can run concurrently up to 480 threads (32 CUDA cores * 15 streaming multiprocessors).

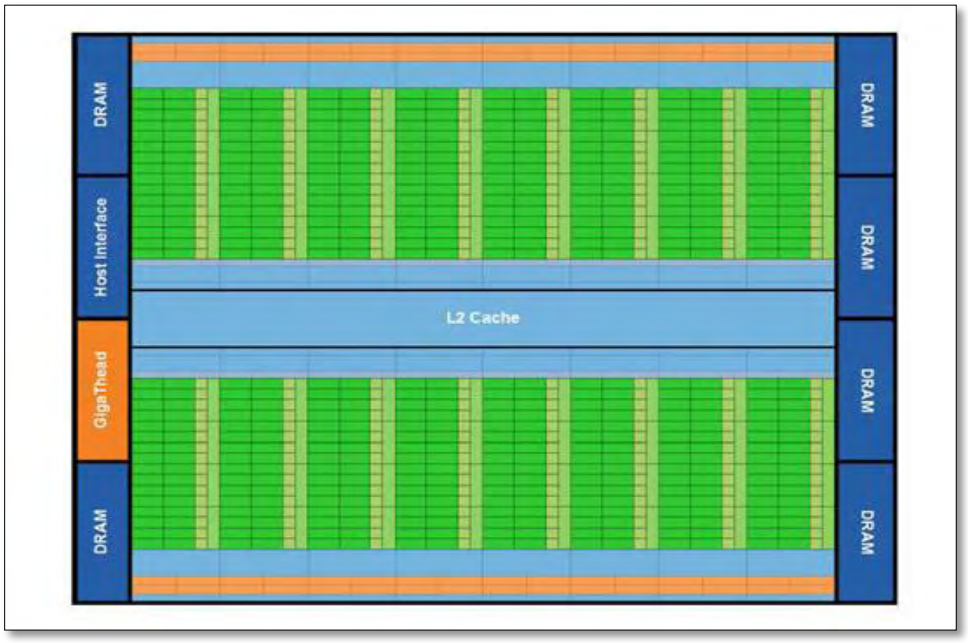


Figure 2.2: *Fermi's block diagram* [11]

The GPU uses GDDR5 memory which is partitioned in six channels. Each channel's interface is 64 bit and the memory's interface is 384 bit. Fermi can support up to 6GB of GDDR5 RAM and GTX480 has 1536MB partitioned in six chipsets. The host interface is responsible for the connection of the CPU to the GPU via PCIe. The GigaThread global scheduler distributes thread blocks to SM thread schedulers.

The next level is called streaming multiprocessor (SM) and is organized like in *Figure 2.3*. Each multiprocessor has 32 CUDA cores designed to execute 32 instructions from a bundle of 32 threads, which NVIDIA calls a warp. The cores of a SM also share the registers, the caches, the local memory, and the load/store units (LDST) of their own SM. There are 16 LDST units in each multiprocessor allowing source and destination addresses to be calculated for sixteen threads per clock. The special function units (SFUs) handle complex math operations, such as square roots, reciprocals, sines, and cosines. Each SFU executes one instruction per thread, per clock.

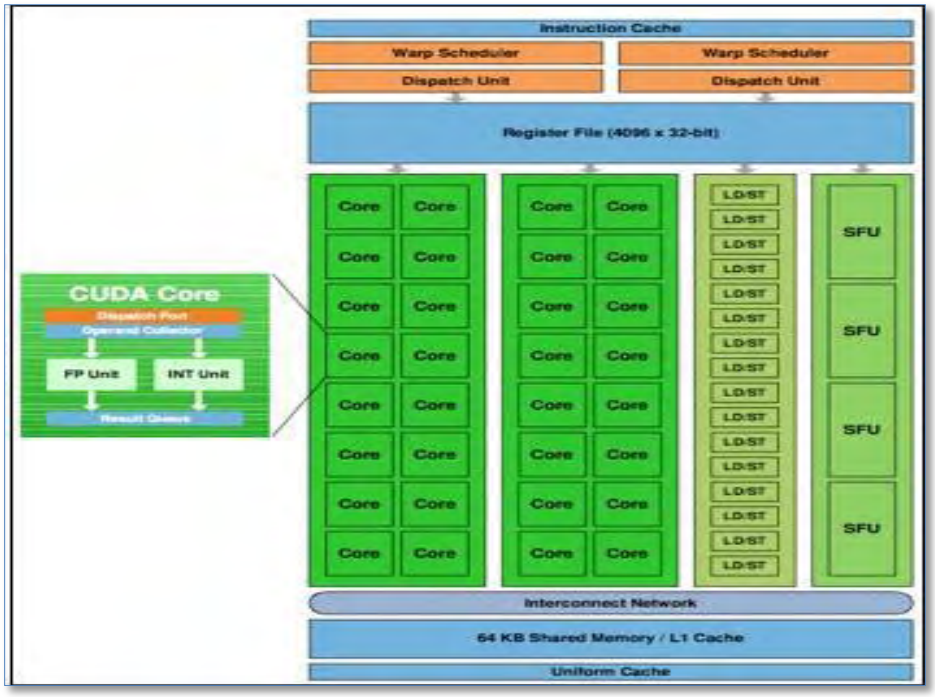


Figure 2.3: *Fermi's SM* [11]

The smallest compute unit is called CUDA core and it is shown in *Figure 2.4*. It contains a pipelined floating-point unit (FPU), a pipelined integer arithmetic logic unit (ALU), some logic for dispatching instructions and operands to these units, and a queue for holding results. It does not have its own register file or L1 cache like CPUs. It does not even have a load or store unit to access memory. A CUDA core incorporates the new IEEE 754-2008 floating-point standard, providing the fused multiply-add (FMA) instruction for both single and double precision arithmetic, improving the over a multiply-add instruction (MAD) performance by doing a multiplication and addition with a single final rounding step.

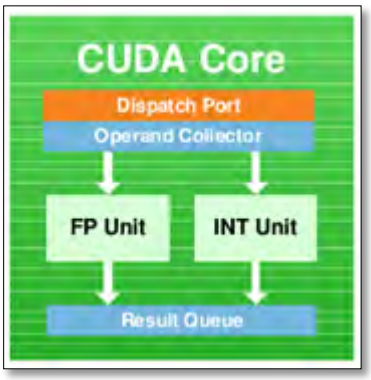


Figure 2.4: *Fermi's CUDA core* [11]

The older GPU generations were designed for graphic applications and there was not a need for a high double precision performance. But HPC applications need double precision arithmetic. Fermi is designed specially to offer unprecedented performance in double precision. It can perform up to 16 double precision fused multiply-add operations per SM per clock.

Fermi also uses a model of dual-issue, as shown in *Figure 2.5*, to achieve peak hardware performance. Each SM features two warp schedules and two instruction dispatch units. This allows two warps of a SM to be executed concurrently. The schedulers select two warps and issue one instruction from each warp to sixteen cores of this warp. Warps execute independently and the schedulers do not need to check for dependencies from within the instruction stream. The dual-issue model can execute two integer instructions, two floating instructions, or a mix of integer, floating point, load, store and SFU instructions. However, double precision instructions do not support dual dispatch.

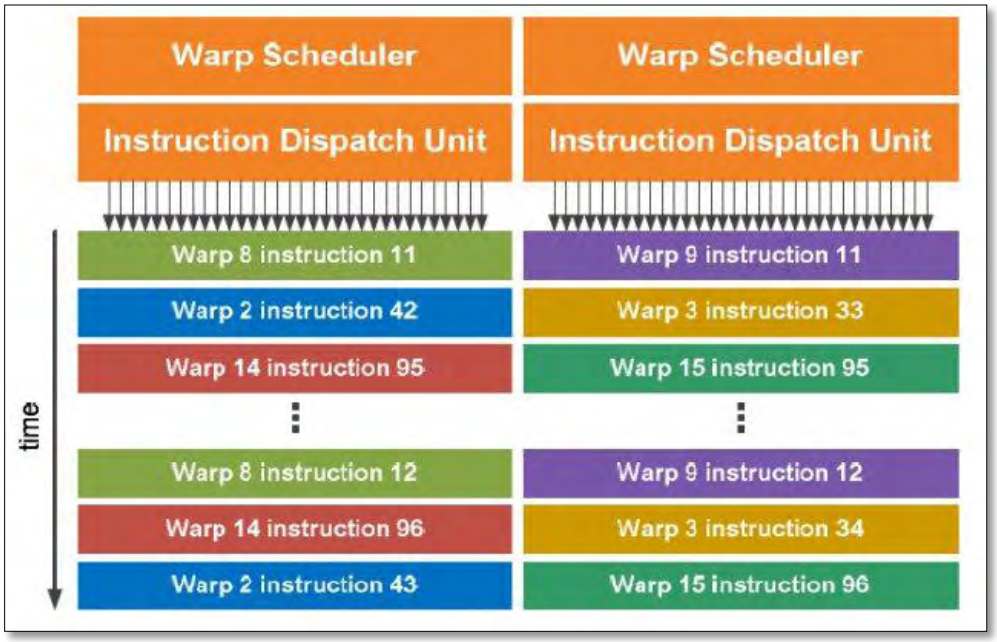


Figure 2.5: *The dual issue model* [11]

The more recent graphic cards have also different cache levels like processors because of the general purpose C or C++ programs they execute. Traditional GPU architectures support a read-only ‘‘load’’ path for texture operations and a write-only ‘‘export’’ path for pixel data output, an approach that is not appropriate for general purposes programs. The Fermi architecture addresses this challenge by implementing a single unified memory request path for loads and stores, with an L1 cache per SM multiprocessor and unified L2 cache that services all operations. The Fermi’s memory hierarchy is shown in *Figure 2.6*.

The first level of Fermi consists of the Shared Memory and the L1 Cache and is shared by the CUDA cores of a multiprocessor but its multiprocessor has its own Shared Memory and L1 Cache. The total capacity of the first level is 64KB. It is configurable and the developer can choose either 16KB of Shared memory and 48KB of L1 Cache or 48KB of Shared memory and 16KB of L1 Cache.

The second level is the L2 Cache. The capacity of this level is 768KB, is shared by all the streaming multiprocessors and services all load, store and texture requests. L2 Cache shares data efficiently across the GPU. In the block diagram of Fermi it is seen that the L2 Cache is wisely put in the center of the chipset. The distance of the L2 Cache is the same for all the multiprocessors and at the same time is very close to them for low latency access.

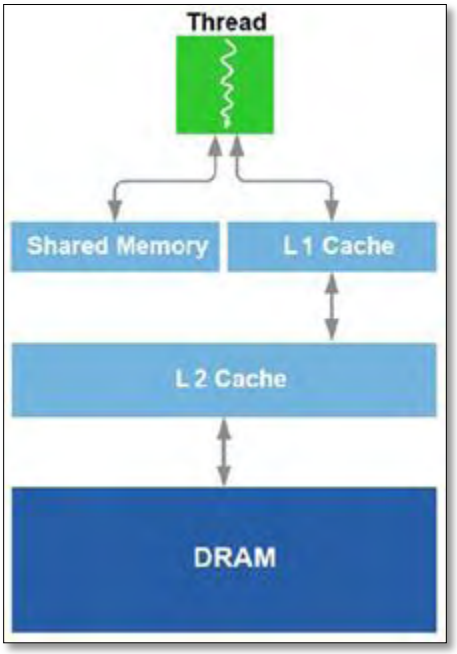


Figure 2.6: *Memory hierarchy of Fermi* [11]

Fermi also offers faster atomic operations. Atomic memory operations are important in parallel computing. Fermi secures that atomic operations such as add, min, max and compare-and-swap are performed without the interruption by other threads. These atomic operations performance is up to 20x faster than the previous NVIDIA’s generations due to a combination of more atomic units in hardware and the addition of L2 Cache.

Another important improvement is the technology of the Gigathread Thread Scheduler. This scheduler is responsible for distributing blocks to various SMs. Fermi's scheduler provides greater thread throughput, dramatically faster context switching, concurrent kernel execution and improved thread block scheduling. Context switch is up to 10x faster than previous generation. The concurrent kernel execution is shown in *Figure 2.7*. This improvement allows small kernels of the same application to run concurrently in order to achieve a better GPU utilization.

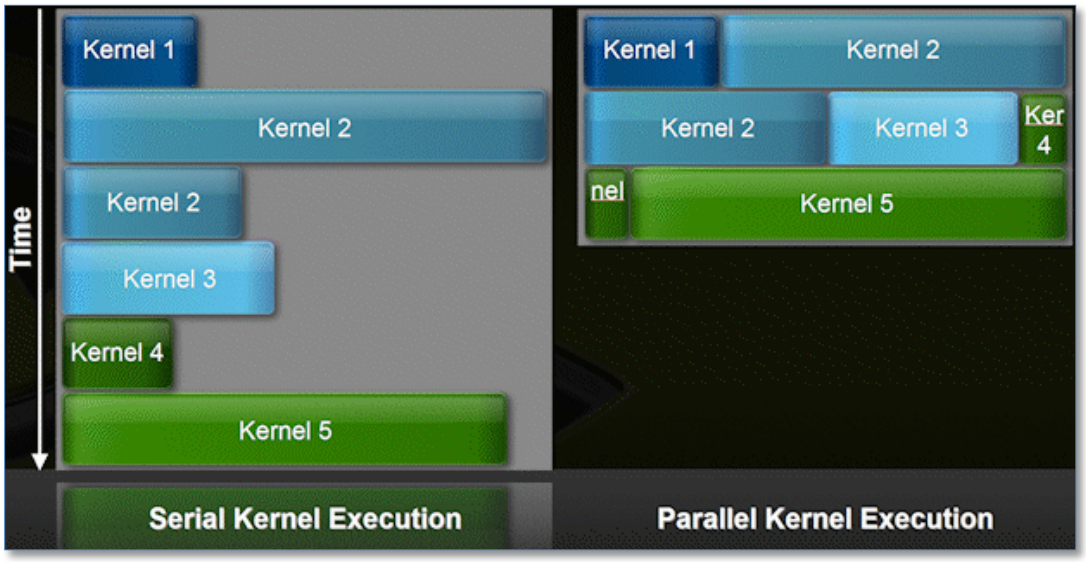


Figure 2.7: *Gigathread Thread Scheduler* [11]

2.3 AMD Cayman architecture

The second graphic card we used was an AMD Cayman [12] and its block diagram is shown in *Figure 2.8*. Cayman packs up in 24 cores, which AMD calls a "SIMD" and are tuned up to 0.88GHz and uses lithography of 40nm. Each core is a 16-wide SIMD processor and each SIMD lane is a 4-wide VLIW. So, there are totally 64 execution units in each core. NVIDIA calls a bundle of 32 threads a warp and these 32 threads must execute the same instruction per cycle. A Cayman has to execute the same instruction for 64 threads and AMD calls it a wavefront.

Each SIMD can have up to 8 work-groups in-flight. Each work-group is 1 or 4 wavefronts meaning the maximum number of wavefronts is 32 for a SIMD and when the dispatch processor schedules two wavefronts for execution in a SIMD, they will run to completion. However, the actual number of wavefronts will depend on the resources needed such as registers and local data share. Each SIMD has 16K registers which have to share between 1-32 wavefronts. These registers hold four 32-bit values, so it is critically important to hold data packed in 128-bit chunks.

Cayman includes two dispatch processors. They are responsible for managing the executing kernels and scheduling wavefronts onto the cores. Each dispatch processor is responsible for the half

of the SIMDs. Also each dispatch can launch 248 wavefronts in-flight. That means that we can have 496 wavefronts with 64 work-items each. So there can be 31.744 work-items in-flight across GPU at a given time.

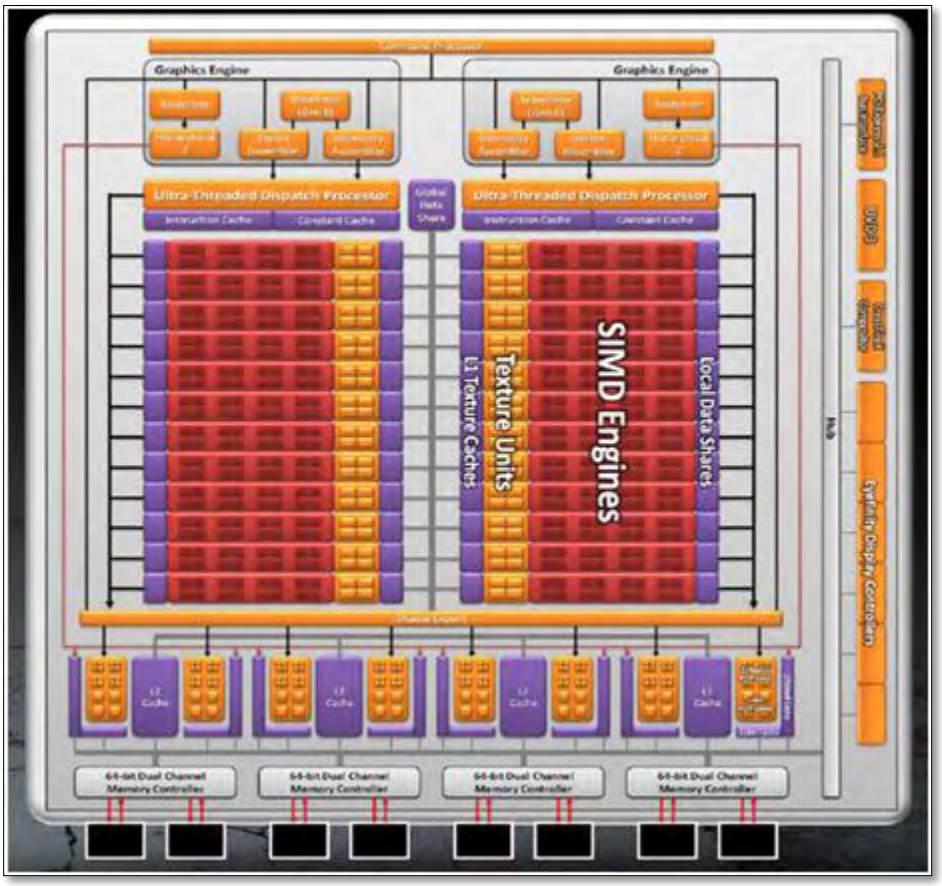


Figure 2.8: *Cayman's block diagram* [12]

The Cayman's VLIWs have been fundamentally re-designed in order to match with general purpose workloads. *Figure 2.9* shows the new architecture of a SIMD. Cayman's VLIW enhances four pipelines the XYZW pipelines to handle all the operations. AMD's VLIW pipelines are a multi-precision, staggered design that can bypass results between the pipelines. The operations in a VLIW bundle can be independent like a 4-wide SIMD, but this is not necessary.

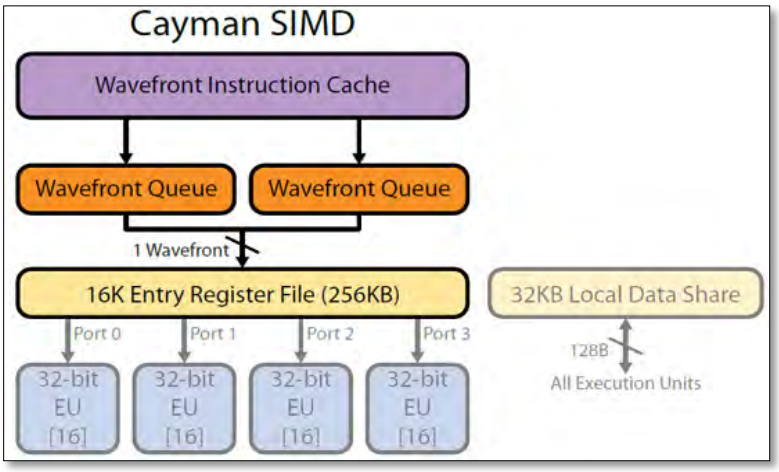


Figure 2.9: *A Cayman's SIMD* [12]

The VLIWs are statically scheduled. In contrast with NVIDIA's SM design which requires scheduling between the two issue ports. Fermi has to handle the contention for the shared load-store unit and the special function unit and also executing 64-bit instructions across both execution pipelines. AMD's approach burdens the compiler because it must find substantial instruction level parallelism (ILP) within each work-item. It is more difficult to achieve peak performance on AMD GPUs because static scheduling is less flexible.

One of the biggest problems when programming on GPUs is the bandwidth of the PCIe. Although Cayman uses PCIe2.1 and not PCIe3.0, has managed to achieve faster data transfers through PCIe. The predecessors were using two DMA controllers, each one controlled transfers in a single direction. The Cayman DMA controllers are both bidirectional, which increases the realizable bandwidth for the PCI-Express link.

Like the Fermi architecture Cayman has a similar memory hierarchy too. AMD makes extensive use of two explicitly addressed memory structures and separate read and write paths, each with specialized caches. It separates the write and read data paths for improving performance, but achieving coherence is very expensive.

The first level memory cache is in the SIMD and is called Local Data Share (LDS). It is 32KB and is exposed through the OpenCL and Direct Compute specifications, which require a 16KB and 32KB array. This level is shared only by the work-items of a work-group. LDS is a 32 way banked structure and includes bank conflict detection serializing the access to each bank. So, N accesses in the same bank means that the bandwidth will be reduced by a factor of N.

The second level memory is the Global Data Share (GDS). It is 64KB and is shared by the entire GPU. It is similar to LDS but it is used for the communication for an entire kernel and not a work-group. It is also 32 way banked structure and is not a part of a SIMD. It is a globally shared structure and available to each SIMD. The GDS is a structure that does not correspond to anything in the OpenCL or DirectCompute specification unlike the LDS.

Cayman also offers two levels of non-coherent texture caches which are read-only memories. AMD uses virtual memory addresses and the Texture Memory Unit (TMU) is responsible for translate

these addresses for accessing the texture cache. The L1 cache memory is 8KB and is organized in 64B lines, so that aligned accesses in each quarter wavefront will target a single cache line. This can achieve a performance of 1.3TB/s.

Maximizing the bandwidth helps the GPUs to hit high performance. One of the keys for maximizing bandwidth is memory coalescing. Memory coalescing means grouping together aligned reads or writes that have good locality. Each coalesced memory access can use a single address and request, while moving many pieces of data. AMD uses two levels for exploiting locality. The first level is maximizing the utilization of each 128-bit register, which can hold four 32-bit data values. The second level of locality is coalescing together 4 different 16B accesses into a single cache line access.

The last level of cache memory is the L2 cache. The L2 cache is shared by all the SIMDs and is 512KB. The L2 is partitioned with a 64KB slice for each of the 8 GDDR5 memory channels. These slices can read a 64B line per cycle achieving a bandwidth of 450GB/s to the L1 caches and L2's intent is to exploit locality of data too.

Each memory channel uses a write combining cache that multiple writes are coalesced to a single cache line into one transaction. The WCCs also buffer up many writes so that they can be performed in a single batch and achieve maximum bandwidth. Simple stores can proceed from the WCC to the memory controllers. AMD uses the term FastPath for that. However, more complicated memory accesses go through additional hardware and require extra latency and bandwidth, along what AMD terms the CompletePath.

The highest level in memory hierarchy is the GDDR5 memory. The memory controllers are 64-bits wide and drive two 32-bit GDDR5 channels. There are 8 channels for the communication with the graphics memory. The memory controller's intent is to maximize bandwidth via memory coalescences and write buffering.

2.4 Comparison of a GPU and a CPU

A GPU is similar to a CPU if they are considered as a compute device. But there are a lot of differences between their architectures.

The first difference is the way they like to execute instructions. A GPU emphasizes massively threaded throughput and SIMD performance, rather than the latency of a single instruction stream. A CPU is clocked very higher than a GPU and is good at executing the instructions single-threaded. The higher clock speed helps them to compensate for the lower number of cores. On the other hand a GPU is not efficient in a serial code.

The second difference lies in the control unit. Each core of a CPU has its own control unit and can run independently from the others, meaning that each core can execute different blocks of instructions at a given time. The cores of a GPU cannot do that. There is a control unit in each SM or SIMD. So, the cores of a SM or a SIMD have to execute the same instructions in parallel.

There is also a difference in caches. A CPU is designed to have many levels of cache memory and the last level has usually a big capacity. Since later years GPUs did not use caches, they started to use caches when it was found that they are useful for general purpose applications. Latest generations have also different levels of cache memory but they do not have so many levels as a CPU. Also they have smaller cache memories than CPUs and some GPU architectures, as the Cayman have read-only caches.

Another difference is that GPUs usually have vector processors which do not have advanced features such as branch prediction, and out of order execution. This enables GPUs to have much higher number of computational units (see *Figure 2.10*) due to the lesser complexity and the larger space available on the die. CPUs however have such advanced features since they are designed for general-purpose computation.

Last but not least, one very important difference between them is the context switching. On GPUs, thread context switching is implemented in hardware, which enables it to switch between thousands of threads very quickly. CPUs depend on the operating system to take care of context switching and this is much slower.

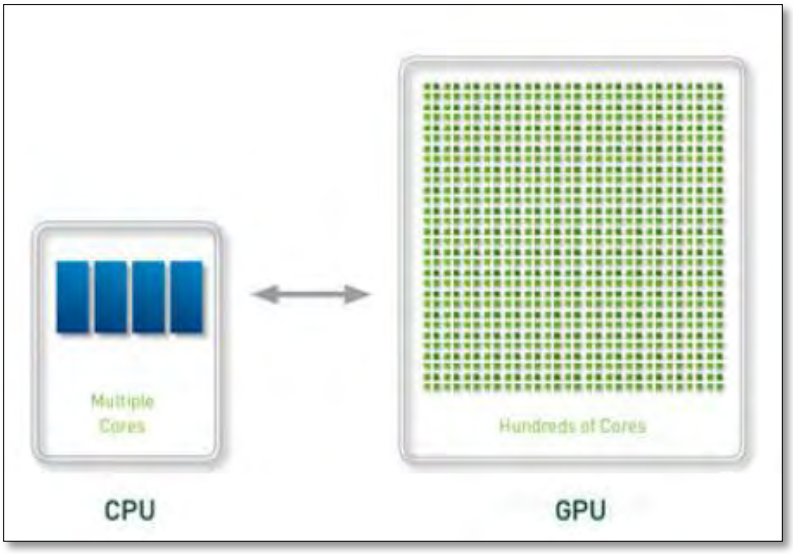


Figure 2.10: *CPUs consist of a few number of cores (2 to 6), whereas GPUs consist of hundreds of cores*

Chapter 3

Basic Concepts of Parallel Computing and OpenCL

3.1 Parallel Computing

In this section, fundamentals of parallel computing will be presented.

Dual-core and quad-core processors on CPUs, general-purpose GPU computing and more recently, heterogeneous computing are the most basic forms of parallelism. In heterogeneous computing, tasks are executed in parallel on CPUs, GPUs, FPGAs and other devices obtaining unprecedented levels of performance.

3.1.1 Flynn's Taxonomy

According to Flynn's taxonomy [3], architectures are classified based on the presence of single or multiple streams of instructions and data. There are four classifications as listed in the *Figure 3.1*. The descriptions are provided below.

SISD: An architecture in which a single processor executes a single instruction to operate on data stored in a single memory.

SIMD: An architecture in which multiple processing elements execute the same operation on multiple data simultaneously.

MISD: An architecture in which multiple processing elements perform different operations on the same data. This can be seen in a pipeline architecture where the same data moves along a pipeline and different operations are performed on it.

MIMD: An architecture in which different processing elements perform different operations on different pieces of data. The data can be stored in a shared memory or a distributed memory.

Flynn's Taxonomy		
	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Figure 3.1: *Flynn's Taxonomy classifies architectures into four categories based on the presence of single or multiple streams of instructions and data* [3]

3.1.2 Levels of Parallelism

In this section, the three levels of parallelism are introduced, namely, instruction-level parallelism, task-level parallelism and data-level parallelism.

***Instruction-level Parallelism*:** In instruction-level parallelism (ILP), more than one instruction is executed during a single clock cycle. Though the program to be executed might be following a sequential execution model, various micro-architectural techniques such as out-of-order execution or pipe-lining can be applied to exploit ILP.

***Task-level Parallelism*:** In task-level parallelism, each processor executes a different thread or process on the same or different data. For e.g., in a dual core processor, two different cores can execute two different threads at the same time. If the threads are part of the same process, the data being worked upon can be the same. Task-parallelism emphasizes on distributing the process or thread across parallel processing nodes.

***Data-level Parallelism*:** In data-level parallelism, each processor executes the same thread or process on different data. For e.g., adding two vectors can be done in a single clock cycle if there are as many processors as the number of additions to be performed. This model is in sync with the SIMD model. Data-parallelism emphasizes on distributing the data across parallel processing nodes.

3.1.3 Processor Architectures

A multi-processor system, as its name suggests is a single computer system which has multiple processing nodes. Multi-processors can be classified based on their execution model. Various processor architectures which form the basis and are implemented in many of the current CPUs and GPUs are presented along with a discussion of their features and shortcomings.

Vector processors: In vector processors, there are multiple, pipelined functional units which has the capability to execute single instructions on vectors or arrays of data. All the functional units execute the instructions in lock-step fashion on the local data. According to Flynn's taxonomy, vector processors follow the SIMD model. Vector processors are very power-efficient as the units consist of simple execution units. There is no instruction checking done at runtime and no other complex features implemented in the processor. Taking the simplicity into account, the space required for the units on the die is also considerably smaller, thereby leading to higher number of units and more power efficiency.

VLIW processors: The VLIW architecture takes advantage of ILP, by executing multiple instructions in parallel but the difference being that the schedule of instructions is determined when the program is compiled. It has multiple execution units like vector processors, but it is capable of executing different instructions at the same time. The VLIW architecture is more power hungry than vector processors. Unlike super-scalar processors, the schedule of instructions is statically determined by the compiler, rather than by the processor.

Super-scalar processors: In super-scalar processors, multiple functional units are available on the processor so that multiple instructions can be executed per clock cycle. Data dependencies between instructions are dynamically checked at runtime for doing this. Super-scalar processors are different from multi-core processors where the redundant units are entire processors and parallelism is achieved by executing one thread per core. Though super-scalar processors process multiple data items in a single clock, they do not process multiple data items for a single instruction. Super-scalar processors are much more power-hungry than VLIW and vector processors due to their dynamic behavior. The units are more complex due to added functionalities such as out-of-order execution, branch prediction, etc.

Multi-core processors: Multi-core processors contain multiple independent cores on a single chip (also known as chip multiprocessor). Though the cores are independent, they do share some resources such as cache memories, main memory between them. Sharing cache memories aids in exhibiting task-parallelism where the cores can work on the same data simultaneously. In addition, implementing multiple functional units (such as ALUs) in a single core aids in data-parallelism. According to Flynn's taxonomy, it follows the MIMD model. Multi-core processors can implement super-scalar or vector architectures or even a hybrid of both for added performance benefits. Most real programs get maximum benefit with a mixture of both data-parallelism and task-parallelism. Most of the processors being manufactured now, try to support a combination of these configurations.

Many-core processors: Many-core processors are similar to multi-core processors but with a much higher number of cores. It is not required that all the cores have to be all on a single chip, but all the cores will be in a single processor package. They are designed for a higher degree of parallelism,

supporting advanced levels of scalability. Many-core processors follow the MIMD model. They usually consist of simpler elements such as vector processors, whereas multi-core processors usually consist of more complex elements such as super-scalar processors. Each core in many-core processor is simple, small, and independent from each other. Typically, a multi-core processor will have fewer cores (two to six) whereas many-core processors usually have 32 or more cores.

3.2 Open Computing Language (OpenCL)

OpenCL is an open industry standard maintained by the Khronos Group for writing programs that execute across heterogeneous computing devices such as CPUs, GPUs and other processors. The OpenCL framework provides a runtime system, libraries and a programming language which is an extension to the standard C language (based on C99). This helps programmers to develop portable general-purpose software which can take advantage of all the different platforms that support OpenCL.

The write once, run anywhere behavior of OpenCL is the one major property which sets it apart from other such languages for the GPU. During runtime, the OpenCL code is compiled just-in-time for the particular architecture and hence the programmer needs not bother about which target architecture the program will be running on, as long as it supports OpenCL. OpenCL supports both data-parallel and task-parallel programming models, as well as the hybrid of them. Primarily driving the design is the data-parallel model. It also provides a broad set of programming APIs using which developers can query and identify the actual device capabilities and create efficient code.

3.2.1 Platform Model

The OpenCL specification defines a platform as a host connected to multiple OpenCL devices which are composed of a number of compute units. Compute units can be further divided into a number of processing elements. *Figure 3.2* illustrates how all of these devices interact together. A brief description for each is provided below.

Host: A host usually consists of a CPU and is responsible for running the host application. The host application runs natively on the host and submits commands to the OpenCL device. The commands to be submitted are queued up in a data structure called the command queue which is then scheduled onto the device. Execution of kernels, reading and writing of memory objects are examples of some of the commands which are submitted.

Devices: A device can correspond to a multi-core CPU, a GPU, a FPGA, a APU etc. A single device is composed of a number of compute units, such as the individual cores in a multi-core CPU.

An aspect to be noted of this model is that, provided the host device also supports OpenCL, programmers can partition a program into serial code and parallel code which are best suited for the CPU and the GPU, respectively. Thereby, the execution can go back and forth between the devices making the best utilization of them.

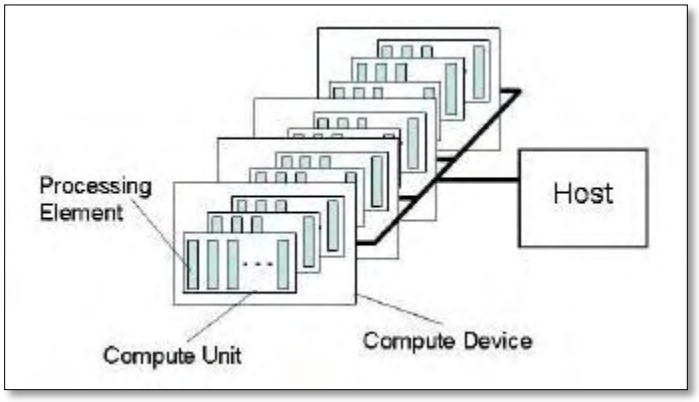


Figure 3.2: *The OpenCL Platform Model specifies a host which is usually a CPU connected to multiple OpenCL compute devices such as GPUs or APUs. The compute devices consist of a collection of compute units (cores) which are further composed of multiple processing elements* [2]

3.2.2 Execution Model

In this section, the OpenCL execution model is presented. Before talking further about the execution, the various terms used are introduced.

Program: An OpenCL program consists of one or more kernels and auxiliary functions which are used by the kernels. Programs are written in an OpenCL-C language. The language has extensions which are for e.g., specifying memory spaces and also additional keywords for specifying a function as a kernel function. The OpenCL compiler which is a part of the runtime compiles programs to create binaries which can be executed or saved for later loading.

Kernel: The kernel is a function in an OpenCL program that is executed on a device. The return types of kernels are always void as all inward and outward communication is done through the memory. All the necessary operations such as copying of memory objects, setting kernel arguments etc., required for the kernel execution are managed by the host application.

Work-Item: Instances of the kernel are executed in parallel on the compute units of the device. The same kernel code is executed by all the work items concurrently, but the specific path taken can vary based on the algorithm. Work-items are identified in two ways - one is by using a global ID and the second way is through a combination of a local ID and work-group ID, which will be explained in detail in the next segment.

Work-Group: As mentioned before, a collection of work-items are assigned for execution on a single compute unit. This collection of work-items is called as a work-group. When a kernel is enqueued for execution, two parameters pertaining to work-groups can be specified, which are the global work size and the local work size. The global work size is the total number of kernel instances or work-items that are to be started for computation, whereas the local work size is the number of work-items that are assigned to one work-group. So the number of work-groups will be always equal to the global work size divided by the local work size. If the local work size is not specified, then the OpenCL implementation will decide how to break down the global work-items into appropriate work-groups. In case there are more work-groups than the available number of compute units, the work-groups will be scheduled one by one on the compute units. A compute unit will always concurrently finish executing the work-items in one work-group before executing work-items from another work-group.

In OpenCL, the execution model is based on parallel execution of the kernel, with the process involving both the host and the compute device. The steps involved in kernel execution are listed in the *Figure 3.3*. The *Figure 3.4* shows how each of the steps presented in the *Figure 3.3* relate to the host and the compute device. Although the host is required for the initial setup of the execution process, the compute device executes the kernel independent of the host and so the host can perform other computations in the meantime. The only way for the host and the compute device to communicate is by copying data from the host memory to the device memory and vice versa. Debugging OpenCL programs are therefore quite difficult as the only way to ensure computation is done correctly is to copy the data back to the host and then verify the output.

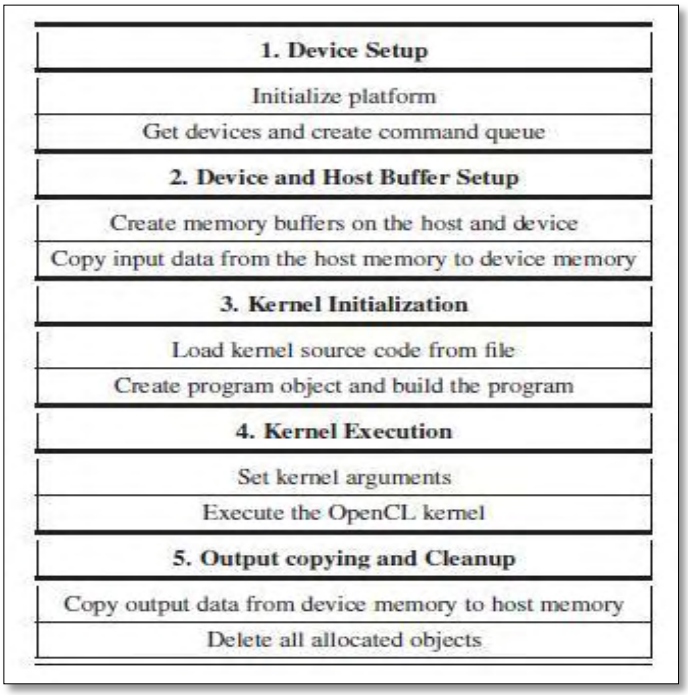


Figure 3.3: *The steps involved in an OpenCL kernel execution* [2]

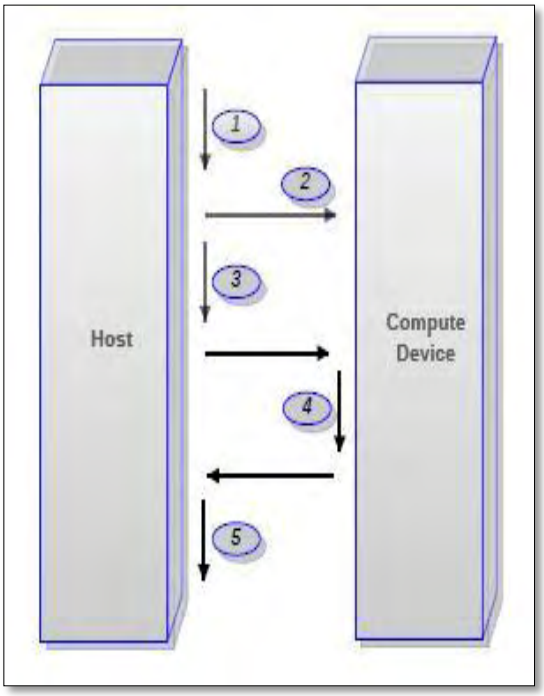


Figure 3.4: *The interactions between the host and compute device for a kernel execution* [2]

3.2.3 Memory Model

In this section, the memory model of OpenCL is presented. The memory in OpenCL devices is classified into five regions - global, constant, texture, local and private. The task of deciding the region to store data is solely up to the programmer. *Figure 3.5* shows the memory hierarchy as defined by OpenCL. A brief description of each of the memory regions is provided below.

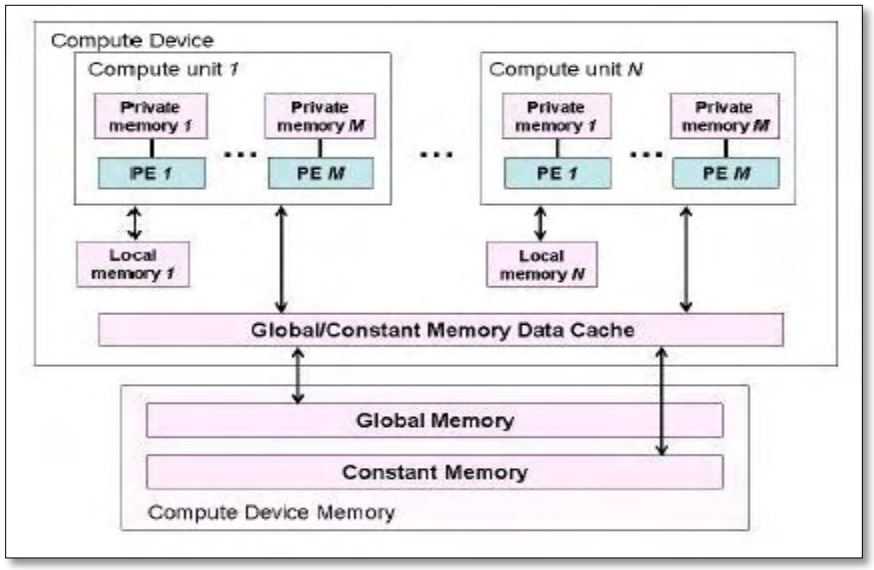


Figure 3.5: *The OpenCL Memory Model defines four regions of memory accessible to work-items while executing a kernel. The global memory in the compute device is accessible to the host also* [2]

Global Memory: The global memory region is the main means of communication between the host and the device. The host can create read/write buffers on the global memory of the device using commands. It is accessible to all the work-items and a programmer can use the global address space qualifier in a kernel to denote that an object is to be stored in global memory. Though it is usually the largest space available on the device, the access latency is also much larger compared to the other regions.

Constant Memory: The constant memory is a part of the global memory, but the difference between them is that while the host can read and write into this region, the kernel has read-only access. The constant address space qualifier is used to denote that an object is to be placed in constant memory.

Local Memory**:** The local memory is named as such as it is the memory that is available only to a local work group. In other words, the local memory is made private to a compute unit. So the compute units local memory will be accessible to all the work-items that are part of the same work-group. The space available is much smaller, but has low latency compared to the global region. Using the local

memory, work-items in a work-group can share data among them quickly and synchronize their execution. The local address space qualifier is used to denote that an object is to be placed in local memory.

***Private Memory*:** This memory region is accessible to only a single work-item. This is the fastest and the smallest memory that is available to a work-item. The private address space qualifier is used to denote that an object is to be placed in private memory.

Chapter 4

The 13 Dwarfs

The increasing proliferation of heterogeneous computing platforms presents the parallel computing community with the challenge of evaluating the efficacy of such parallel architectures, particularly given the diversity of hardware architectures and their associated (non-interoperable) programming environments.

OpenCL and 13 Dwarfs or OCD for short [4] is a benchmark suite that aims to provide a future-proof software methodology to enable the evaluation of hardware innovation across a variety of architectures. To this end, application kernels following computation and communication patterns are selected from the Berkeley 13 Dwarfs. The focus is on these because they offer a diverse set of patterns, each of which is relevant across a variety of domains. For example, the n-body method is relevant across physics, chemistry, and a variety of other domains.

Overall, it is believed that OpenCL and the 13 Dwarfs will provide a useful baseline for the evaluation of platforms and runtime systems across application domains. In the future, representative applications for each dwarf will be prepared hoping that in this way a set of implementations will be created which may be used to make generalizations about the higher-level patterns and the effectiveness of a given platform for executing a given pattern.

An indicative brief description of each of the 13 Berkeley dwarfs follows:

***Dense linear algebra*:** Consists of dense matrix and vector operations. It has a high ratio of math-to-load operations and a high degree of data interdependency between threads.
Application areas: Linear Algebra (LAPACK, ATLAS), Data Mining (Streamcluster, K-means)

***Sparse linear algebra*:** Solves the same problem as dense linear algebra but has matrices with few non-zero entries. To reduce space and computation, such algorithms store and operate on a list of values and indices rather than proper matrices, resulting in more indirect memory accesses.
Application areas: Finite Element Analysis, Partial Differential Equations

***Spectral methods*:** Transform data from/to either a spatial or temporal domain. The execution profile is typically characterized by multiple stages of processing, where dependencies within a stage form a "butterfly" pattern of computation.
Application areas: Fluid Dynamics, Quantum Mechanics, Weather Prediction

***N-body methods*:** Calculate interactions between many discrete points and are characterized by large numbers of independent calculations within a timestep, followed by all-to-all communication between timesteps.
Application areas: Molecular Modeling, Molecular Dynamics, Cosmology

***Structured grids*:** Organize data in a regular multidimensional grid, where computation proceeds as a series of grid updates. For each grid update, all points are updated using values from a small neighborhood around each point. The neighborhood is normally implicit in the data and determined by the algorithm.
Application areas: Image Processing (SRAD), Physics Simulations (HotSpot)

***Unstructured grids*:** Possess data structures, e.g., linked list of pointers that keep track of the location and neighborhood of points which are used to update the location. Like sparse linear algebra, updates typically involve multiple levels of memory reference indirection, as an update to any point requires first determining a list of neighboring points and then loading values from those neighboring points.
Application areas: Computational Fluid Dynamics, Belief Propagation

***MapReduce*:** Captures the repeated independent execution of a map function and results are aggregated at the end via a reduce function. No communication is required between processes in the map phase, but the reduce phase requires global communication.
Application areas: Distributed Searching, Sequence Alignment, Parallel Monte Carlo Simulations

***Combinational logic*:** Exploits bit-level parallelism in order to achieve high throughput. Such a workload involves performing simple operations on very large amounts of data.
Application areas: Encryption & Decryption, Hashing

***Graph traversal*:** Visits and evaluates a number of objects in a graph. Such algorithms typically involve a significant amount of random memory access for indirect lookups. The bottleneck is generally due to access latency rather than access bandwidth.
Application areas: Searching, Sorting, Collision Detection

***Dynamic programming*:** Solves a complex problem by solving a series of simpler subproblems.
Application areas: Graph Problems (Floyd's All-Pairs shortest path, Bellman - Ford algorithm),
Sequence Alignment (Needleman – Wunsch, Smith - Waterman)

***Backtrack & branch-and-bound*:** Approaches generally search a very large search space to find a globally optimal solution. Because the search space is so large, an implicit method is needed to prune the search space to make this approach computationally tractable.
Application areas: Artificial Intelligence (N – Queens), Integer Linear Programming,
Boolean Satisfiability, Combinatorial Optimization

***Graphical models*:** Map variables into nodes and conditional probabilities into edges, e.g., Bayesian networks.
Application areas: Computational Biology (Sequence homology search), Machine
Learning (Hidden Markov Models), Embedded Computing (Viterbi decode)

***Finite state machines*:** Capture a system whose behavior is defined by states, transitions defined by inputs and the current state, and events associated with transitions or states. These dwarf algorithms are highly dependent on conditional operations and interdependent data, which are also commonly found in graph traversal.
Application areas: Video Decoding, Parsing, Compression,
Data Mining → Reverse Engineering the Brain

Chapter 5

Related Work

In this section some indicative related works will be presented about the performance portability of OpenCL on heterogeneous architectures.

There are few publications that have studied the performance portability of OpenCL on heterogeneous architectures. In [5] Rul *et al.* have studied the performance portability of OpenCL. The study has been done on four different architectures, including an Intel Core i7 720 - QM (a 1.60 GHz quad-core), an NVIDIA Tesla c1060, an ATI FirePro V8700 and a Sony/Toshiba/IBM Cell Broadband Engine (Playstation 3). For the experiments, three Parboil benchmarks (cp, mri-fhd, mri-q) and three optimization parameters (loop unrolling, vectorization, number of threads in a thread block) were used.

They found that when optimizing for CPUs, loop unrolling is not an important optimization for these kernels, but loop unrolling is crucial for good performance on Cell. For Tesla, the thread block size is the most important parameter, while big loop unrolling factor degrades performance on Tesla. Also, while the Tesla and FirePro respond largely similarly to the optimizations, the FirePro is much more sensitive to parameter values for the optimizations and it has different optimal parameter values than the Tesla. Furthermore, when adding vectorization, this degradation becomes even larger and the optimal unrolling factor becomes smaller.

They conclude that the impact of the optimizations varies hugely between architectures, confirming the need for architecture-specific optimization and that the OpenCL does not support performance portability. However, they have only observed the behavior of the benchmarks and have not looked into the causes for the behavior. In this thesis, this has been improved upon by explaining the rationality behind the behavior of the benchmarks for each architecture.

Another indicative work about the OpenCL portability and performance on software heterogeneous architectures is a master thesis [6]. In this survey, well-known image-processing algorithms (binarization, dct, convolution, sum, histogram) are evaluated on three different architectures (NVIDIA Fermi GTX 470, AMD Evergreen HD 5850, Intel Core Nehalem i7-930).

The author has shown that for GPUs the basic optimizations that are useful someone to implement on OpenCL code are the maximization of parallel execution, the memory access coalescing, the local memory usage and the loop unrolling. The architecture of the AMD GPU is comparable to the architecture of the NVIDIA GPU, but one significant optimization parameter that has extra the AMD is the VLIW packing. Efficient use of AMD GPU hardware requires that the kernel contains enough parallelism to fill all VLIW slots. This can be achieved in two ways, by loop unrolling or by

using explicit vector data types.

On the other side, this thesis states that for OpenCL to reach hardware efficient implementations on a CPU a different mapping as compared to the GPU implementations is required and some important optimization techniques are the maximization of parallel execution and the exploitation of cache hierarchy. The available parallelism on a CPU is limited to the width of the vector units inside a core multiplied by a relatively small number of cores, while for the cache hierarchy to improve the performance of a program, the program should contain spatial and/or temporal locality. Moreover, it is recommended that in order to improve the performance on an Intel CPU, one must avoid to use local memory and barrier synchronization instructions, offered by OpenCL. Local memory for CPUs is mapped onto a region of the global memory and it isn't low-latency, while the barrier synchronization is unnecessary if someone assigns one work-item per work-group in CPUs.

Generally, it is shown that OpenCL cannot guarantee performance portability, as the architectural philosophy gap between CPUs and GPUs is huge and an optimal implementation for a CPU performs very badly on GPUs. As an indicative example, for most of the algorithms tested, the Intel optimized implementations only launch one thread per Compute Unit, which results in heavy under-utilized Compute Units on the GPUs.

Finally, in one of the most promising works that was published recently [7], the performance of OpenCL programs is evaluated on out-of-order multi-core CPUs from the architectural perspective. In this paper, the authors evaluated various aspects of OpenCL programs, including scheduling overhead, instruction-level parallelism, data location, locality and vectorization, comparing OpenCL to conventional parallel programming models for CPUs such as OpenMP. Their study was performed on two different architectures (Intel(R) Xeon (R) CPU E5645, NVIDIA GeForce GTX 580) and for the experiments they used simple applications and Parboil benchmarks [8].

Key findings of their evaluation are that opting for a large work-group size and coarsening the granularity of work per work-item are helpful for better performance on CPUs, because each one of them causes the reduction of the total number of work-groups which entails reduction of total context-switches. Coarsening GPUs, this survey verifies that as expected, they have different optimization requirements compared with CPUs, as they need many work-groups to exploit the vast parallelism execution opportunities that their hardware offers. However, a small work-group size is also bad on GPUs, since it makes GPUs unable to launch many warps within a streaming multiprocessor, thus minimizing the opportunities to overlap computation with data accesses.

Moreover, they observed that the large ILP value helps performance on CPUs. For example, in the case of ILP = 1, the next instruction depends on the output of the previous instruction; but in the case of ILP = 2, there is an independent instruction between two dependent instructions. They showed that an increasing ILP benefits the CPUs, while it keeps the performance in about the same levels for GPUs. Another important finding is that adding affinity support to OpenCL may help performance in some cases, due to the fact that this can lead to fewer cache misses on the private caches of CPU processors. Last but not least, this survey shows that vectorization can improve the CPU performance, as the vectorized code would result to less thread creation compared to non-

vectorized code, since SIMD instructions can perform computation on more than one data item at the same time.

Generally, this paper shows that since OpenCL has the same background as CUDA, most OpenCL applications are written to better utilize thread level parallelism (TLP), which is a scheme that cannot be applied on CPUs since even when the TLP of the application is large, the physical TLP available on CPUs is limited by the number of CPU cores, so that the context switching overhead is much higher on CPUs than on GPUs, for which this overhead is negligible. However, considering the characteristics of CPU architectures, the OpenCL application can be optimized further for CPUs, and the programmer needs to consider these insights for portable performance.

Chapter 6

Analysis of optimizations on dwarf benchmarks

In this section we tested the optimizations *Geometry* (*Granularity*, *Geometry of work-items*), *Vectorization* (*Vector Types*), *Loops* (*Loop Unrolling*) and *Branches* (*Padding*) and the OpenCL Berkeley dwarfs on which they are evaluated are the applications *lud*, *crc*, *needle*, *srad* and *bfs*. Moreover, some important things referred about the cache memory and the cache lines, in the applications that the latter affect the performance. This happens in kernels that make use of local memory and we tried to find which computational patterns offer more cache hits and how data should be organized in memory for better performance.

Geometry was the first optimization we tested. We separated it in two categories: granularity and Geometry of work-items. Testing granularity leads to the reduction of the number of work-items, but to the increment of the workload in each of them. This is expected to improve the performance of the CPU, because they are designed to execute many instructions per work-item and reduces the number of context switches, but it can be proved catastrophic for the GPUs. Changing the Geometry of work-items means that we keep the same number of the total work-items as the initial code, but the size of each work-group or grid is different. It is important to remember here, that except for the cases that our kernel is called iteratively, the work per work-item remains the same.

Vectorization is supported by architectures with vector type units such as the Intel CPU and the AMD GPU. The NVIDIA GPU can sometimes execute some instructions vectorized because of the double issue. In order to use vectorization, the data must be allocated on a consecutive memory area. We expect that if the vectorization can be applied on an application, it will give a speedup because it lets two or more instructions to be executed concurrently in the same clock cycle.

The loop unrolling is one of the most popular optimizations. It is supposed that can boost the performance because it can hide the cache misses, leads to better use of registers, or it can re-schedule the instructions in the loop. Different steps in loop unrolling may give a totally different performance. Moreover, AMD GPU and Intel CPU are expected to take advantage of their ILP metric that support.

Last but not least, we inspected how the branches affect the performance. Branches can kill the performance of a GPU because of the divergencies but it can also affect the CPU. We tried to eliminate them to see the difference of the performance on the code.

We must also mention that through an experimental study that we done in a set of our benchmarks, we found that the work-groups execute per core in Intel CPU and the work-items within them execute sequentially. Namely, each work-group is assigned to a core of Intel CPU and when it finishes its execution, a context switch happens in order another one to take place in this core.

6.1 lud

In LU Decomposition, a matrix is decomposed into a product of two matrices, a lower triangular and upper triangular matrix. Application *lud* belongs to the *dense linear algebra* category of dwarfs and comprises three kernels, the *lud_diagonal*, *lud_perimeter* and *lud_internal*. The most time-consuming kernel is the *lud_internal* and the *lud_perimeter* follows. The time of *lud_diagonal* is negligible and this kernel does not be considered in the analysis and implementation of this thesis.

6.1.1 Analysis of *lud_perimeter* kernel

6.1.1.1 Data Dependencies

In *lud_perimeter* kernel half of the work-items of each work-group fill the local, per work-group *peri_row* array, while the others fill the local, per work-group *peri_col* array. Every work-item in both computations follows a form of computation which is similar to the *prefix_sum* pattern. Namely, in every consecutive iteration more calculations are done per element per work-item compared to the previous iteration. *Figure 6.1* shows the way that the calculations are performed in this kernel.

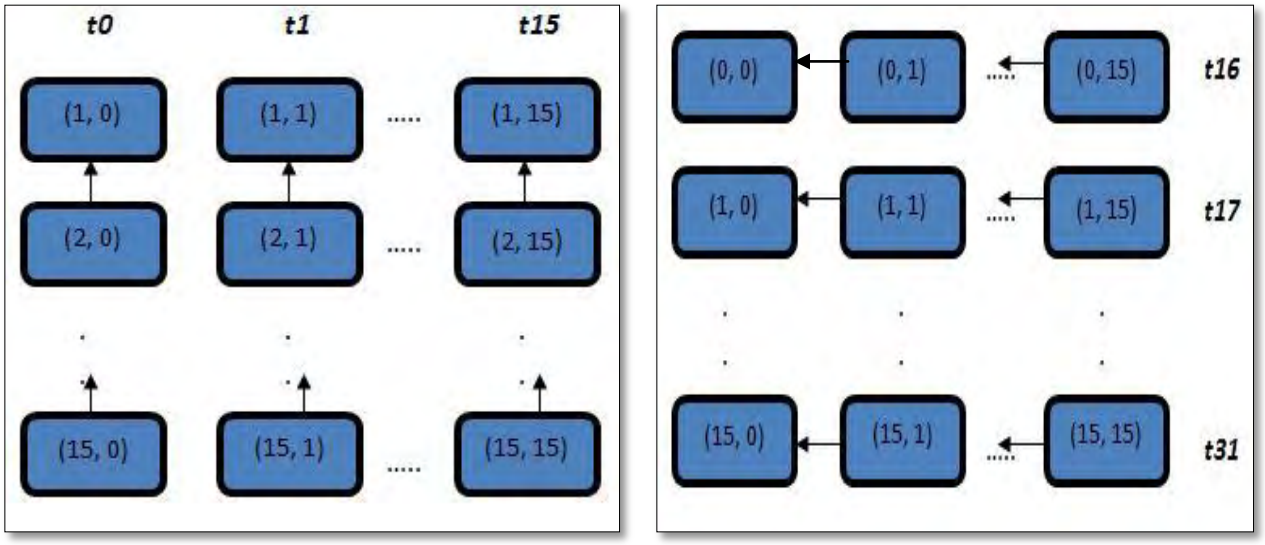


Figure 6.1: *The first image depicts the computation data dependencies of local, per work-group peri_row array, where the second image depicts the respectives for local, per work-group peri_col array*

The local, per work-group *dia* array is used only for reading. Moreover, the BLOCK_SIZE that is used in the original version of the code is 16 (as verified by the *Figure 6.1*), but the work-group size is 32 (2 * BLOCK_SIZE), so 16 work-items correspond to *peri_row* and 16 work-items correspond to *peri_col*. The BLOCK_SIZE here is the TILE_SIZE of *local memory.*

The arrays *peri_row* and *peri_col* are 2-dimensional BLOCK_SIZE*BLOCK_SIZE size local, per work-group arrays. As can be inferred from the *Figure 6.1*, work-items 0 to 15 of *peri_row* have vertical data dependencies on their private calculation data and as they don't have any dependencies among them, they are executed in parallel in each iteration. On the other hand, work-items 16 to 31 of *peri_col* have horizontal data dependencies on their atomic computations and they are also executed in parallel in each iteration, since they do not need any synchronization points among them. Furthermore, we must mention that in *Figure 6.1*, the first row of *peri_row* doesn't be included because it doesn't participate in computations.

Last but not least, we must mention that the *lud_perimeter* kernel is executed iteratively. The number of work-groups decreases by one in each consecutive iteration and the work-group size remains constant in all kernel calls.

6.1.1.2 Basic code segment

The time-consuming code segment of *lud_perimeter* kernel on which our optimizations are applied, presented here in order to help one to realize better the form of communication and computation that exists.

```
If ( get_local_id(0) < BLOCK_SIZE ) {
   idx = get_local_id(0);

   for (i = 1, i < BLOCK_SIZE; i++) {
      for (j = 0; j < i; j++)
         peri_row[i][idx] -= dia[i][j] * peri_row[j][idx];
   }

}
else {
   idx = get_local_id(0) – BLOCK_SIZE;

   for (i = 0; i < BLOCK_SIZE; i++) {
      for (j = 0; j < i; j++)
         peri_col[idx][i] -= peri_col[idx][j] * dia[j][i];
      peri_col[idx][i] /= dia[i][i];
   }
}
```

6.1.1.3 Optimization Efforts and Results

1) Execution Geometry

a) *Granularity*

2x, 4x, 8x and 16x coarsening of the workload per work-item is tested.
Larger coarsening per work-item is better for the Intel CPU, whereas it proves catastrophic for GPUs. In kernel *lud_perimeter*, *2x coarsening* means that the work-group size reduces from 32 to 16, while with *4x coarsening* it reduces from 32 to 8 and so on. Therefore, the work per work-item is gradually bigger along work-group size reduction.

Specifically, we use the *Figure 6.1* to explain it further. We choose the *4x coarsening* case for that and the induction to other cases is similar. We have explained previously how the data dependencies of *peri_row* and *peri_col* arrays are per work-item. In *4x coarsening* case, the work-group size reduces from 32 to 8 and this means that 4 work-items per work-group are responsible for the computation of *peri_row* and the other 4 are responsible for the computation of *peri_col*. Namely, the work-item 0 of *peri_row* array will get the summed work of work-items 0 to 3 of initial implementation, work-item 1 will get the summed work of work-items 4 to 7 and so on. In other words, in initial implementation during the first iteration, work-item 0 computes the element peri_row[1][0], work-item 1 computes the element peri_row[1][1] and so on. Instead, in *4x coarsening* case during the first iteration, work-item 0 computes the elements peri_row[1][0] to peri_row[1][3], work-item 1 computes the elements peri_row[1][4] to peri_row[1][7] and so on. Therefore, it is obvious that the coarsening per work-item in *4x coarsening* case, as related to *peri_row* array, increases by 4 compared to initial implementation.

Additionally, similar is the situation for the computation of *peri_col* array. However, the thing that deserves to mention here is the following. As the *Granularity* optimization leads usually to a form of data that it is feasible to be vectorized due to the consecutive memory locations that the data have, it is good when one applies that to take that into consideration. Observing the second image of *Figure 6.1*, that of *peri_col* case, we see that the coarsening here can be implemented per column and not per row, as in the case that happens in *peri_row* array. This holds because the data dependencies per work-item have vertical form for *peri_row* case, while they have horizontal form for *peri_col* case. The problem is that the consecutive data per column cannot be vectorized, since they do not belong to successive memory locations. For example peri_col[0][0] and peri_col[1][0] elements belong to different memory locations. A solution to this is the transition of *peri_col* array that gives the same computational pattern that *peri_row* array has and the things that explained previously can exactly be applied in this case. Therefore, the coarsening per work-item is the same for *peri_row* and *peri_col* local, per work-group arrays and the *Vectorization* optimization (we will refer to that later in this section) can very effectively be implemented.

Figure 6.2 and *Figure 6.3* depict the results for matrix dimension 4096 and 8192.

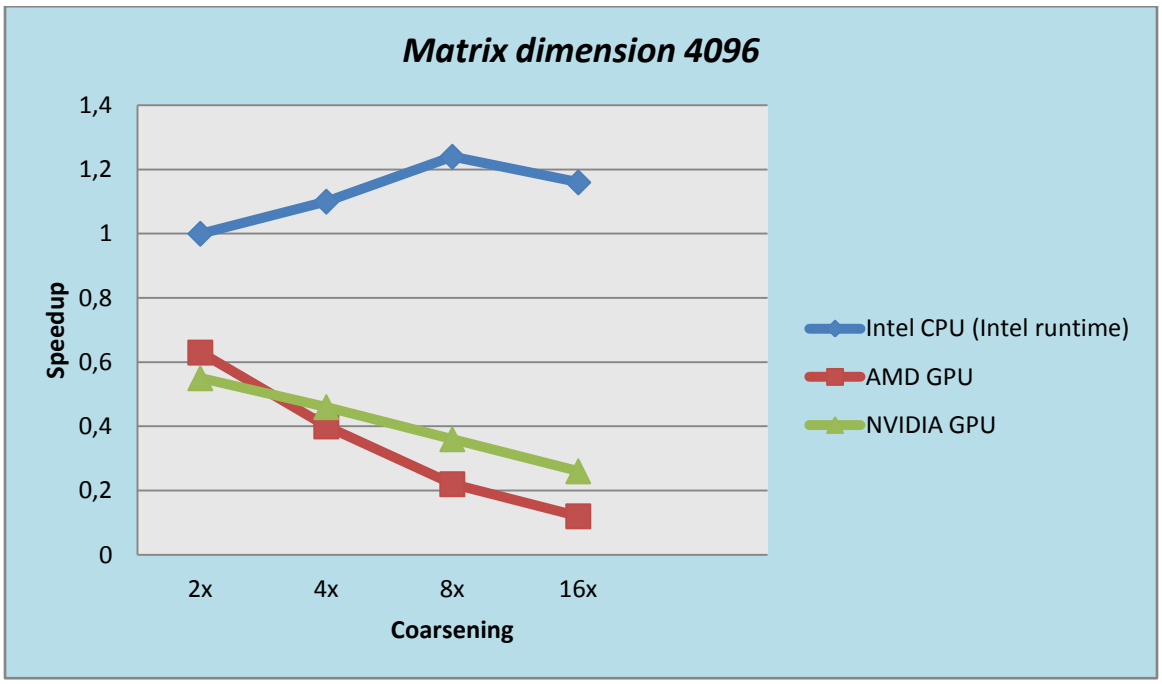
Optimizations / Devices (4096)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	63,321,554 ns	110,547,551 ns	30,023,648 ns
2x coarsening	60,574,101 ns	174,268,240 ns	55,017,120 ns
4x coarsening	57,433,163 ns	279,506,877 ns	65,330,656 ns
8x coarsening	51,001,526 ns	495,125,112 ns	83,164,480 ns
16x coarsening	54,496,953 ns	925,336,786 ns	116,684,256 ns

Optimizations / Devices (4096)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
2x coarsening	1	0.63	0.55
4x coarsening	1.10	0.40	0.46
8x coarsening	1.24	0.22	0.36
16x coarsening	1.16	0.12	0.26

Optimizations / Devices (8192)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	352,030,387 ns	410,034,996 ns	100,092,608 ns
2x coarsening	250,110,372 ns	653,036,769 ns	183,355,968 ns
4x coarsening	221,658,341 ns	1,065,818,329 ns	218,478,112 ns
8x coarsening	196,498,045 ns	1,901,088,454 ns	278,167,008 ns
16x coarsening	203,916,734 ns	3,571,183,673 ns	391,285,888 ns

Optimizations / Devices (8192)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
2x coarsening	1.40	0.63	0.55
4x coarsening	1.59	0.38	0.46
8x coarsening	1.79	0.22	0.36
16x coarsening	1.73	0.11	0.26

Figure 6.2: *Execution times and speedup of optimization 'Granularity' in lud_perimeter kernel for matrix dimension 4096 and 8192*



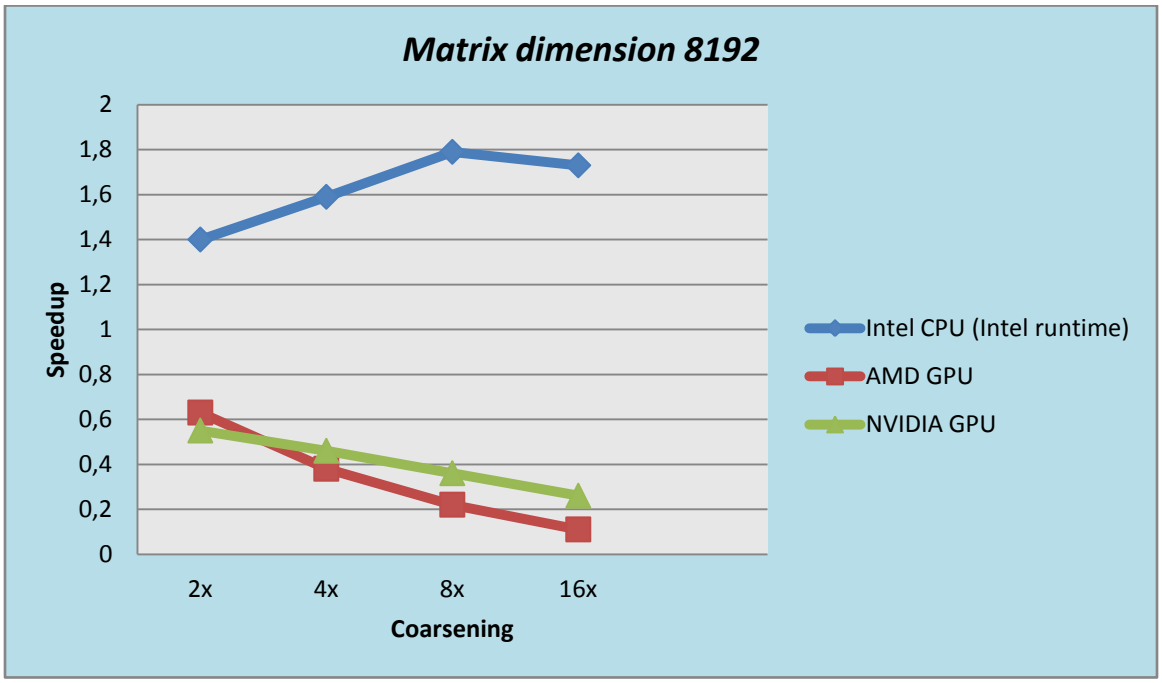


Figure 6.3: *Speedup graphical representation of optimization 'Granularity' in lud_perimeter kernel for matrix dimension 4096 and 8192*

As can be inferred from the *Figure 6.2*, both GPUs perform gradually poorer along the increase of coarsening and equally for both matrix dimensions. This can be explained via the fact that gradually more hardware units remain unexploited. The total work-groups per kernel are the same, compared to the initial implementation, and independent from the increase of coarsening per work-item, but the number of work-items inside them reduces and this results to a degradation in the parallelism

that the warp/wavefront execution unit of NVIDIA/AMD can exploit.

An indicative example that explains that, is the case of *16x coarsening* of AMD GPU, where the total number of wavefronts in the first call of *lud_perimeter* kernel for matrix dimension 4096 is 255 and remains the same as in the initial implementation. However, each wavefront of initial code utilizes the half of the work-items that includes, whereas in *16x coarsening* case, the wavefront utilization drops to 2 work-items. This happens as in the first case there are 255 work-groups of 32 work-items, while in the second case there are 255 work-groups of 2 work-items.

On the other hand, coarsening the workload of work-items is beneficial for the Intel CPU, because the total number of work-items reduces and this results to fewer context-switches. Context switches are associated with significant overhead on CPUs due to the operating system intervention. The Intel CPU performance is gradually better along higher coarsening and its maximum is achieved with *8x coarsening*, resulting to 4 work-items per work-group. Moreover, it is worth to mention that the speedup over the CPU execution is, as expected, much better for matrix dimension 8192, since the overhead of context switches is higher, as much more kernel calls are done and this results to a higher number of work-groups to be handled.

b) *Geometry of work-items*

In this category, two alternative BLOCK_SIZE values, namely 32 and 64 are evaluated in the *lud_perimeter* kernel. We remind that the case of BLOCK_SIZE = 16 corresponds to the initial implementation. However, only the case of BLOCK_SIZE = 32 it was feasible to implement for the Intel CPU and the NVIDIA GPU. BLOCK_SIZE = 32 it was unfeasible to be evaluated for the AMD GPU due to the maximum work-group size limitations of *lud_internal* kernel. Furthermore, the same reason explains why BLOCK_SIZE = 64 operates as a deterrent for all the architectures.

The *lud_internal* kernel uses 2-dimensional BS*BS size work-groups and BS = 32 yields work-groups of size 1024, while BS = 64 yields work-groups of size 4096. The maximum work-group size for the AMD GPU is 256 work-items, while this value is 1024 for the Intel CPU and the NVIDIA GPU. The *lud_perimeter* kernel belongs to the same program as *lud_internal* kernel does and the BLOCK_SIZE parameter is common among them. As can be inferred, this means that the execution parameters of one kernel affect the other. If *lud_perimeter* kernel was executed separately, the BS = 128 would be the maximum value to be evaluated for all the architectures, since this kernel uses work-groups of 2*BLOCK_SIZE size.

The results of this optimization effort are not good for all the devices that used due to the *prefix_sum* form of the computation. With the increase of BLOCK_SIZE in *lud_perimeter* kernel, the total number of work-groups reduces, the number of work-items inside them increases, but the work per work-item increases geometrically for the computation of local, per work-group arrays *peri_row* and *peri_col*. With the term geometrically we mean that in every consecutive loop iteration during the computation of *peri_row* and *peri_col*, the workload per work-item doubles compared to the one that each of them has in the previous iteration. Moreover, taking into account that the arrays *peri_row* and *peri_col* are 2-dimensional of BS*BS size, an increase in BLOCK_SIZE leads to a higher degree of

geometric increase of afore-mentioned computations, as more elements in both local, per work-group arrays must be computed per work-item. In short, the work per work-item gets significantly more compared with the initial kernel implementation.

The reason that Intel CPU performance degrades less than that of NVIDIA GPU during the BLOCK_SIZE change is the fact that Intel compiler implements a form of implicit vectorization in *peri_row* array and can execute general-purpose code, such as *prefix-sum* pattern in this kernel, faster per work-item. Namely, the much more added workload per work-item during the change of BLOCK_SIZE from 16 to 32, affects less the performance of CPU work-items, as they have the ability to execute faster a sequential code, like the *prefix-sum* per work-item pattern of data dependencies, in our case. Furthermore, the reduction of total work-groups entails fewer context-switches for Intel CPU and less parallelism for NVIDIA GPU.

Figure 6.4 and *Figure 6.5* depict the results for matrix dimension 4096 and 8192.

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	63,321,554 ns	110,547,551 ns	30,023,648 ns
BS 16 to 32	130,281,197 ns	-----	164,805,536 ns

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
BS 16 to 32	0.49	-----	0.18

Optimizations / Devices (8192)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	352,030,387 ns	410,034,996 ns	100,092,608 ns
BS 16 to 32	521,613,660 ns	-----	554,673,600 ns

Optimizations / Devices (8192)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
BS 16 to 32	0.67	-----	0.18

Figure 6.4: *Execution times and speedup of optimization 'Geometry of work-items' in lud_perimeter kernel for matrix dimension 4096 and 8192*

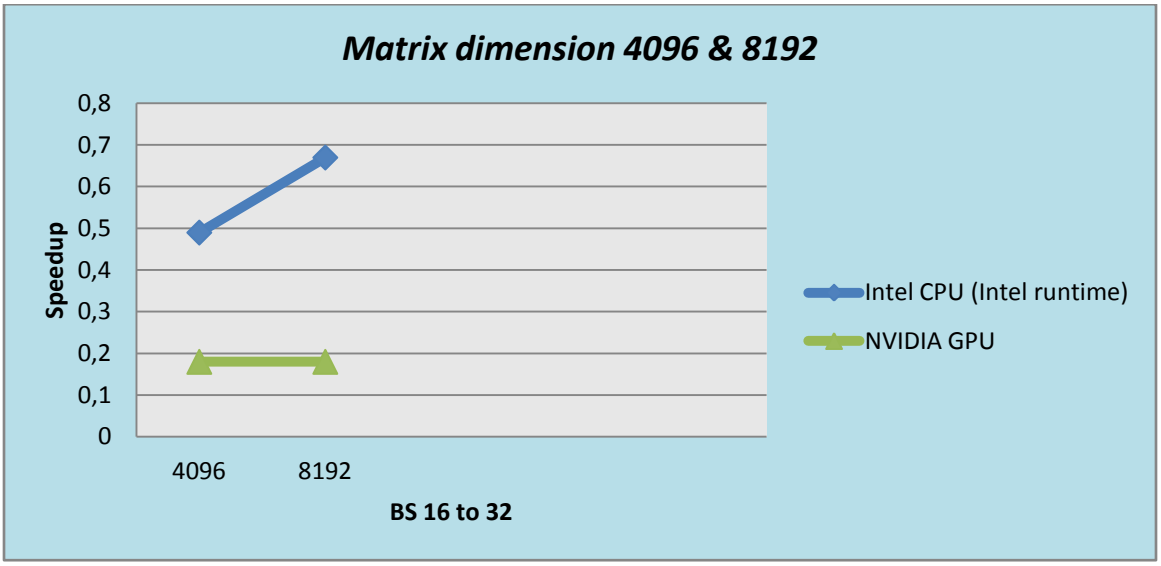


Figure 6.5: *Speedup graphical representation of optimization 'Geometry of work-items' in lud_perimeter kernel for matrix dimension 4096 and 8192*

2) Vectorization

Vectorization is tested in this kernel on code that resulted after the *Granularity* optimization.

Vector Types

The way that shows how the consecutive memory locations of *peri_row* and *peri_col* data elements are induced from the initial code, was analyzed previously in *Granularity* optimization sub-section. The point is that the afore-mentioned analysis makes feasible and effective the implementation of vectorization on these data. The degree of vectorization that can be applied ranges from 2 (*2x vectorization* case) to 16 (*16x vectorization* case). The value of 16 is the upper limit of vectorization, as the matrix dimensions of both local, per work-group arrays *peri_row* and *peri_col* is 16. Moreover, we must mention that the local array *dia* it was not effective to be vectorized, since its data pattern can't yield consecutive memory locations and the use of vectorization to this array would be useless and ineffective.

Vectorization proves very effective for the Intel CPU and the degrees of 4 and 8 are the ones that achieve the best performance. The vectorization is not considered for GPUs, since the form of code that the *Granularity* optimization induces in *lud_perimeter* kernel is very unsuitable for them, as it was explained previously in the relative section. Namely, as the coarsening of workload per work-item increases, less work-items per work-group are used and this results to more and more work-items within a warp/wavefront for NVIDIA/AMD GPU to be inactive and useless. In other words, the degree of underutilization of a warp/wavefront gets gradually higher with the increase of coarsening per-work item.

Figure 6.6 and *Figure 6.7* depict the results for matrix dimension 4096 and 8192.

Optimizations / Devices (4096)	Intel CPU (Intel runtime)
Initial Code	63,321,554 ns
2x vectorization	48,461,563 ns
4x vectorization	31,535,570 ns
8x vectorization	28,353,160 ns
16x vectorization	44,411,482 ns

Optimizations / Devices (4096)	Intel CPU (Intel runtime)
2x vectorization	1.31
4x vectorization	2.00
8x vectorization	2.23
16x vectorization	1.42

Optimizations / Devices (8192)	Intel CPU (Intel runtime)
Initial Code	352,030,387 ns
2x vectorization	173,823,526 ns
4x vectorization	112,223,109 ns
8x vectorization	115,138,060 ns
16x vectorization	228,075,795 ns

Optimizations / Devices (8192)	Intel CPU (Intel runtime)
2x vectorization	2.00
4x vectorization	3.14
8x vectorization	3.00
16x vectorization	1.54

Figure 6.6: *Execution times and speedup of optimization 'Vectorization' in lud_perimeter kernel for matrix dimension 4096 and 8192*

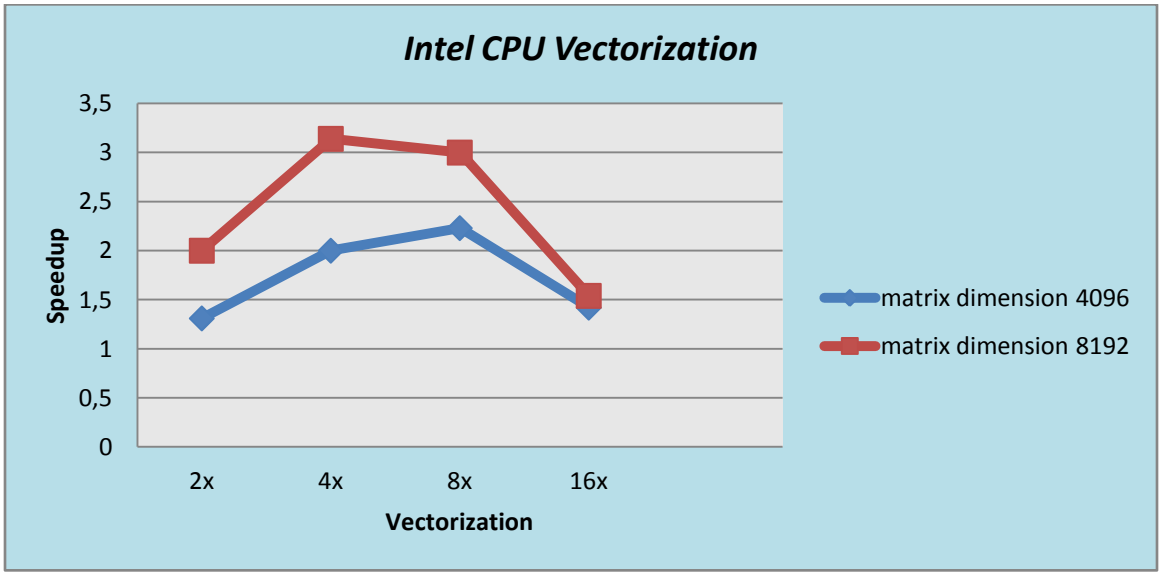


Figure 6.7: *Speedup graphical representation of optimization 'Vectorization' in lud_perimeter kernel for matrix dimension 4096 and 8192*

As can be deduced from the *Figure 6.6*, the effect of vectorization is greater for matrix dimension 8192 compared to that of 4096 and for both matrix dimensions is gradually bigger except for the case of *16x vectorization*, where the performance falls. The reason for that is that Intel CPU recommends for float data (the type of data that kernel *lud_perimeter* uses) vector width with degree of 4. Therefore, a big increase of this width can cause an inappropriate use of vector types, since many of the operations that exceed the preferred vector width are done serially and the parallelism among threads reduces. This is the reason also why *4x vectorization* and *8x vectorization* yields the best performance which fluctuates in the same levels for both matrix dimensions that used.

3) Loops

Loop Unrolling

In this category, three optimization parameters are examined.

- **Case_1**: Implementation of full loop unrolling in all the outer loops of kernel and implementation of loop unrolling with step 2 in the even inner loops of kernel
- **Case_2**: Implementation of full loop unrolling in all the outer loops of kernel and implementation of full loop unrolling in the even inner loops of kernel
- **Case_3**: Implementation of full loop unrolling in all the outer loops of kernel and implementation of full loop unrolling in all the inner loops of kernel

The result is that loop unrolling generally improves the performance of GPUs, while the performance of Intel CPU remains constant in all cases in this kernel. Namely, AMD GPU performs better compared to initial code and as move from one case to another for matrix dimension 4096, whereas for matrix dimension 8192 degrades a little the performance in last case compared to its previous case. On the other side, NVIDIA GPU performs best in the last case and gradually better along cases for both matrix dimensions that used. What's more, the speedup of NVIDIA GPU is the same for both matrix dimensions, whereas the speedup of AMD GPU is much better for matrix dimension 4096 due to the fewer kernel calls that done, as explained further below.

Figure 6.8 and *Figure 6.9* depict the results for matrix dimension 4096 and 8192.

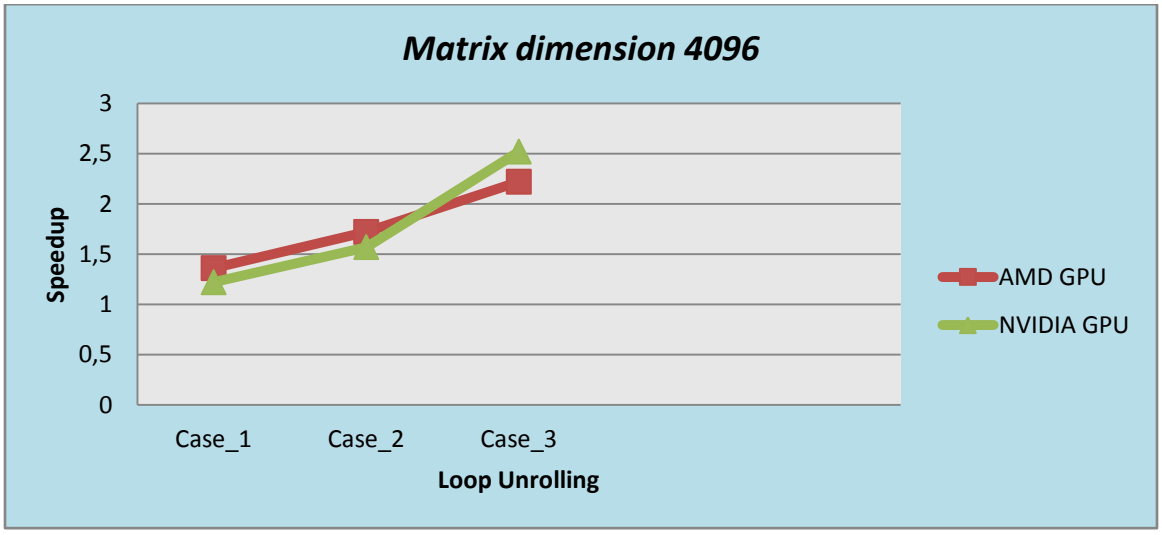
Optimizations / Devices (4096)	**AMD GPU**	**NVIDIA GPU**
Initial Code	110,547,551 ns	30,023,648 ns
Case_1	81,375,458 ns	24,675,360 ns
Case_2	64,221,663 ns	19,134,592 ns
Case_3	49,767,109 ns	11,906,592 ns

Optimizations / Devices (4096)	AMD GPU	NVIDIA GPU
Case_1	1.36	1.22
Case_2	1.72	1.57
Case_3	2.22	2.52

Optimizations / Devices (8192)	AMD GPU	NVIDIA GPU
Initial Code	410,034,996 ns	100,092,608 ns
Case_1	307,785,108 ns	81,931,456 ns
Case_2	266,473,002 ns	63,210,848 ns
Case_3	287,662,116 ns	39,366,848 ns

Optimizations / Devices (8192)	AMD GPU	NVIDIA GPU
Case_1	1.33	1.22
Case_2	1.54	1.58
Case_3	1.43	2.54

Figure 6.8: *Execution times and speedup of optimization 'Loop Unrolling' in lud_perimeter kernel for matrix dimension 4096 and 8192*



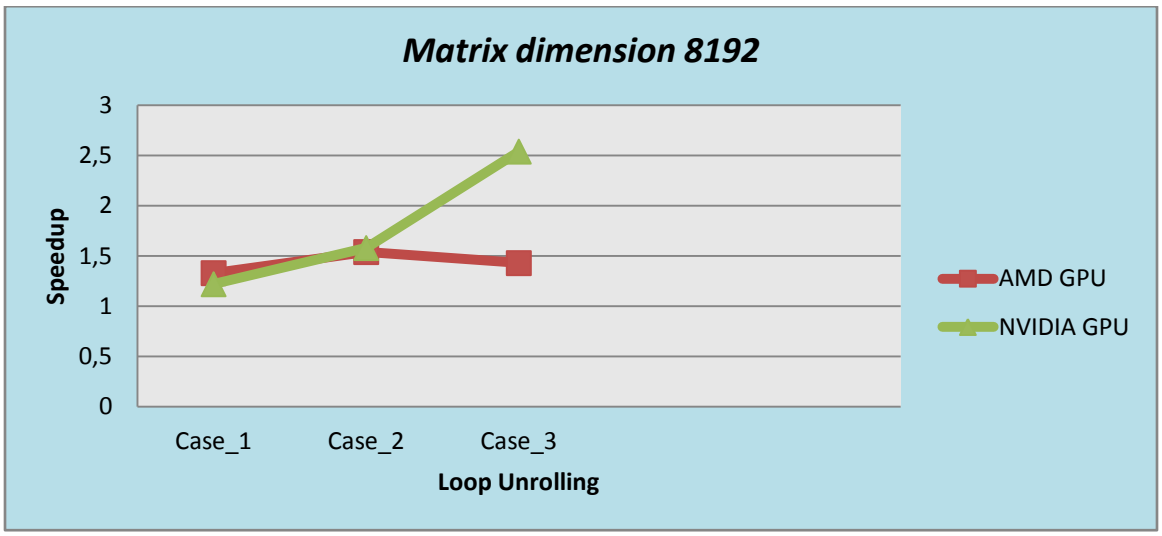


Figure 6.9: *Speedup graphical representation of optimization 'Loop Unrolling' in lud_perimeter kernel for matrix dimension 4096 and 8192*

The reason that loop unrolling proves so effective for AMD GPU is that its VLIW processors can reduce the average number of ALU instructions executed per work-item. Specifically, the ALU instructions executed per work-item in initial implementation are 2257, in *Case_1* are 1409, in *Case_2* are 1262 and in *Case_3* are 1110 for both matrix dimensions that used. Furthermore, the number of general purpose vector registers used by the kernel is 21, 12, 14 and 20 respectively.

The VLIW architecture takes advantage of ILP by executing multiple instructions in parallel, which means that it is capable of executing independent or dependent instructions at the same time. Dependent instructions can be executed in parallel due to the four execution pipelines of the 4-wide VLIW (VLIW4), which consists the simpler execution unit of AMD Cayman architecture. Loop unrolling in *lud_perimeter* kernel creates many consecutive dependent instructions, so the afore-mentioned pipelines of every VLIW4 of every SIMD core are exploited as the degree of unrolling increases. Specifically , these instructions are for array *peri_row*, *peri_row[i][idx] -= dia[i][j] * peri_row[j][idx]*, *peri_row[i][idx] -= dia[i][j+1] * peri_row[j+1][idx]*, *peri_row[i][idx] -= dia[i][j+2] * peri_row[j+2][idx]* and so on and for array *peri_col*, *peri_col[idx][i] -= peri_col[idx][j] * dia[j][i]*, *peri_col[idx][i] -= peri_col[idx][j+1] * dia[j+1][i]*, *peri_col[idx][i] -= peri_col[idx][j+2] * dia[j+2][i]* and so on.
Of course, we must mention that if we had independent instructions, the VLIW architecture would offer even higher parallelism as none of the data would need to be pipelined within and among VLIW4 execution units of AMD GPU.

Furthermore, the fact that the full loop unrolling that we implemented for matrix dimension 4096 in last case proves the best performing option for AMD GPU, can be explained observing that it helps the compiler to exploit the maximum of parallelism that ILP offers in kernel *lud_perimeter*. The deviation of speedup among matrix dimensions of 4096 and 8192 can be attributed to the increased number of kernel calls between them. Namely, taking into account that the number of ALU instructions for matrix dimension 4096 in each case is the same with the respective one of matrix

dimension 8192, we can infer that the overhead of times that *lud_perimeter* kernel is called is much higher than the benefit that the reduction of ALU instructions offers in each one of the kernel executions that done. We must mention that in the first call of kernel, for matrix dimension 4096, 255 work-groups are used with 32 work-items in each one of them, while for matrix dimension 8192 there are 511 work-groups of 32 work-items. Moreover, we remind that the *lud_perimeter* kernel is called iteratively with a decreasing by one number of work-groups in each consecutive iteration keeping constant the size of work-items in each one of them.

On the other hand, NVIDIA GPU does not have vector processors in its hardware, but as it can be seen from *Figure 6.8*, the loop unrolling optimization also benefits it and more than AMD, especially for matrix dimension 8192. This happens, as NVIDIA is able to exploit the vast register file that possesses. Therefore, as the degree of unrolling increases during the transition from one case to another, the performance gradually improves, since the cumulative register requirements by all work-items doesn't seem to exceed the capacity of the register file. Additionally, it is important to mention that there is no speedup deviation for NVIDIA GPU between the matrix dimensions 4096 and 8192, since the overhead in this case is not affected by the number of times that this kernel is called and the faster execution times that NVIDIA GPU achieves compared to AMD GPU for both matrix dimensions that used, as can be verified by *Figure 6.8*, can justify that.

Last but not least, it is worth to note that the execution on the Intel CPU is not affected by *Loop Unrolling* optimizations. This happens because in *lud_perimeter* kernel there aren't any independent instructions between dependent ones for both computations of arrays *peri_row* and *peri_col*, and this results to the CPU ILP metric be unexploited.

6.1.2 Analysis of *lud_internal* kernel

6.1.2.1 Data Dependencies

This kernel calculates the inner product of BS elements of a column with BS elements of a row of an array M and stores the result in the corresponding position in the M. At first, two matrices are allocated in the local memory with size BSxBS. These matrices are named *peri_row* and *peri_col*. Then each work-item of each work-group is responsible for storing in these matrices the corresponding element from the matrix M. When all the work-items of a work-group have stored in their elements to the matrices, the computations begin as it is shown in *Figure 6.10*. Each work-item of a work-group is responsible for calculating the value of an element by calculating the inner product of the corresponding row of the *peri_col* with the corresponding column of the *peri_row* and storing it in the array of the global memory. In our example we suppose that the BS is 4 and the size of the array is 8x8.

The algorithm uses 2-dimensional work-groups. The work-groups are BSxBS and the original BS is 16:

localWorkSize[0] = BLOCK_SIZE;
localWorkSize[1] = BLOCK_SIZE;

The total number of the work-items differs in each kernel invocation. The first iteration has the most work-items and their number decreases in each iteration because there are fewer inner products to be calculated. The kernel is invoked one time in each iteration of a for-loop. In each iteration the step *i* of the loop is increased by BS and the total work-items are calculated by the following formula, where it is obvious that the lower bound of globalWorkSize depends on the matrix_dim and the BS. Different values in these variables lead to a different lower bound.

*globalWorkSize[0] = ((matrix_dim-i)/BLOCK_SIZE-1)*localWorkSize[0];*
*globalWorkSize[1] = ((matrix_dim-i)/BLOCK_SIZE-1)*localWorkSize[1];*

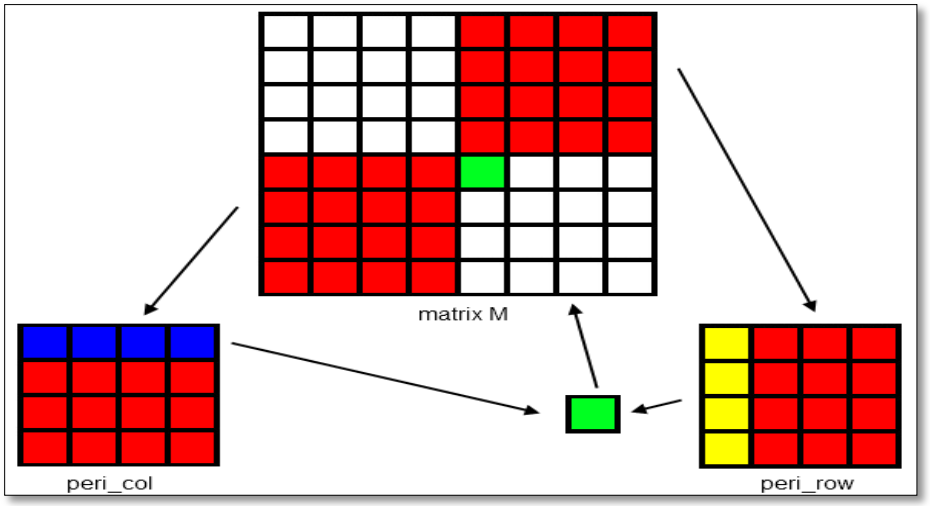


Figure 6.10: Pattern of lud_internal kernel

6.1.2.2 Basic code segment

We present the code of the stores to the local memory from the global memory and the computation of the inner product of each element and the store of it to the global memory.

*global_row_id = offset + (get_group_id(1)+1)*BLOCK_SIZE;*
*global_col_id = offset + (get_group_id(0)+1)*BLOCK_SIZE;*

```
peri_row[get_local_id(1)][get_local_id(0)] =
m[(offset+get_local_id(1))*matrix_dim+global_col_id+get_local_id(0)];

peri_col[get_local_id(1)][get_local_id(0)] =
m[(global_row_id+get_local_id(1))*matrix_dim+offset+get_local_id(0)];

barrier(CLK_LOCAL_MEM_FENCE);

sum = 0;
for (i=0; i < BLOCK_SIZE; i++)
   sum += peri_col[get_local_id(1)][i] * peri_row[i][get_local_id(0)];
m[(global_row_id+get_local_id(1))*matrix_dim+global_col_id+get_local_id(0)] -= sum;
```

6.1.2.3 Optimization Efforts and Results

1) Execution Geometry

a) *Granularity*

Choosing the total number of the work-items and the workload of each of them is crucial for optimizing an application. We made experiments with coarsening 2x. That means that the work-items in each work-group were decreased from 256 to 128. Also each work-item was responsible for storing two elements in each matrix in the local memory, taking them from the global memory, instead of one and calculating two inner products instead of one.

The CPUs are designed to execute more efficiently many instructions in one work-item than a few instructions in many work-items. On the other hand GPUs prefer to execute only a few instructions per work-item. We tested scheduling 2x, 4x and 8x workload per work-item in two different ways.

1st way:

In this scenario we divided the *localWorkSize[0]* by a factor of 2, 4 and 8. This means that the work-items in *local_id(0)* had to do 2x, 4x or 8x workload as shown in *Figure 6.11*. The work-items in *local_id(0)* are responsible for the *peri_row* matrix in the loop. Increasing their workload, each work-item gets two elements of the matrix instead of one. But these two elements are BS elements apart.

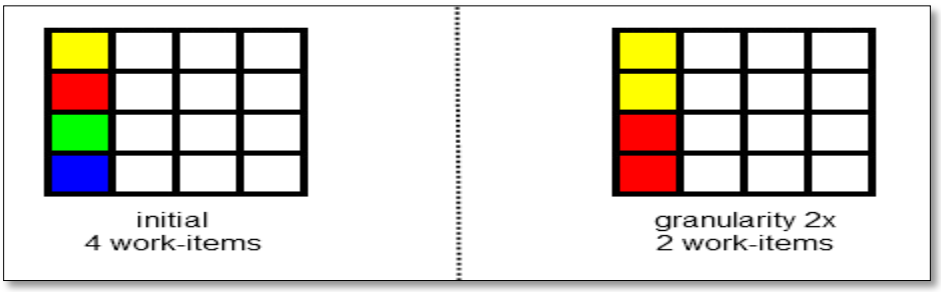


Figure 6.11: *granularity 2x on get_local_id(0)*

Accessing two elements that are BS positions apart did not work for the CPU because a CPU prefers to access adjacent elements, because of the locality in a cache line. Although a CPU may prefer to have more workload per work-item, accessing the matrix in such a way gave a worse execution time than the initial. The NVIDIA GPU had a slightly worse execution time, as expected, but the difference was too small. GPUs are designed for fine grain parallelism and coarsening the parallelism may decrease their performance unless there is a reduction in their utilization.

On the other hand, the AMD GPU had a small speedup. The number of ALU instructions was increased per work-item from 40 to 60. However, the total number of work-items was decreased by a factor of 2. So there were less ALU instructions in total from the initial code. Initially we had 16,646,400 work-items executing 40 ALU instructions each, meaning there were 665,856,000 ALU instructions executed totally. Decreasing the work-items to 8,323,200, the ALU instructions executed per work-item increased to 60, but there were 499,392,000 executed totally. Also with the use of the VGPRs, the vector type registers were increased from 5 to 8.

Testing a bigger coarsening, the performance was the same for the Intel CPU but there was a great loss for the GPUs. This happened because of the higher rate of cache misses. More specifically the cache hit rate decreased from 26% to 13%.

Figure 6.12 and *Figure 6.13* depict the results for matrix dimension 4096 and 8192.

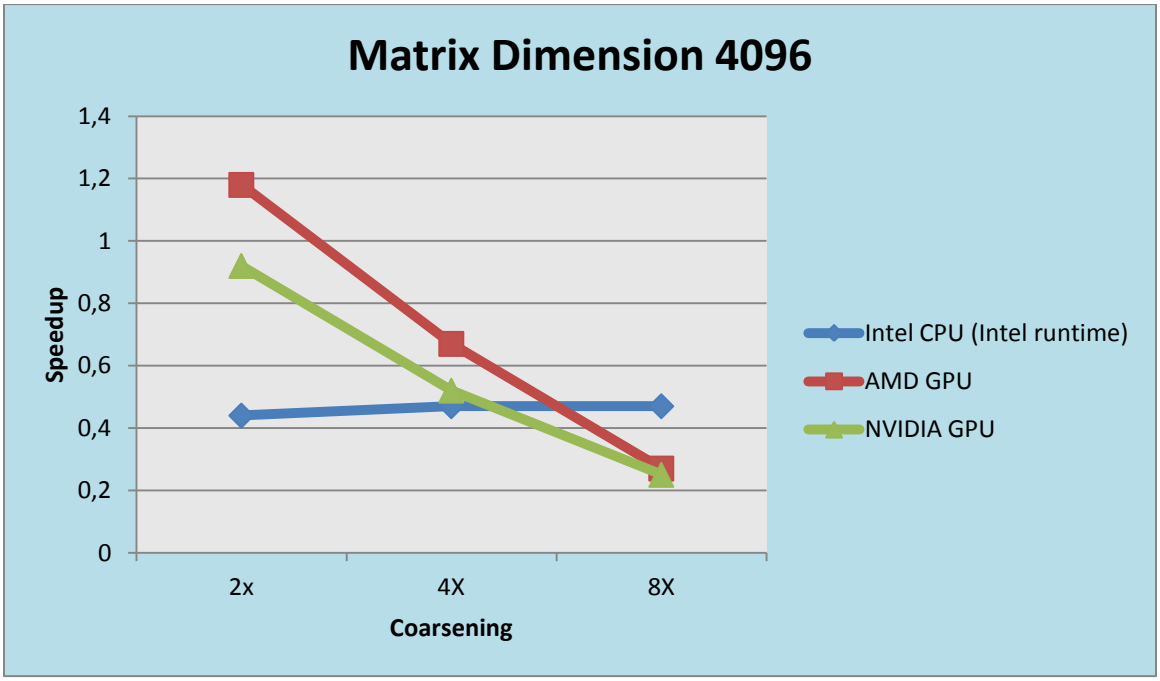
Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	1.99 sec	1 sec	0.23 sec
2x coarsening	4.5 sec	0.85 sec	0.25 sec
4x coarsening	4.2 sec	1.5 sec	0.44 sec
8x coarsening	4.2 sec	3.7 sec	0.91 sec

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
2x coarsening	0.44	1.18	0.92
4x coarsening	0.47	0.67	0.52
8x coarsening	0.47	0.27	0.25

Optimizations / Devices (8192)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	16.2 sec	7.68 sec	1.9 sec
2x coarsening	34.2 sec	7 sec	2 sec
4x coarsening	29.2 sec	15.78 sec	3.65 sec
8x coarsening	32.3 sec	36.43 sec	7.49 sec

Optimizations / Devices (8192)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
2x coarsening	0.47	1.10	0.95
4x coarsening	0.65	0.49	0.51
8x coarsening	0.5	0.21	0.25

Figure 6.12: *Execution times and speedup of optimization 'Granularity' in lud_internal kernel for matrix dimension 4096 and 8192*



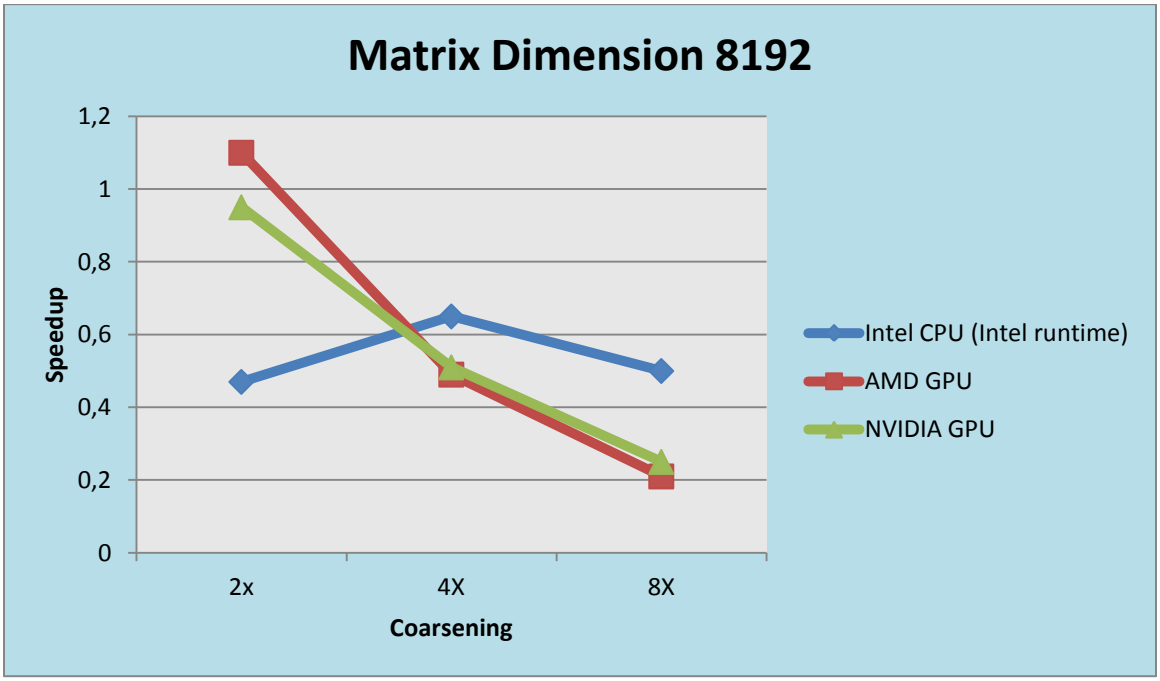


Figure 6.13: *Speedup graphical representation of optimization 'Granularity' in lud_internal kernel for matrix dimension 4096 and 8192*

2nd way:

In this case we divided the *localWorkSize[1]* by a factor of 2, 4 again. In this case the work-items in *local_id(1)* had to do 2x, 4x or 8x workload. These work-items were responsible for the *peri_col* matrix in the loop as shown in *Figure 6.14*. Increasing their workload, each work-item gets two elements of the matrix instead of one, but in this case the two elements are adjacent.

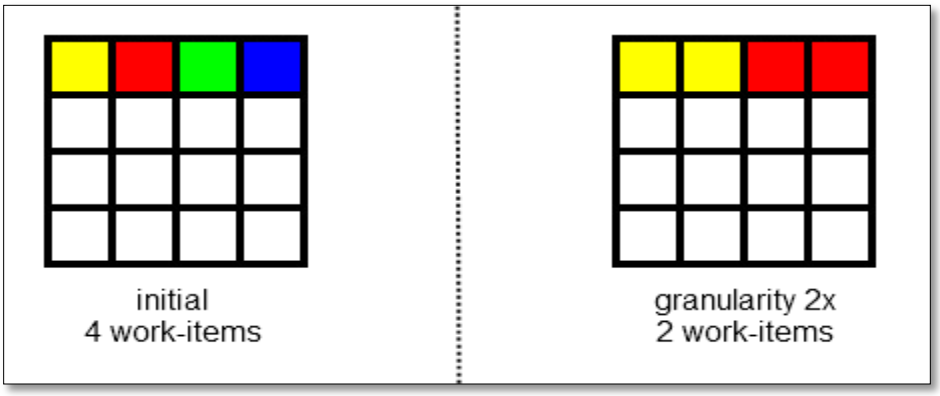


Figure 6.14: *granularity 2x on get_local_id(1)*

Testing the new implementation we observed that the CPU had a better execution time than the initial one. Accessing the matrix in a more correct way gave us better results. Also fewer work-

items on a CPU means fewer context switches which cost too much because they are controlled by the software. The NVIDIA's execution time was a little worse, as expected again, but showed us that this optimization did not affect it. The AMD GPU had the same results as previously. The number of ALU instructions was increased per work-item from 40 to 60 again, but the total number of the ALU instructions was smaller. The number of VGPRs was increased again from 5 to 8 too. Also the cache hit rate increased from 26% to 49%.

Testing coarsening 4x, we observed that all the architectures had the same execution time compared to coarsening 2x. The problem arose with the coarsening 8x for AMD GPU. The ALU instructions per work-item and the cache hit rate was the same again but the ALUPacking metric decreased from 78% to 67%. This is the ALU vector pack efficiency and indicates how well the Shader Compiler packs the scalar or vector ALU in the kernel. What is more, a value below 67% indicates that ALU dependency chains may be preventing full utilization of the processor.

Figure 6.15 and *Figure 6.16* depict the results for matrix dimension 4096 and 8192.

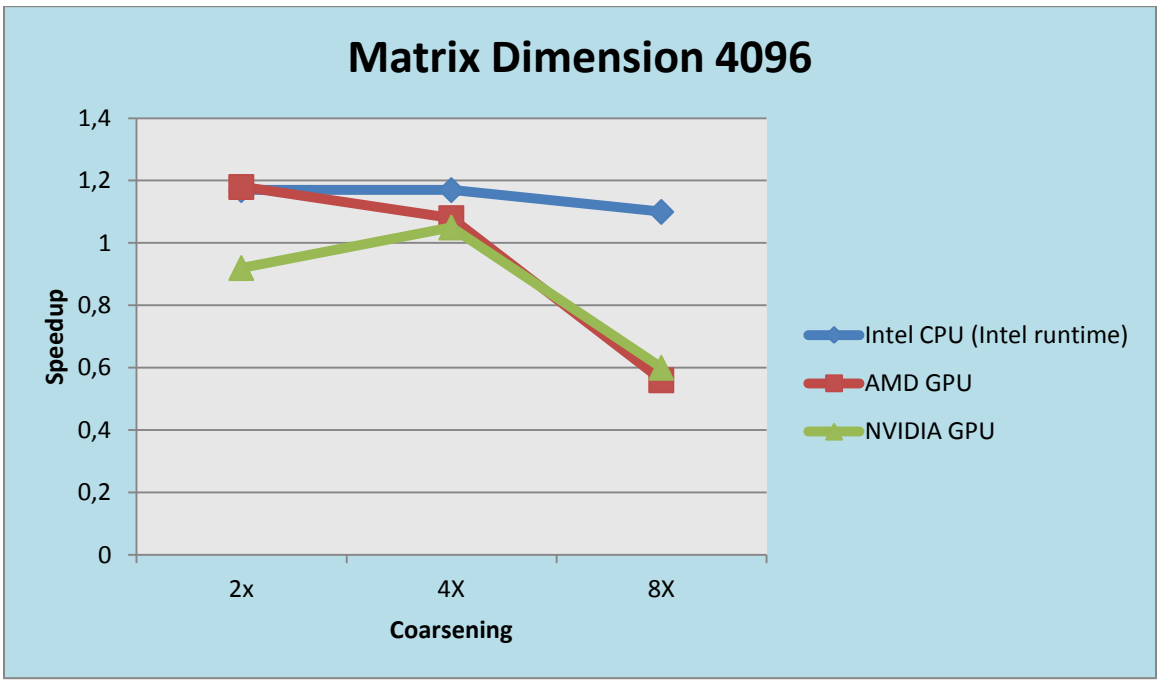
Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	1.99 sec	1 sec	0.23 sec
2x coarsening	1.7 sec	0.85 sec	0.25 sec
4x coarsening	1.7 sec	0.88 sec	0.22 sec
8x coarsening	1.8 sec	1.78 sec	0.28 sec

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
2x coarsening	1.17	1.18	0.92
4x coarsening	1.17	1.13	1.05
8x coarsening	1.1	0.56	0.82

Optimizations / Devices (8192)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	16.2 sec	7.68 sec	1.9 sec
2x coarsening	13.2 sec	7 sec	2 sec
4x coarsening	14.1 sec	9.22 sec	1.8 sec
8x coarsening	14.2 sec	15.1 sec	3 sec

Optimizations / Devices (8192)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
2x coarsening	1.22	1.10	0.95
4x coarsening	1.14	0.83	1.05
8x coarsening	1.15	0.5	0.63

Figure 6.15: *Execution times and speedup of optimization 'Granularity' in lud_internal kernel for matrix dimension 4096 and 8192*



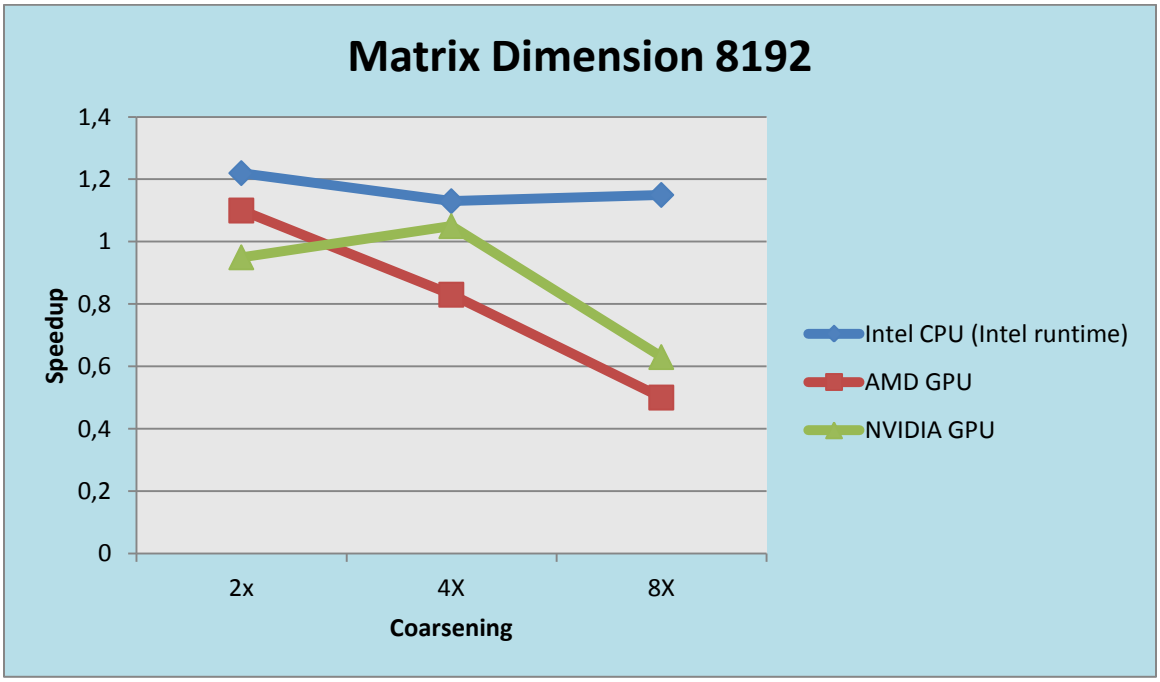


Figure 6.16: *Speedup graphical representation of optimization 'Granularity' in lud_internal kernel for matrix dimension 4096 and 8192*

b) *Geometry of work-items*

In this section we tested different sizes of the BS. The initial BS was 16. The total number of the work-items in each work-group was 256 initially because the work-groups were 2-dimensional as previously stated. We could only test a BS that was 32 because a 2-dimensional BS 32x32 is the maximum for the NVIDIA and the Intel CPU. What is more, we could not test this BS for the AMD GPU because we were already at its limit, which is 256 work-items per work-group.

Increasing the BS there was a speedup for the CPU. Having a bigger BS means that the matrices in the local memory are larger and the work-groups are fewer as we can understand from the formula:

*globalWorkSize[0] = ((matrix_dim-i)/BLOCK_SIZE-1)*localWorkSize[0];*

Also the matrices in the local memory are accessed by the work-items of one work-group. All the work-items of each work-group are executed on one core of the CPU, all sharing the same local memory of that core. When the computations start there is one cache miss for the data that is fetched for the first time of each work-item. When this data is accessed from the same or other work-items in the same work-group for a second time there is a cache hit. Having more work-items in a work-group offers us fewer cache misses than work-groups with a small number of work-items for such computational patterns that reuse some values.

The NVIDIA GPU had a better execution time too. Fermi usually does not like to have 1024 work-items per work-groups and its performance is decreased with such geometry. Each SM supports up to 32768 registers and each work-group up to 16384. The current kernel uses only one register and thus, having more work-items does not make them stall. Also using one register and 1024 work-items the occupancy was 1.

Figure 6.17 and *Figure 6.18* depict the results for matrix dimension 4096 and 8192.

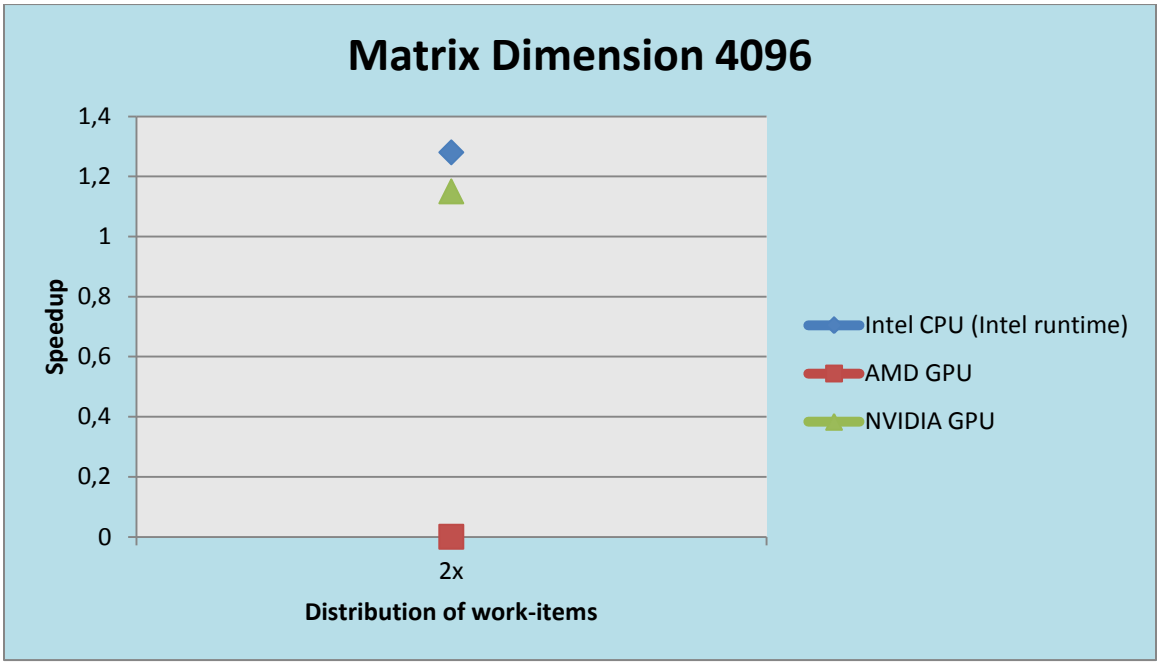
Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	1.99 sec	1 sec	0.23 sec
BS 16 to 32	1.55 sec	-----	0.20 sec

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
BS 16 to 32	1.28	-----	1.15

Optimizations / Devices (8192)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	16.12 sec	7.68 sec	1.9 sec
BS 16 to 32	12.84 sec	-----	1.78 sec

Optimizations / Devices (8192)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
BS 16 to 32	1.26	-----	1.07

Figure 6.17: *Execution times and speedup of optimization 'Geometry of work-items' in lud_internal kernel for matrix dimension 4096 and 8192*



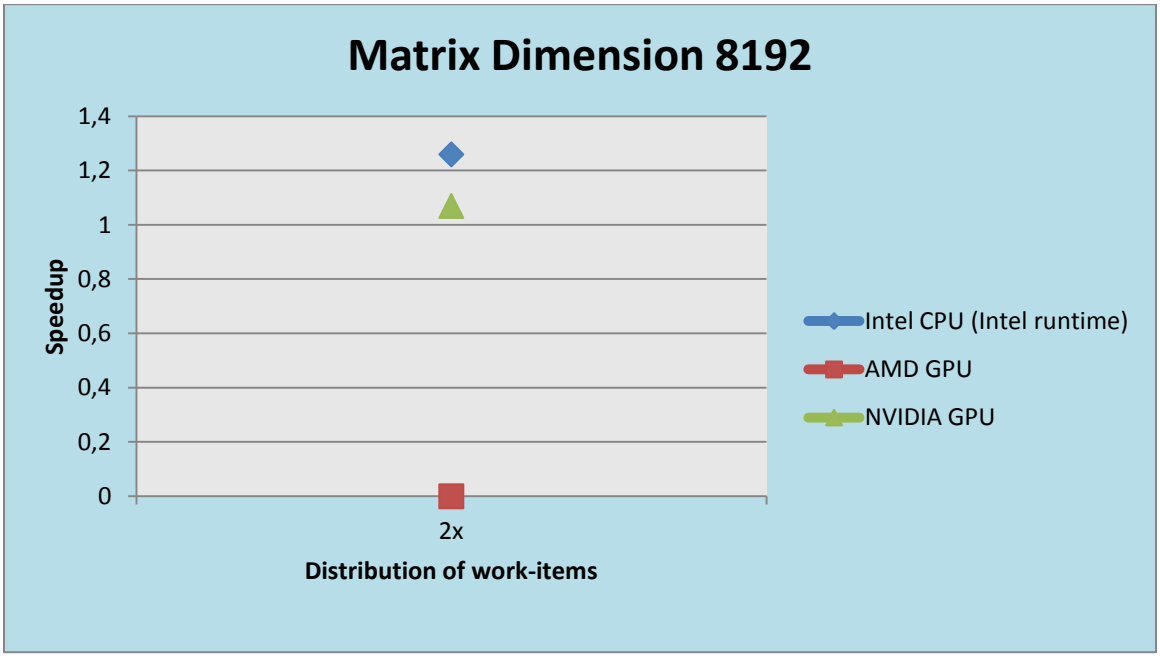


Figure 6.18: *Speedup graphical representation of optimization 'Coarsening' in lud_internal kernel for matrix dimension 4096 and 8192*

2) Vectorization

Vector Types

Applying vectorization on this kernel was feasible but not too effective. Vectorization can only be applied to favorably aligned, neighboring elements in memory. The kernel calculates the inner product of a row and a column of two 2-dimensional matrices. So, we had to transpose the one matrix in order to calculate the inner product of a row and a row of these matrices as shown in *Figure 6.19*. We made the inversion on the load from the global to the local memory but the effect of this was bad increasing the execution time of the kernel. This happened due to more cache misses. For example in the execution with BS 16 there were only 16 cache misses while in the new implementation there were 256 cache misses. The yellow line in the Figure is stored in the first cacheline in the initial load burdening with one cache miss. Storing the matrix transposed in the local memory in the second case there is a cache miss for each element.

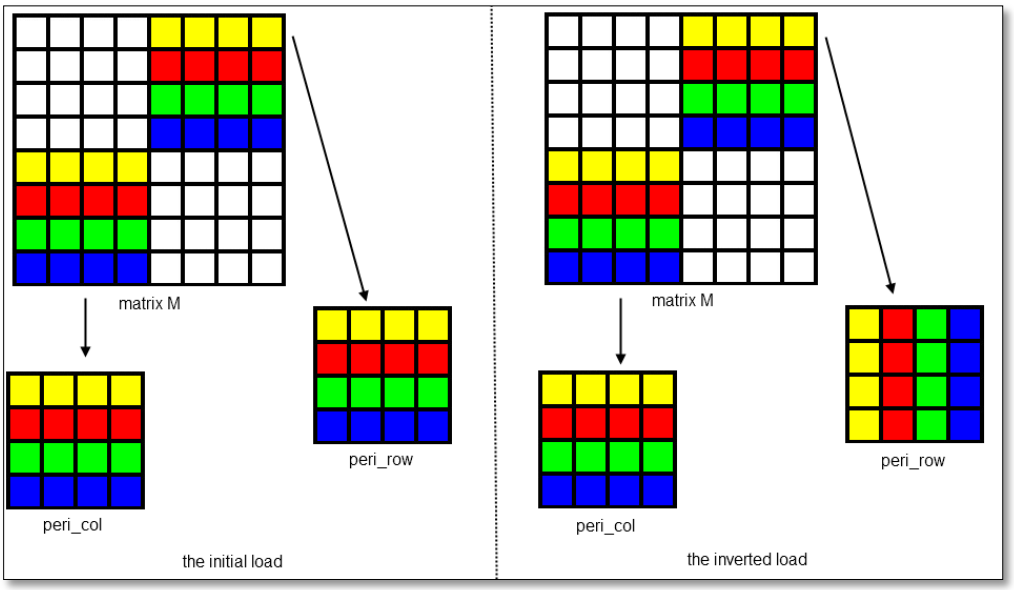


Figure 6.19: *The initial and the inverted load*

Figure 6.20 and *Figure 6.21* depict the results for matrix dimension 4096 and 8192.

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	1.99 sec	1 sec	0.23 sec
transposed matrix	4.10 sec	1.78 sec	0.77 sec

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
transposed matrix	0.48	0.56	0.3

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	16.12 sec	7.68 sec	1.9 sec
transposed matrix	35 sec	13.9 sec	6.16 sec

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
transposed matrix	0.46	0.55	0.3

Figure 6.20: *Execution times and speedup of the 'transposed load implementation' for matrix dimension 4096 and 8192*

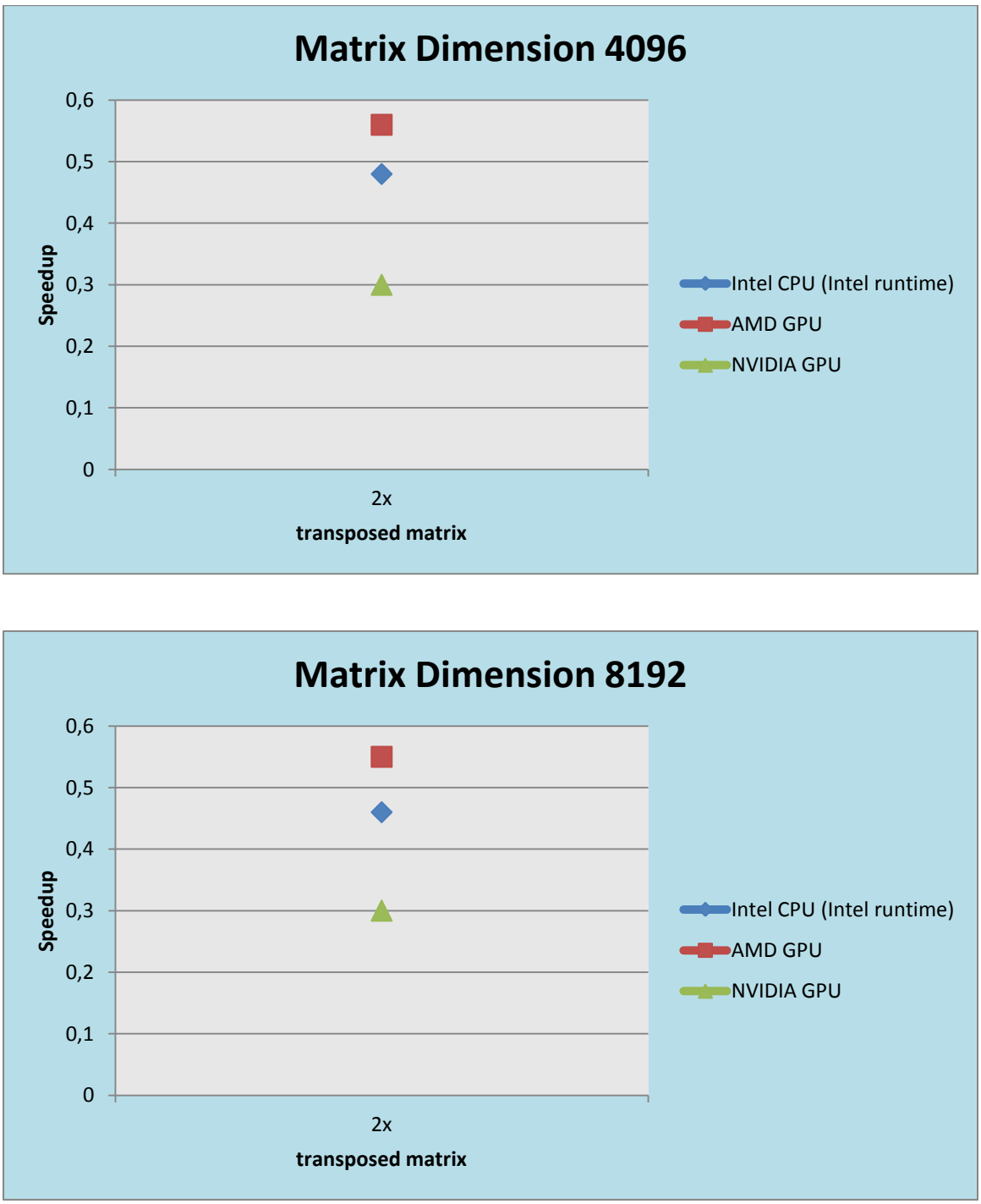


Figure 6.21: *Speedup graphical representation of optimization 'transposed matrix' in lud_internal kernel for matrix dimension 4096 and 8192*

Although this different approach did not work, we could now apply vectorization on this. Vectorization gave us a speedup in comparison with the approach we had made in order to transpose the second matrix. But the execution time was bigger than the initial time. Vectorization is a very good technique and can give a great speedup when it can be applied efficiently in an application,

especially in architectures with vector type units because they are designed to calculate more than one element per cycle if these elements are adjacent in a matrix or a vector. In our case we first had to set the matrices right and only then could we apply vectorization. Therefore we did not have a speedup in comparison to the initial code.

Although vectorization enhanced the performance of CPU, it was catastrophic for the AMD in one experiment and more specifically for a matrix 8192x8192. The problem was on the last level the 8x. The execution time was 2 times slower. The ALUPacking metric was decreased from 78% to 69% and the cache hit rate from 26% to 15%.

Figure 6.22 and *Figure 6.23* depict the results for matrix dimension 4096 and 8192.

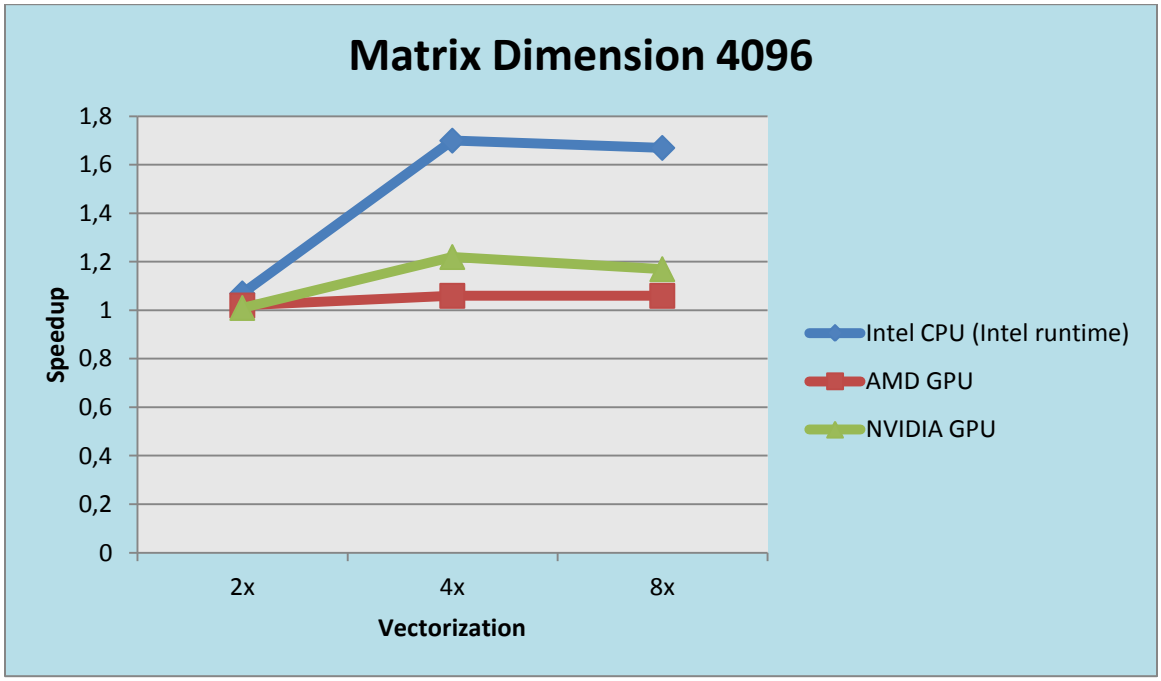
Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
transposed matrix	4.10 sec	1.78 sec	0.77 sec
Vec. 2x	3.80 sec	1.75 sec	0.76 sec
Vec. 4x	2.40 sec	1.68 sec	0.63 sec
Vec. 8x	2.46 sec	1.68 sec	0.66 sec

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Vec. 2x	1.07	1.02	1.01
Vec. 4x	1.7	1.06	1.22
Vec. 8x	1.67	1.06	1.17

Optimizations / Devices (8192)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
transposed matrix	35 sec	13.9 sec	6.16 sec
Vec. 2x	32 sec	13.7 sec	6.14 sec
Vec. 4x	22.1 sec	15.9 sec	5 sec
Vec. 8x	22.4 sec	35 sec	5.2 sec

Optimizations / Devices (8192)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Vec. 2x	1.09	1.01	1.01
Vec. 4x	1.59	0.85	1.23
Vec. 8x	1.56	0.4	1.18

Figure 6.22: Execution times and speedup of 'Vectorization' against the transposed load implementation *for matrix dimension 4096 and 8192*



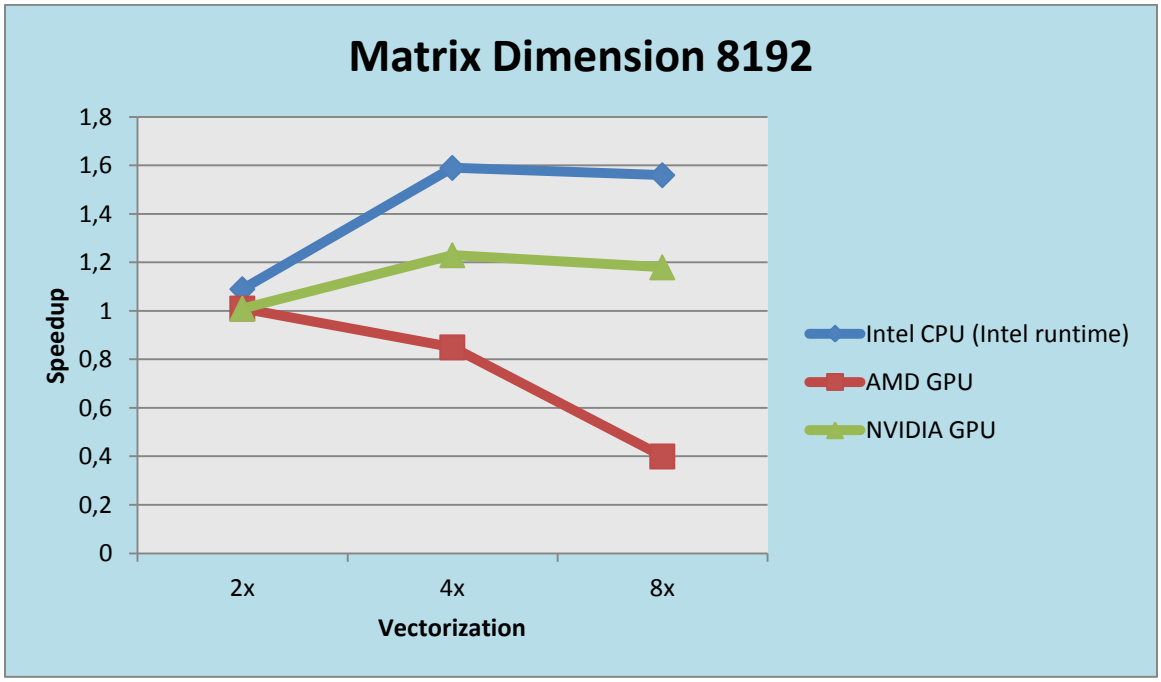


Figure 6.23: *Speedup graphical representation of optimization 'Vectorization' in lud_internal kernel for matrix dimension 4096 and 8192*

Figure 6.24 and *Figure 6.25* depict the results for matrix dimension 4096 and 8192.

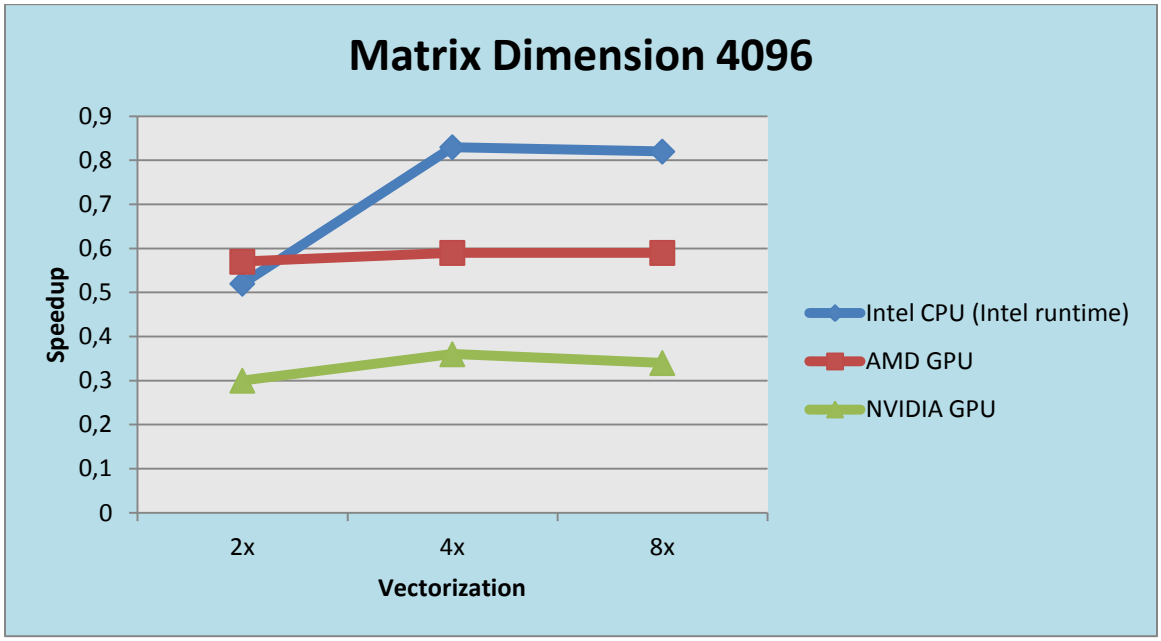
Optimizations / Devices (4096)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	1.99 sec	1 sec	0.23 sec
Vec. 2x	3.80 sec	1.75 sec	0.76 sec
Vec. 4x	2.40 sec	1.68 sec	0.63 sec
Vec. 8x	2.46 sec	1.68 sec	0.66 sec

Optimizations / Devices (4096)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Vec. 2x	0.52	0.57	0.3
Vec. 4x	0.83	0.59	0.36
Vec. 8x	0.82	0.59	0.34

Optimizations / Devices (8192)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	16.12 sec	7.68 sec	1.9 sec
Vec. 2x	32 sec	13.7 sec	6.14 sec
Vec. 4x	22.1 sec	15.9 sec	5 sec
Vec. 8x	22.4 sec	35 sec	5.2 sec

Optimizations / Devices (8192)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Vec. 2x	0.5	0.56	0.32
Vec. 4x	0.73	0.48	0.38
Vec. 8x	0.72	0.22	0.36

Figure 6.24: Execution times and speedup of 'Vectorization' against the initial implementation for *matrix dimension 4096 and 8192*



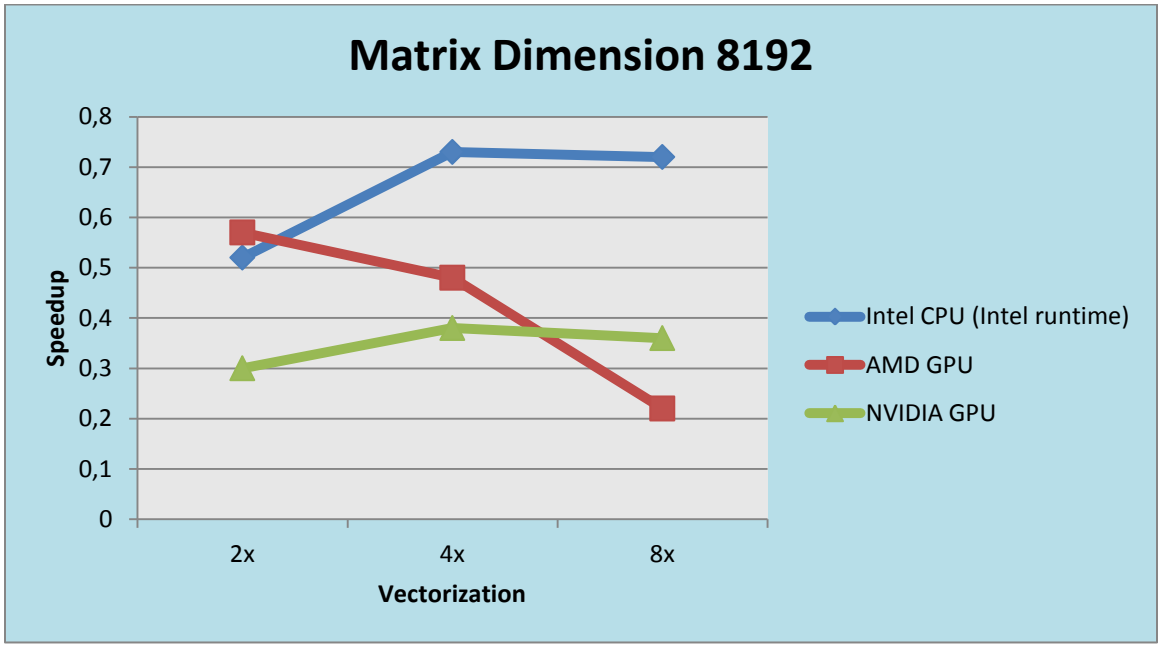


Figure 6.25: *Speedup graphical representation of optimization 'Vectorization' in lud_internal kernel for matrix dimension 4096 and 8192*

3) Loops

Loop Unrolling

The last optimization for this kernel was the loop unrolling. We did the loop unrolling with step 2 and step 4. Loop unrolling is useful because it can hide the cache misses or allow the instructions to be scheduled better. It can improve the performance using a technique called pipeling. Each

instruction can be split into a sequence of dependent steps. The first step is to fetch the instruction from the memory and the last to store the result in the memory. Pipeling can execute different steps of many instructions to increase instruction throughput and seeks to keep every portion of the CPU busy with some instruction. Although the loop unrolling offers these advantages, the execution times were the same with the initial ones, except for the unrolling with step 4 on CPU, where it was slightly faster.

Figure 6.26 and *Figure 6.27* depict the results for matrix dimension 4096 and 8192.

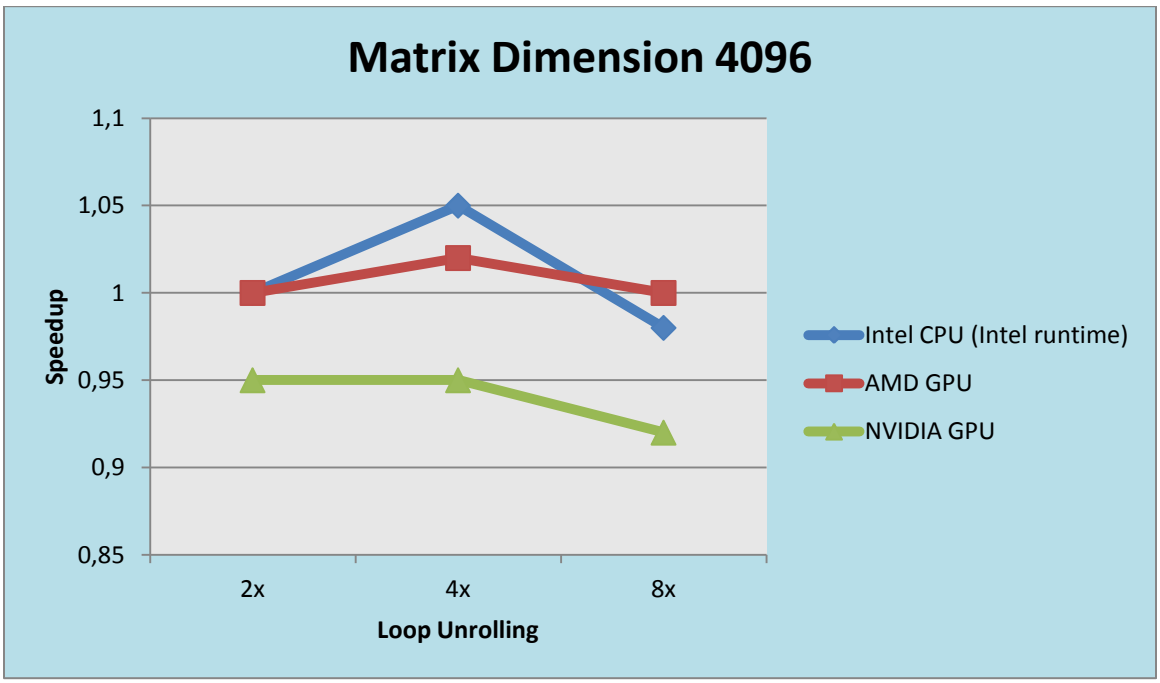
Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	1.99 sec	1 sec	0.23 sec
Unroll 2	1.99 sec	1 sec	0.24 sec
Unroll 4	1.90 sec	0.98 sec	0.24 sec
Unroll 8	2.02 sec	1 sec	0.25 sec

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Unroll 2	1	1	0.95
Unroll 4	1.05	1.02	0.95
Unroll 8	0.98	1	0.92

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	16.12 sec	7.68 sec	1.9 sec
Unroll 2	15.21 sec	8 sec	1.9 sec
Unroll 4	15.42 sec	7.68 sec	1.9 sec
Unroll 8	15.6 sec	7.76 sec	1.9 sec

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Unroll 2	1.06	0.96	1
Unroll 4	1.05	1	1
Unroll 8	1.07	0.99	1

Figure 6.26: *Execution times and speedup of optimization 'Loop Unrolling' in lud_internal kernel for matrix dimension 4096 and 8192*



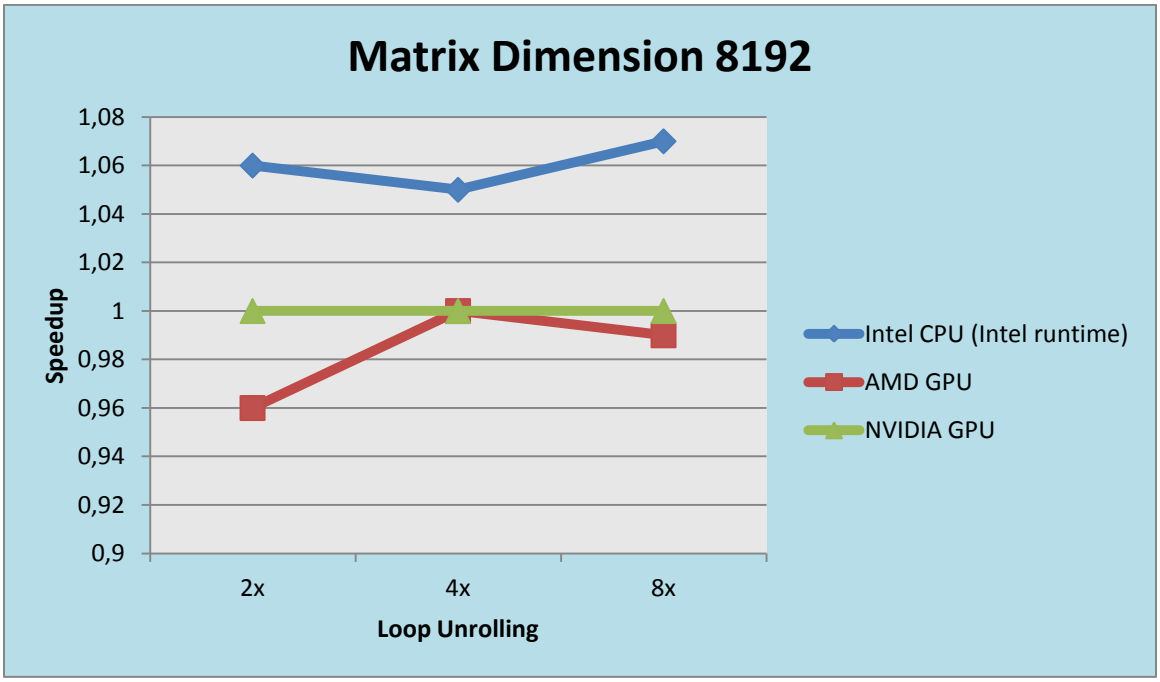


Figure 6.27: *Speedup graphical representation of optimization 'Loop Unrolling' in lud_internal kernel for matrix dimension 4096 and 8192*

6.2 crc

Cyclic Redundancy Check (CRC) is an error-detecting code which is designed to detect errors caused by network transmission or any other accidental error. Specifically, CRC is designed to catch burst errors in data that is transferred. Polynomial division is performed on the data stream S by the CRC polynomial. This polynomial is predetermined. The remainder from this division is the CRC value. This value is typically added to the end of the data stream as it is sent out. When the receiver divides the received data stream, the division will return no remainder on a successful transmission.

Application *crc* belongs to the *combinational logic* category of dwarfs and aparts from one kernel, the *compute* kernel. All the optimizations that studied, applied in this kernel.

6.2.1 Analysis of *compute* kernel

6.2.1.1 Data Dependencies

In *compute* kernel every work-item is responsible for the computation of one element (unsigned char type) of the data stream size (*data_stream_size*) and there is no dependency on data between different work-items. The dependency is located on the value that this element takes which affects the next iteration of a *for loop* that executed 32 times per work-item. Within the loop statement, exists an *if statement* that defines if the afore-mentioned unsigned char value must change or not. Therefore the parallelism per work-item can be found in the check condition of these *if statements* among the iterations of loop. *Figure 6.28* depicts the data dependencies of *compute* kernel.

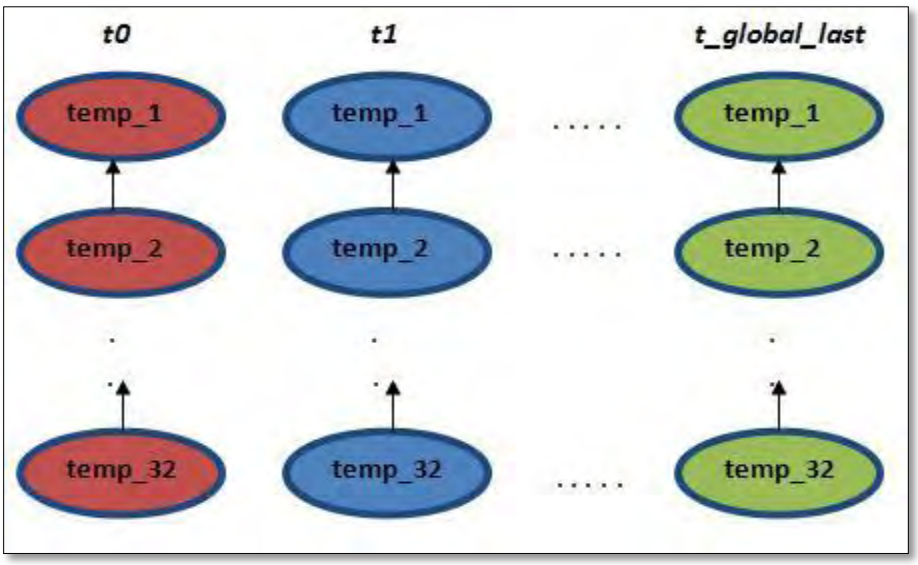


Figure 6.28: *Data dependencies of compute kernel*

As can been seen from the afore-depicted figure, the first column corresponds to work-item 0, the second to work-item 1 and the last to work-item with the highest global id. Every work-item has a private variable, which called *temp*, and executes one loop iteration 32 times. The processing of variable *temp* is independent among work-items and therefore the latter can all executed in parallel at each iteration step. We must mention that with the names *temp_1*, *temp_2* and *temp_32* we denote the instant values that variable *temp* has at each iteration.

Within loop there is an *if statement* which defines if the value of variable *temp* changes or not. If it changes, the instant value of *temp* that follows, needs the previous instant value in order to be computed, otherwise, it takes the same value as its previous instant has (the value of *temp* doesn't change among consecutive iterations). For instance, instant value *temp_2* is computed via an arithmetic expression in which instant value *temp_1* participates if the check condition of *if statement* holds, otherwise *temp_2* takes the value of *temp_1* as it is. Therefore, as can be inferred, the data dependencies of *compute* kernel have a vertical per work-item dependency pattern.

Last but not least, it's worth to mention that the *compute* kernel is called one time and the way that the work-group size (*local_size*) and the total number of work-items (*global_size*) are defined, is the following:

```
local_size = maximum_per_work_group_size

if ( (data_stream_size % local_size) == 0 ) {
   global_size = data_stream_size;
}
else {
   global_size = ( (data_stream_size / local_size) + 1 ) * local_size;
}
```

6.2.1.2 Basic code segment

The code of *compute* kernel is small and can be presented here as it is so as to help one to realize better the form of communication and computation that exists. The *num_size* here represents the *data_stream_size*.

```
unsigned int tid = get_global_id(0);

if (tid < num_size) {
   int val = num_size – tid;
   int i = 0;
   unsigned char temp = g_num[tid];
```

```
    for (i = 0; i < numTables; i++) {
        If ( ( (val >> i) % 2 ) == 1 ) {
            temp = g_tables[i*256 + temp];
        }
    }

    g_answer[tid] = temp;
}
```

6.2.1.3 Optimization Efforts and Results

1) Execution Geometry

a) *Granularity*

In this category, four optimization parameters are examined in *compute* kernel. In every one of them, more work added per work-item. Therefore, 2x, 4x, 8x and 16x coarsening per work-item is tested. *2x coarsening* means that the work-item computes two elements of data stream, while *4x coarsening* corresponds to four elements, *8x coarsening* to eight and *16x coarsening* to sixteen.

The result is that larger coarsening per work-item proves gradually better for Intel CPU and AMD GPU, whereas in NVIDIA GPU gets poorer in the case of *8x coarsening* and ends to the same performance with initial implementation of *compute* kernel in the case of *16x coarsening*. The speedup is independent of the data stream size that used for both GPUs, while it is bigger, as data stream size increases, for Intel CPU. The best performance achieved in *16x coarsening* case for Intel CPU and AMD GPU, while the *2x or 4x coarsening* is the most suitable for NVIDIA GPU.

Figure 6.29 and *Figure 6.30* depict the results for data stream size (num_size) 50,000,000, 100,000,000 and indicatively for num_size 200,000,000.

Optimizations / Devices (50,000,000)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	1,254,630,292 ns	290,753,334 ns	38,345,792 ns
2x coarsening	1,175,741,674 ns	240,108,667 ns	32,125,920 ns
4x coarsening	1,059,316,692 ns	222,618,556 ns	32,211,808 ns
8x coarsening	1,040,567,243 ns	211,527,112 ns	33,862,304 ns
16x coarsening	1,034,929,668 ns	205,409,667 ns	36,038,400 ns

Optimizations / Devices (50,000,000)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
2x coarsening	1	1.21	1.19
4x coarsening	1.18	1.31	1.19
8x coarsening	1.21	1.37	1.13
16x coarsening	1.21	1.42	1

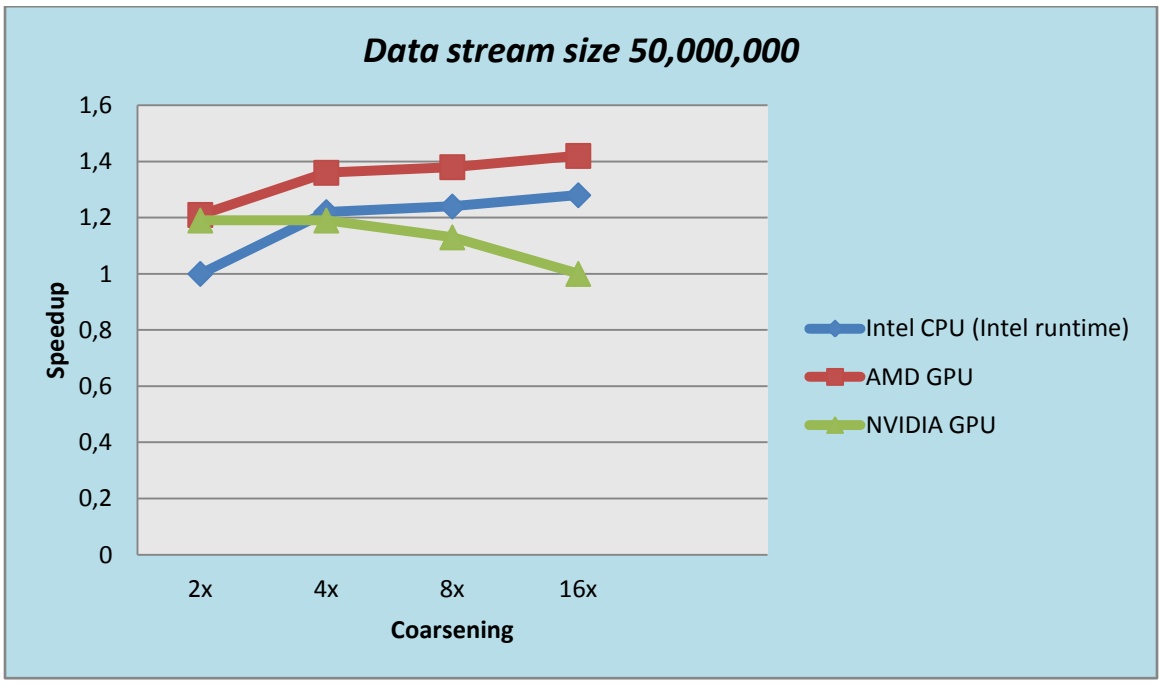
Optimizations / Devices (100,000,000)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	2,486,422,284 ns	591,497,333 ns	76,708,192 ns
2x coarsening	2,604,607,235 ns	487,691,777 ns	64,008,160 ns
4x coarsening	2,031,097,642 ns	434,757,778 ns	64,394,464 ns
8x coarsening	2,002,005,016 ns	427,972,000 ns	67,547,168 ns
16x coarsening	1,948,490,480 ns	416,722,222 ns	72,094,048 ns

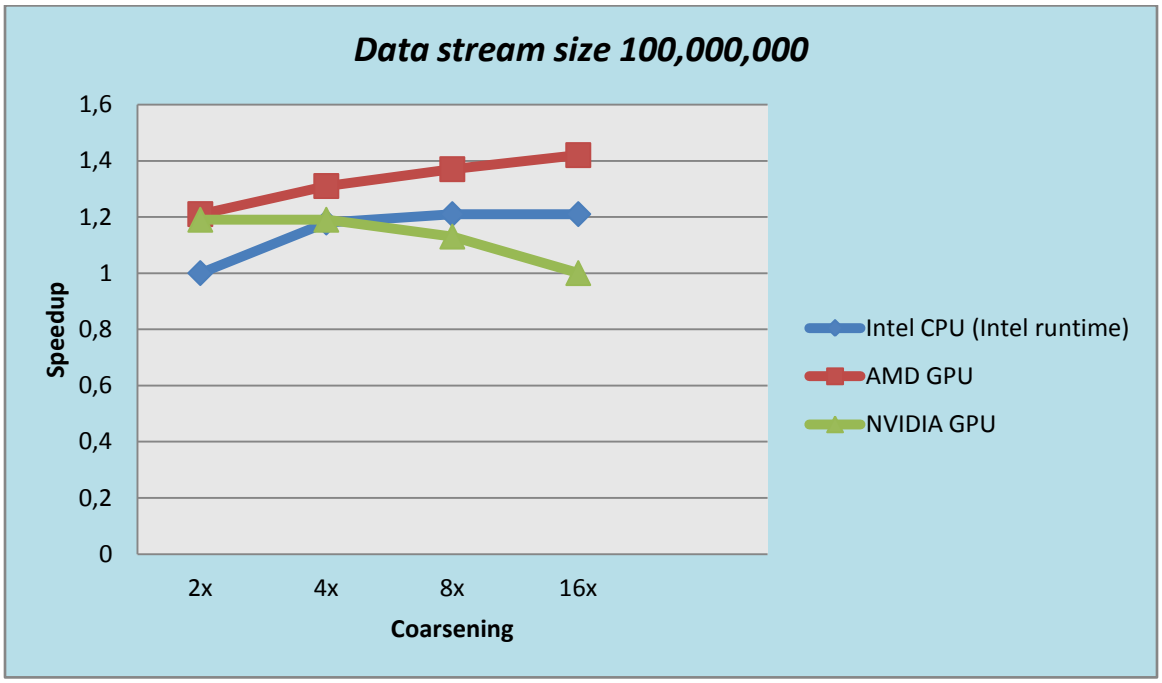
Optimizations / Devices (100,000,000)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
2x coarsening	1	1.21	1.19
4x coarsening	1.22	1.36	1.19
8x coarsening	1.24	1.38	1.13
16x coarsening	1.28	1.42	1

Optimizations / Devices (200,000,000)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	5,102,533,206 ns	1,192,723,444 ns	153,434,400 ns
16x coarsening	3,884,520,078 ns	843,412,000 ns	143,868,288 ns

Optimizations / Devices (200,000,000)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
16x coarsening	1.31	1.41	1

Figure 6.29: *Execution times and speedup of optimization 'Granularity' in compute kernel for num_size 50,000,000, 100,000,000 and indicatively for num_size 200,000,000*





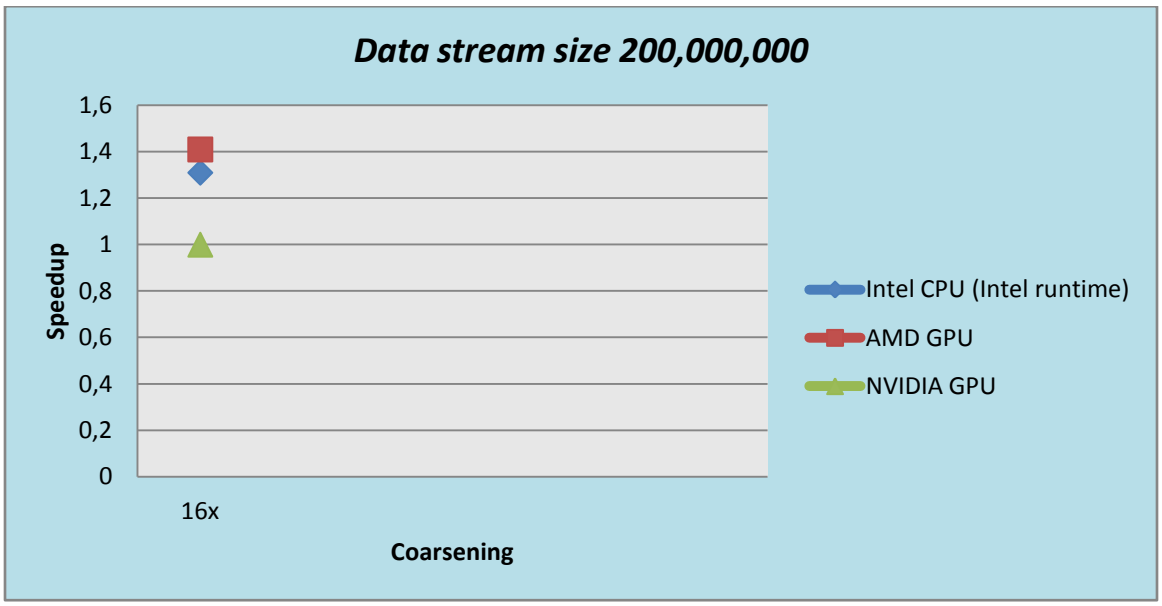


Figure 6.30: *Speedup graphical representation of optimization 'Granularity' in compute kernel for num_size 50,000,000, 100,000,000 and indicatively for num_size 200,000,000*

The reason that larger coarsening per work-item helps the Intel CPU to perform well is that the total number of work-groups per kernel is reduced dramatically as the coarsening per work-item increases. The work-group size remains the same and each work-item processes more elements of data stream, but this yields fewer context switches and this is very beneficial for CPUs, as the operation system intervenes fewer times. A characteristic example can be the fact that in *initial code*, the kernel is called for Intel CPU and data stream size 200,000,000, with 195,313 work-groups of 1024 work-items, whereas in *16x coarsening* case it is called with 12,208 work-groups of 1024 work-items.

On the other side, AMD GPU performs well with larger coarsening, as it makes effective use of its hardware paths. Specifically, an indicative paradigm is that for data stream size 200,000,000 in *initial code*, the *FastPath* is 0 and the *CompletePath* is 781,252, whereas in *16x coarsening* case the *FastPath* is 195,316 and the *CompletePath* is 0. The *PathUtilization* in the first case is 25%, while in the second is 100%. The values of paths denote the total kilobytes written to the video memory of AMD GPU through their utilization. Generally, the *FastPath* does not support atomics or sub-32 bit data types, while the *CompletePath* supports them. In *compute* kernel, larger coarsening leads each work-item to use bigger data-types (vector types) so as to process effectively more elements and this has as a result the utilization of hardware *FastPath* instead of *CompletePath*, something that is very significant for enhancement in performance for our AMD GPU.

Another important reason that explains the great performance that AMD GPU achieves via the use of larger coarsening per work-item is the decrease of average number of ALU instructions executed by the total number of work-items within kernel and the increase of the number of general purpose vector registers used by the kernel. For instance, for data stream size 200,000,000, in *initial code*, the total average number of ALU instructions that executed is 200,000,000 * 444.62 = 8.89 * 10^{10}, while in *16x coarsening* case, this number drops to 12,500,224 * 4936.54 = 6.17 * 10^{10}.

The first multiplier is the total number of work-items per kernel, while the second is the average number of ALU instructions executed per work-item. Moreover, the increase of vector registers used by the kernel means that the respective computations are done faster in the second case.

Last but not least, as can be derived from the *Figure 6.29*, for the cases of *2x coarsening* and *4x coarsening*, NVIDIA GPU seems to make effective use per work-item of the registers that each one of them has and this enhances the performance. However, coarsening per work-item larger than 4, causes bad utilization of hardware resources, as the total number of work-groups drops significantly.

b) *Geometry of work-items*

In this category, five optimization parameters are examined in *compute* kernel. Namely, the maximum work-group size of initial implementation reduces /2, /4, /8, /16 and /32. The results are disappointing for both GPUs (mainly for AMD GPU) that tested and are independent of the data stream size that used.

Figure 6.31 and *Figure 6.32* depict the results for data stream size (num_size) 50,000,000, 100,000,000 and indicatively for num_size 200,000,000. The results are presented only for GPUs, as the performance of Intel CPU remains steady among the afore-mentioned optimization efforts.

Optimizations / Devices (50,000,000)	**AMD GPU**	**NVIDIA GPU**
Initial Code	290,753,334 ns	38,345,792 ns
BS max to /2	292,913,444 ns	37,170,528 ns
BS max to /4	350,808,889 ns	37,081,504 ns
BS max to /8	686,559,333 ns	37,099,104 ns
BS max to /16	1,358,387,000 ns	42,651,520 ns
BS max to /32	2,668,649,667 ns	79,585,120 ns

Optimizations / Devices (50,000,000)	**AMD GPU**	**NVIDIA GPU**
BS max to /2	1	1
BS max to /4	0.83	1
BS max to /8	0.42	1
BS max to /16	0.21	0.90
BS max to /32	0.11	0.48

Optimizations / Devices (100,000,000)	**AMD GPU**	**NVIDIA GPU**
Initial Code	591,497,333 ns	76,708,192 ns
BS max to /2	591,379,111 ns	74,348,352 ns

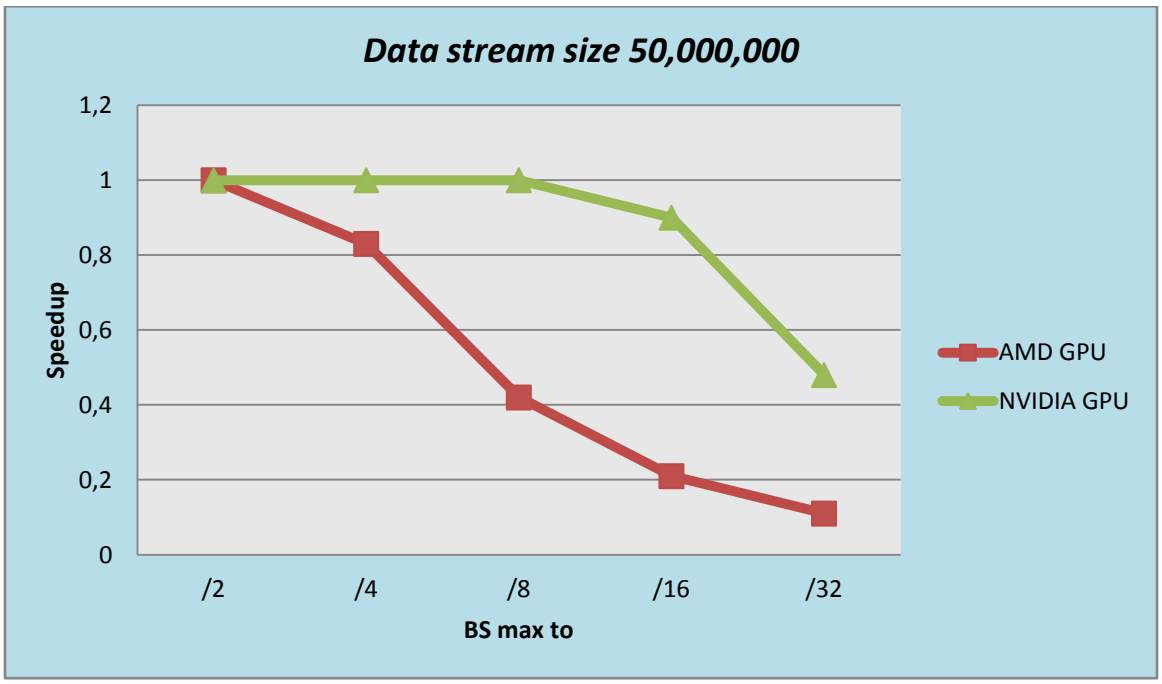
BS max to /4	708,741,445 ns	74,135,040 ns
BS max to /8	1,386,998,445 ns	74,206,368 ns
BS max to /16	2,743,588,777 ns	85,506,016 ns
BS max to /32	5,384,053,111 ns	159,707,520 ns

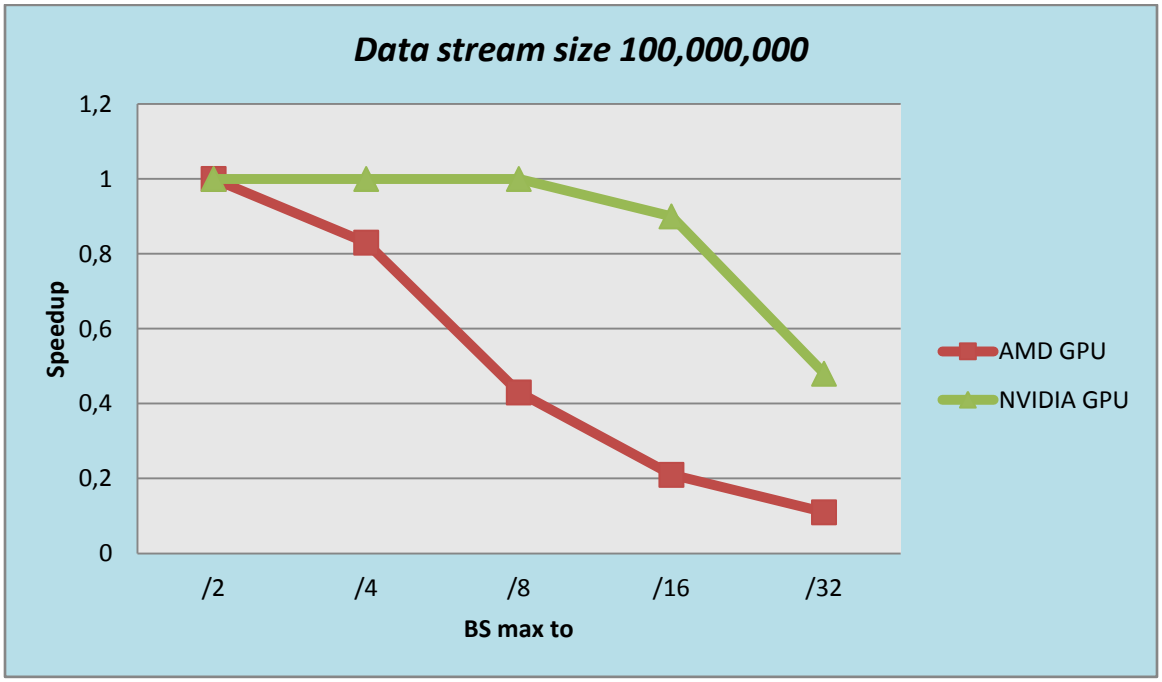
Optimizations / Devices (100,000,000)	**AMD GPU**	**NVIDIA GPU**
BS max to /2	1	1
BS max to /4	0.83	1
BS max to /8	0.43	1
BS max to /16	0.21	0.90
BS max to /32	0.11	0.48

Optimizations / Devices (200,000,000)	**AMD GPU**	**NVIDIA GPU**
Initial Code	1,192,723,444 ns	153,434,400 ns
BS max to /32	10,895,424,667 ns	320,562,656 ns

Optimizations / Devices (200,000,000)	**AMD GPU**	**NVIDIA GPU**
BS max to /32	0.11	0.48

Figure 6.31: *Execution times and speedup of optimization 'Geometry of work-items' in compute kernel for num_size 50,000,000, 100,000,000 and indicatively for num_size 200,000,000*





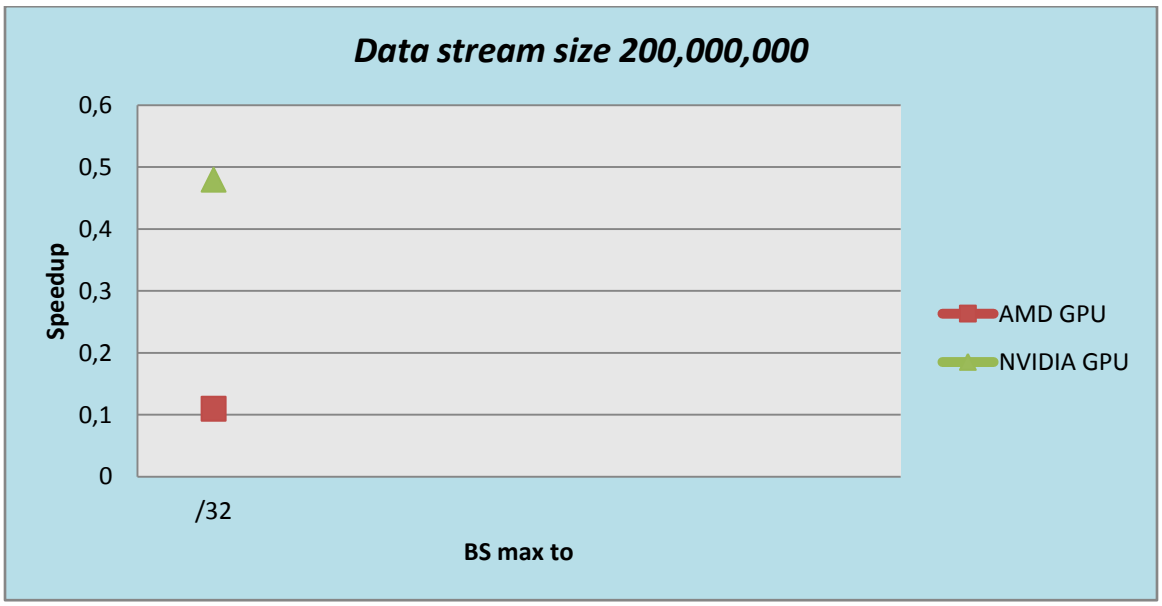


Figure 6.32: *Speedup graphical representation of optimization 'Geometry of work-items' in compute kernel for num_size 50,000,000, 100,000,000 and indicatively for num_size 200,000,000*

The bad results can be attributed to the ineffective utilization of hardware resources that GPUs offer. An indicative paradigm can be the fact that for AMD GPU and data stream size 200,000,000 in *initial code*, *compute* kernel is called with 781,250 work-groups of 256 work-items, while in *BS max to /32* case, it is called with 25,000,000 work-groups of 8 work-items in each of them. A *wavefront* in AMD GPU is the basic parallel execution unit and aparts from 64 work-items which execute in parallel in every cycle. The reduction of work-group size leads to poor utilization of hardware resources and has as a result many of them to prove useless and unexploited, causing at the same time unnecessary hardware scheduling overhead.

The lower degradation of performance in NVIDIA GPU can be explained knowing that in NVIDIA the respective basic parallel execution unit called *warp* and comprises 32 work-items. Therefore, the underutilization of hardware resources exists, but is less in its case.

Something that it's worth to mention here is the steady performance of Intel CPU. 195,313 work-groups of 1024 work-items in *initial code* transform to 6,250,000 work-groups of 32 work-items for data stream size 200,000,000 without any loss or gain of performance. Presumambly, the overhead of context switches in *compute* kernel is minor compared to the computation overhead per work-item.

2) Loops

Loop Unrolling

In this category, five optimization parameters are investigated. The first parameter is *2x loop unrolling*, the second is *4x loop unrolling*, the third is *8x loop unrolling*, the forth is *16x loop unrolling* and the last is *32x loop unrolling*. The performance results are gradually better as the degree of loop unrolling increases for all the devices that used. What's more, the speedup is a little higher in Intel CPU for larger data stream sizes, whereas remains unaffected between different data stream sizes in both GPUs.

Figure 6.33 and *Figure 6.34* depict the results for data stream size (num_size) 50,000,000, 100,000,000 and indicatively for num_size 200,000,000.

Optimizations / Devices (50,000,000)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	1,254,630,292 ns	290,753,334 ns	38,345,792 ns
2x loop unrolling	1,174,701,296 ns	269,877,667 ns	34,979,744 ns
4x loop unrolling	1,100,311,153 ns	259,103,222 ns	32,358,560 ns
8x loop unrolling	1,006,472,444 ns	253,875,222 ns	31,236,224 ns
16x loop unrolling	986,318,872 ns	251,222,000 ns	30,707,360 ns
32x loop unrolling	939,605,499 ns	193,335,667 ns	30,368,736 ns

Optimizations / Devices (50,000,000)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
2x loop unrolling	1	1	1.10
4x loop unrolling	1.14	1.12	1.18
8x loop unrolling	1.25	1.14	1.23
16x loop unrolling	1.27	1.16	1.25
32x loop unrolling	1.33	1.50	1.26

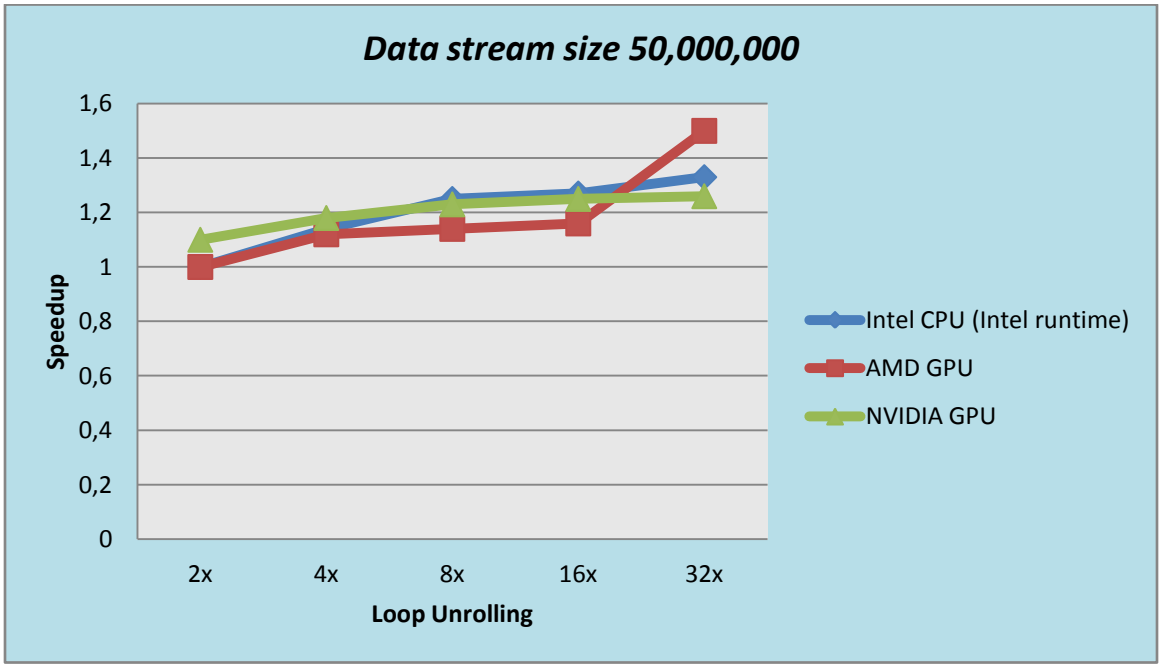
Optimizations / Devices (100,000,000)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	2,486,422,284 ns	591,497,333 ns	76,708,192 ns
2x loop unrolling	2,346,486,526 ns	549,576,334 ns	69,977,984 ns
4x loop unrolling	2,102,031,729 ns	528,980,555 ns	64,739,680 ns
8x loop unrolling	1,921,112,067 ns	518,508,333 ns	62,511,488 ns
16x loop unrolling	1,876,315,374 ns	513,093,889 ns	61,460,192 ns
32x loop unrolling	1,784,348,121 ns	396,903,333 ns	61,075,264 ns

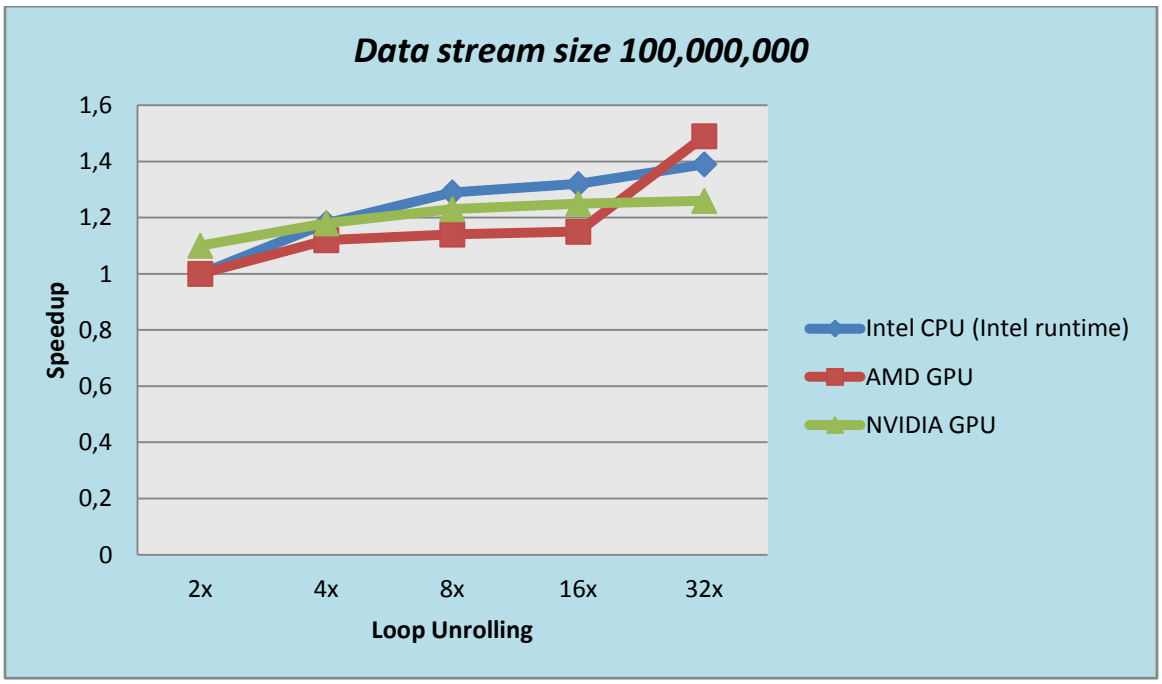
Optimizations / Devices (100,000,000)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
2x loop unrolling	1	1	1.10
4x loop unrolling	1.18	1.12	1.18
8x loop unrolling	1.29	1.14	1.23
16x loop unrolling	1.32	1.15	1.25
32x loop unrolling	1.39	1.49	1.26

Optimizations / Devices (200,000,000)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	5,102,533,206 ns	1,192,723,444 ns	153,434,400 ns
32x loop unrolling	3,770,744,351 ns	804,700,556 ns	123,113,664 ns

Optimizations / Devices (200,000,000)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
32x loop unrolling	1.35	1.48	1.25

Figure 6.33: *Execution times and speedup of optimization 'Loop Unrolling' in compute kernel for num_size 50,000,000, 100,000,000 and indicatively for num_size 200,000,000*





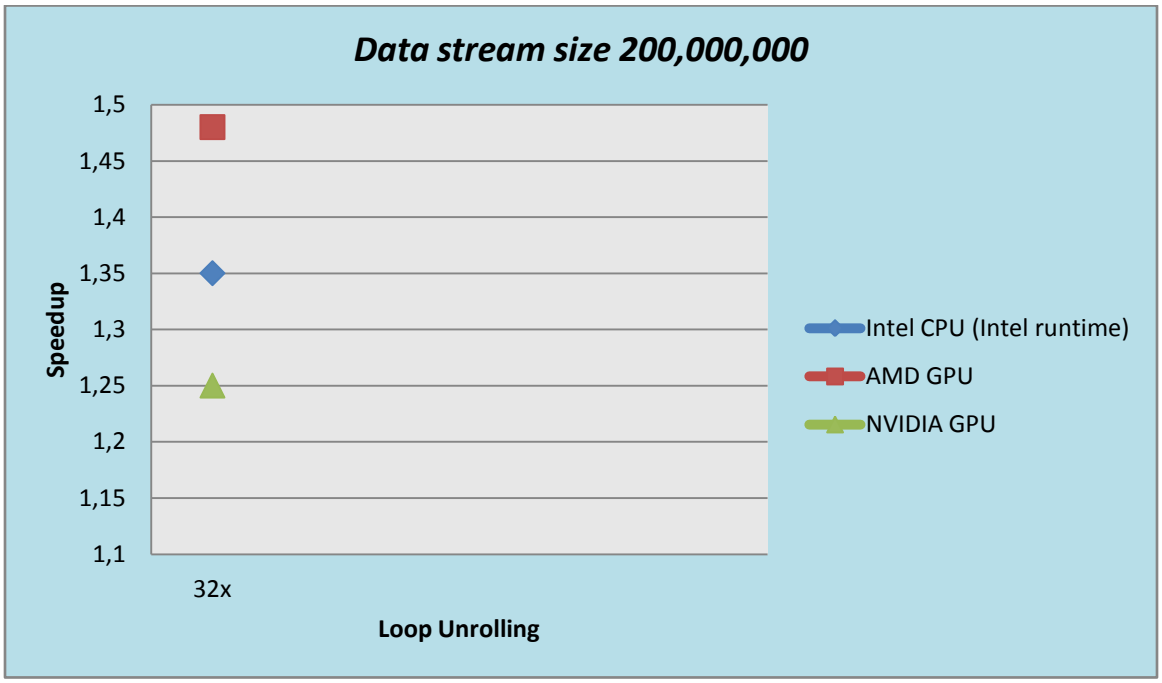


Figure 6.34: *Execution times and speedup of optimization 'Loop Unrolling' in compute kernel for num_size 50,000,000, 100,000,000 and indicatively for num_size 200,000,000*

A general reason that explains the uniform improvement in performance to all the devices relates with the known advantages that the optimization technique of loop unrolling offers. These are that fewer instructions are executed per work-item and that each architecture makes more effective use of the registers that possesses.

But, in more detail, as we said, AMD's VLIW pipelines are a multi-precision, staggered design that can bypass results between the pipelines. The operations within a VLIW bundle (VLIW4) can be independent (just like a 4-wide SIMD). This means that ideally four pairs of serially dependent or independent instructions can be packed into a single VLIW bundle. We remind that the best is having independent instructions so as not use the pipeline's transfer within and among VLIW4 units. In *compute* kernel this happens for the consecutive *if statements* that the technique of loop unrolling creates. For instance, the if statements *if (((val >> i) % 2) == 1), if (((val >> i+1) % 2) == 1) , if (((val >> i+2) % 2) == 1)* and *if (((val >> i+3) % 2) == 1)* can grouped in a single VLIW4 unit and executed in paralllel with a very effective way reducing the average number of ALU instructions executed per work-item.

A characteristic case is that for data stream size 100,000,000 where ALU instructions in initial implementation are 440, whereas in the two most indicative cases, 16x unrolling and 32x unrolling, this number falls to 381 and 238 respectively. The huge increase of speedup for AMD GPU during the transition from *16x unrolling* case to *32x unrolling* case can be attributed to the fact that the number of general vector registers used by the kernel in the second case increased from 3 to 18 and this results to better utilization of VLIW4 execution units and dramatic reduction to ALU instructions.

On the other side, NVIDIA GPU doesn't have vector processors, so it cannot handle very effectively a form of code that is overwhelmed by if statements, since this doesn't let it make optimal use of registers in comparison with an unrolling without branches, whereas Intel CPU can exploit effectively its ILP characteristic in *compute* kernel compared to the case of *lud_perimeter* kernel because in this kernel some independent instructions intermediate between dependent ones.

A paradigm which exhibits that is the case where instruction *temp = g_tables[(i + 2) * 256 + temp]* is dependent with the instruction *temp = g_tables[(i + 1) * 256 + temp]* provided that the relative *if condition* of the latter holds, but among them intermediates the independent instruction *if (((val >> i+2) % 2) == 1)* and Intel CPU can take advantage of its ILP metric to all of the instruction dependencies of this form. Moreover, higher degree of loop unrolling creates more these instruction dependencies, so the speedup gets gradually higher.

6.2.1.3 Unfeasible Optimizations

1) Vectorization

Vector Types

The optimization technique *Vectorization* wasn't feasible to be implemented here, since the *if statement* of *compute* kernel code doesn't guarantee that the check condition will give the same logical result (true or false) for the *if statements* of successive elements of data stream size. Vectorization requests that the vectorized data must be executed in the same clock cycle and this can't happen in this kernel.

6.3 needle

Needleman-Wunsch is an algorithm for calculating optimal global alignment of two DNA sequences. All-pairs matching are represented by an NxM 2D matrix. Usually the matrix dimensions N, M are equal and some typical values for these are 4096, 8192, 16384. Matrix cells are scored from northwestern-most to southeastern-most (*wavefront computational pattern*), depending solely on northern, northeastern, and eastern neighboring scores. Finally, the algorithm backtracks through the array to return an optimal alignment.

Application *needle* belongs to the *dynamic programming* category of dwarfs and aparts from two kernels, the *needle_opencl_shared_1* kernel and the *needle_opencl_shared_2* kernel. Execution times of these kernels are exactly the same, as both kernels have the same code and follow the same communication and computation pattern. Therefore, the analysis of both of them would be unnecessary, so the optimization efforts implemented solely on *needle_opencl_shared_1* kernel and the explanation of the results is done only for this kernel. The difference among both kernels can be depicted in *Figure 6.35*. Blue boxes correspond to kernel_1, while red ones to kernel_2.

6.3.1 Analysis of *needle_opencl_shared_1* kernel

6.3.1.1 Data Dependencies

This kernel supports two sequences with the same length, which can be divided by 16. The kernel is called iteratively and the number of iterations depends from the work-group and the 2D matrix dimension sizes. In each successive call of kernel, more work-groups are used with the same work-item size and work-groups follow a wavefront computation pattern. The maximum number of work-groups is needed in order to compute the elements of main diagonal of 2D matrix that used. This point also denotes the end of *needle_opencl_kernel_1* kernel and after that begins the iterative execution of *needle_opencl_kernel_2* kernel, as can be seen in *Figure 6.35*. The synchronization point among work-groups is the different kernel calls that done.

Additionally, the work-items within work-groups apply also the wavefront computation pattern. This means that they are allocated in order to fill all the diagonal elements of work-group size iteration per iteration and there is a sunchronization point among previous and next diagonals among consecutive iterations. The number of active work-items increases as the size of diagonal increases in each next iteration. However, this happens only till the main diagonal of work-group size, since after this point the number of active work-items gradually decreases.

In short, the wavefront form of computation and communication is among the work-items of work-group and among work-groups of kernel. Furthermore, the data dependencies of each work-item within a work-group can be shown in *Figure 6.36*.

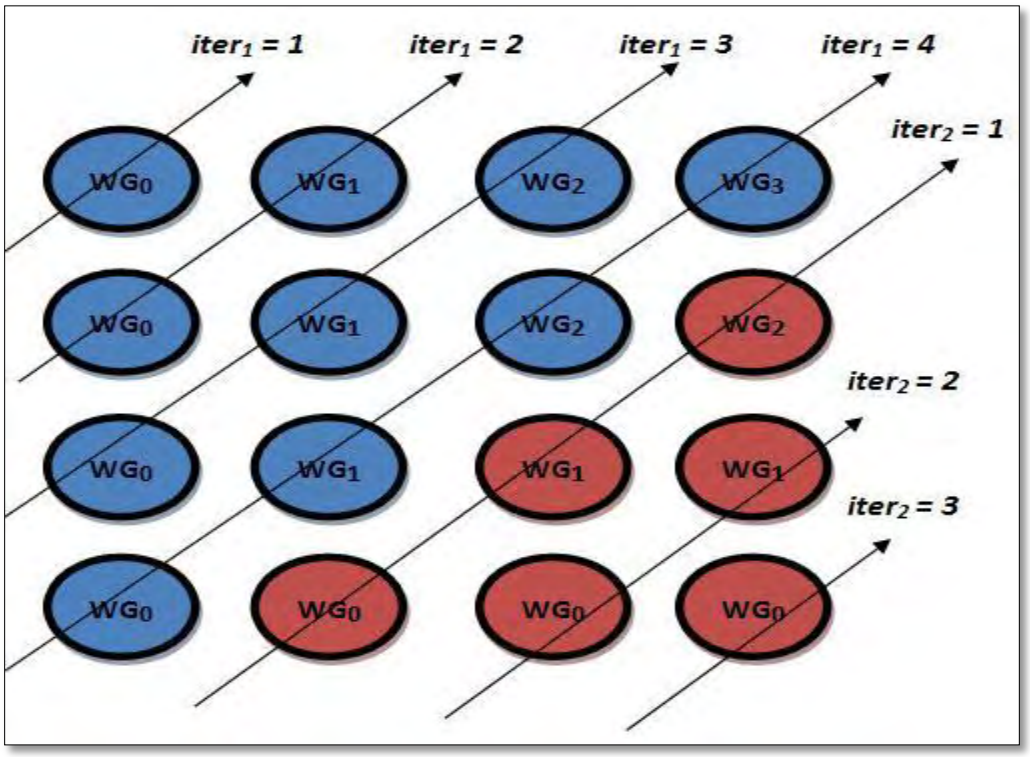


Figure 6.35: *Wavefront computation pattern of work-groups in needle for a 2D matrix of size 64*64 = 4096 elements with work-group size 16*16 = 256 work-items*

As can be deduced from the *Figure 6.35*, $iter_1$ is the iteration counter of *needle_opencl_shared_1* kernel, while $iter_2$ is the iteration counter of *needle_opencl_shared_2* kernel. The cummulative work per kernel is slightly more in the kernel_1 and this happens as this kernel computes iteratively all the elements of the respective 2D matrix till its main diagonal, whereas kernel_2 computes all the rest till the complete computation of total matrix elements. Moreover, it must be noted that during the transition from one iteration to another a different kernel call is done in both kernel cases and operates as a synchronization point of the current computed elements of 2D array.

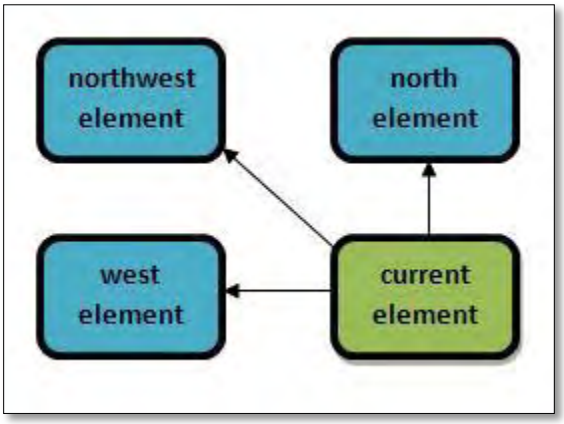


Figure 6.36: *Data dependencies of work-item in needle*

6.3.1.2 Basic code segment

The code fundamentals of *needle_opencl_shared_1* kernel presented here so as to help one to realize better the form of communication and computation that exists.

```
int bx = get_group_id(0);
int tx = get_local_id(0);

for (int m = 0; m < BLOCK_SIZE; m++) {

   if (tx <= m) {
      int t_index_x = tx + 1;
      int t_index_y = m – tx + 1;

      temp[t_index_y][t_index_x] = maximum( temp[t_index_y – 1][t_index_x – 1] +
                                            ref[t_index_y – 1][t_index_x – 1],
                                            temp[t_index_y][t_index_x – 1] – penalty,
                                            temp[t_index_y – 1][t_index_x] – penalty );
   }

    barrier(CLK_LOCAL_MEM_FENCE);
}

for (int m = BLOCK_SIZE - 2; m >= 0; m--) {

   if (tx <= m) {
      int t_index_x = tx + BLOCK_SIZE - m;
      int t_index_y = BLOCK_SIZE – tx ;

      temp[t_index_y][t_index_x] = maximum( temp[t_index_y – 1][t_index_x – 1] +
                                            ref[t_index_y – 1][t_index_x – 1],
                                            temp[t_index_y][t_index_x – 1] – penalty,
                                            temp[t_index_y – 1][t_index_x] – penalty );
   }

    barrier(CLK_LOCAL_MEM_FENCE);
}
```

6.3.1.3 Optimization Efforts and Results

1) Execution Geometry

a) *Geometry of work-items*

In this category, two optimization parameters are examined in *needle_opencl_shared_1* kernel. In the first one, the BLOCK_SIZE of initial implementation that has size 16 changed to 32 and in the second case it changed to 64. It wasn't feasible to collect all the execution times of all the devices during the afore-mentioned changes due to the local and global memory limitations that the latter have.

The result is that generally the use of larger BLOCK_SIZE enhances the performance in AMD GPU and degrades the performance in NVIDIA GPU. In Intel CPU the performance doesn't change for all the combinations of work-group size and sequence lengths that used.

Figure 6.37 and *Figure 6.38* depict the results for sequence length 4096, 8192 and 16384. The execution times represent the total execution time of kernel including all the kernel calls.

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	48,643,800 ns	75,390,113 ns	6,841,248 ns
BS 16 to 32	49,030,091 ns	46,194,110 ns	12,490,464 ns
BS 16 to 64	*insufficient local resources*	*insufficient local resources*	29,570,672 ns

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
BS 16 to 32	1	1.63	0.55
BS 16 to 64	-----	-----	0.23

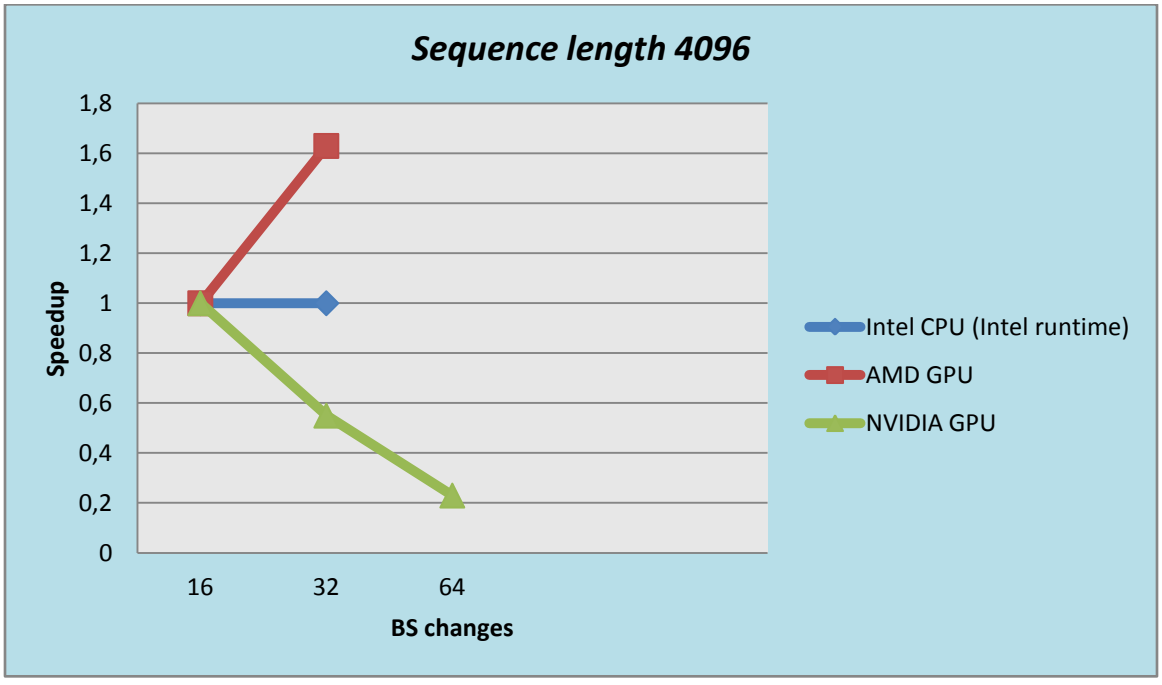
Optimizations / Devices (8192)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	189,728,472 ns	*out of bounds device memory*	23,510,992 ns
BS 16 to 32	176,210,298 ns	*out of bounds device memory*	39,227,440 ns
BS 16 to 64	*insufficient local resources*	*out of bounds device memory*	105,930,064 ns

Optimizations / Devices (8192)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
BS 16 to 32	1	-----	0.60
BS 16 to 64	-----	-----	0.22

Optimizations / Devices (16384)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	678,118,668 ns	*out of bounds device memory*	*out of bounds device memory*
BS 16 to 32	661,343,873 ns	*out of bounds device memory*	*out of bounds device memory*
BS 16 to 64	*insufficient local resources*	*out of bounds device memory*	*out of bounds device memory*

Optimizations / Devices (16384)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
BS 16 to 32	1	-----	-----
BS 16 to 64	-----	-----	-----

Figure 6.37: *Execution times and speedup of optimization 'Geometry of work-items' in needle_opencl_shared_1 kernel for sequence length 4096, 8192 and 16384*



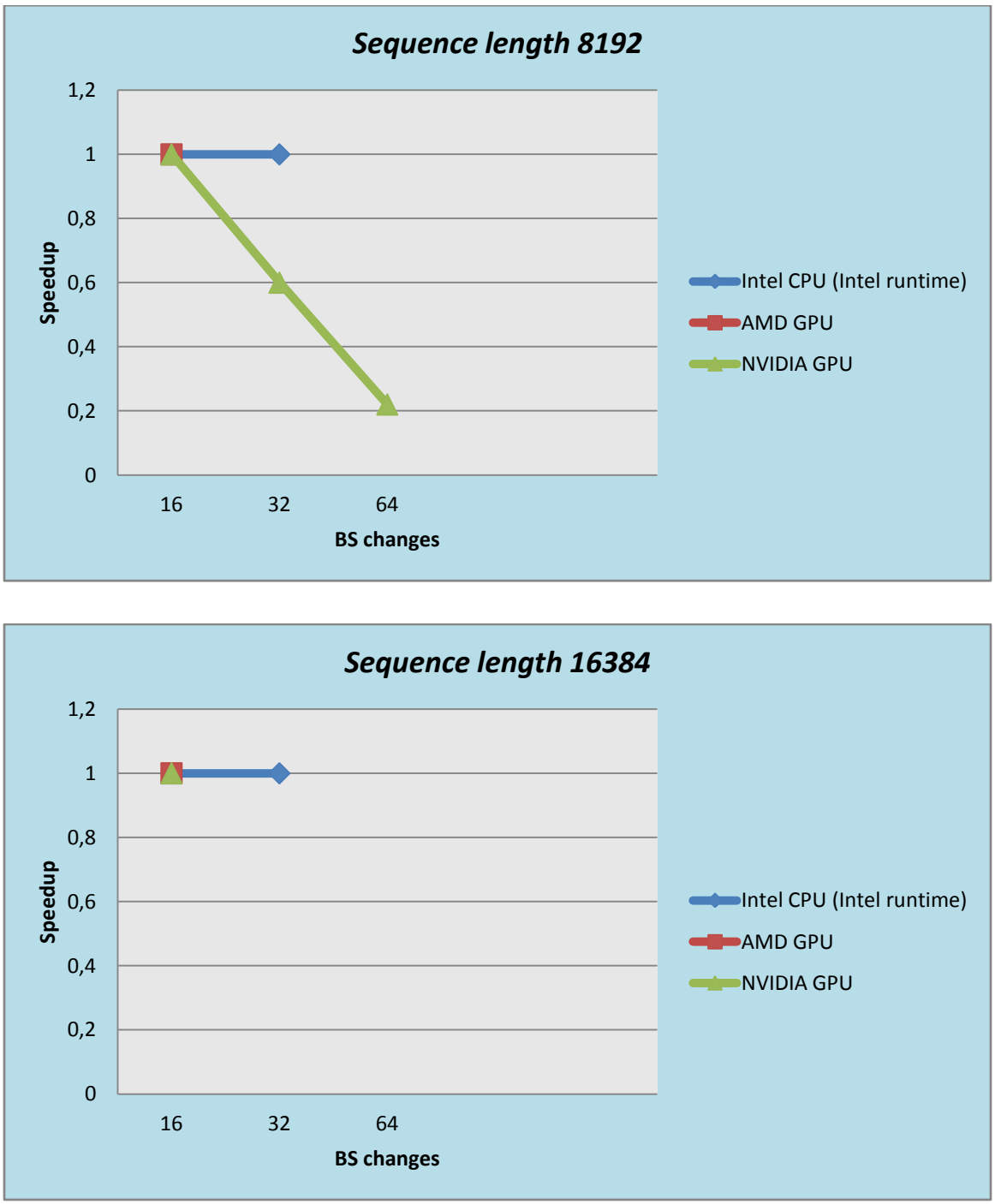


Figure 6.38: *Execution times and speedup of optimization 'Geometry of work-items' in needle_opencl_shared_1 kernel for sequence length 4096, 8192 and 16384*

The reason that the *BS 16 to 32* case improves significantly the performance in AMD GPU is that the product *total kernel calls * execution time per kernel call* is smaller when work-group size 32 is used. It is characteristic that execution time per kernel is higher in the case of BS 32, as the number of average ALU instructions executed per work-item increases, but this adds a small overhead compared to the times that this kernel is called.

On the other hand, in NVIDIA GPU the pre-referred overhead is much higher compared to the initial implementation and gradually increases as the work-group size increases. This can be attributed to the fact that NVIDIA GPU does not have vector processors as AMD does and the added more workload per work-item proves detrimental. Moreover, it is important to note that for all the values of BLOCK_SIZE, the total number of warp/wavefront divergencies per kernel remains the same for NVIDIA/AMD GPU respectively.

Another important point to mention is that for BS 64 in this kernel, AMD GPU would utilize with the best manner its hardware if its local and global memory limitations let it do that, while NVIDIA GPU does that for BS 32. For BS > 32, NVIDIA GPU executes more parts of its work-groups serially and this degrades much the performance, beyond of the fact that the workload per work-item increases as the work-group size increases.

In Intel CPU, although totally fewer work-groups per kernel call are used when the BLOCK_SIZE increases and this causes fewer context switches that benefits the CPUs, the large computation overhead per work-item has as a result the performance to remain steady through all work-group size changes.

6.3.1.4 Unfeasible Optimizations

1) Execution Geometry

b) *Granularity*

This optimization *Geometry* sub-category wasn't feasible to be implemented here, as the workload per diagonal isn't the same and so there is not a safe manner to add uniformly more work per work-item. This optimization would be feasible if the initial code padded to a form where every diagonal would have the size of main diagonal of work_group size that used.

2) Vectorization

Vector Types

Vectorization optimization wasn't feasible to be implemented here for the same reasons that *Granularity* optimization doesn't. Usually, the latter optimization forms the appropriate conditions so as the first can be applied. In case that this doesn't happen, vectorization can't be implemented too.

3) Loops

Loop Unrolling

Loop Unrolling optimization technique cannot be applied effectively in *needle_opencl_shared_1* kernel due to the serial form of work-items execution and the existence of synchronization points among the transition from one diagonal to another. However, it is tested in the kernel, but the performance remained in the same levels, as expected, in all the devices that used.

6.4 srad

Srad belongs to the structured-grid applications. These applications organize their data in a multidimensional grid. They perform a series of calculations for each element using the neighborhood around them.

6.4.1 Analysis of *srad_cuda_1* kernel

6.4.1.1 Data Dependencies

At first this kernel loads one segment of the matrix in the global memory to a smaller matrix in the local memory for faster computations. It uses 2-dimensional work-groups in order to suit better the 2-dimensional arrays. Then each work-item of a work-group calculates four values using the green points around the red one in the matrix, called this the “east”, “west”, “north” and “south” values as shown in *Figure 6.39* that are stored in four matrices in the global memory. After composing these four values of the four green points the new value is calculated and this value is stored in the output matrix in the global memory too.

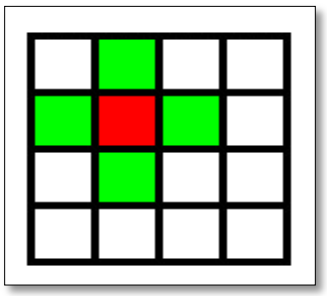


Figure 6.39: *Computational pattern of srad_cuda_1 kernel*

Calculating values in such structures usually leads to a different treatment of the points in the edges of the matrices. For example there is not a “north” element for the elements that are in the first row. So, in the current algorithm there are four more matrices allocated in the local memory, each one for the “west”, “east”, “north” and “south” elements of a matrix. These matrices are filled with the corresponded values needed and are used only for the elements in the edges.

6.4.1.2 Basic code segment

```
temp[ty][tx] = get the corresponding element of the matrix in the global memory

If(element==in the edge of the matrix){
        set the right matrix “south”, “north”, “east” or “west”
}

jc = temp[tx][ty];

If(element==in the edge of the matrix){
        Special treatment with the corresponding matrix “south”, “north”, “east” or “west”
}
else{
   n  = temp[ty-1][tx] - jc;
   s  = temp[ty+1][tx] - jc;
   w  = temp[ty][tx-1] - jc;
   e  = temp[ty][tx+1] - jc;
}

Store the n, s, w and e to the corresponding matrices in the global memory
Calculate the value from the n, s, w and e and store it to output matrix in the global memory
```

6.4.1.3 Optimization Efforts and Results

1) <u>Padding</u>

In many-core computing it is often better to perform redundant work, even if the results are to be discarded, rather than to suffer the overhead of branches. This is more obvious in GPUs because of their parallel throughput and their sensitivity to divergent execution of threads within the same warp/wavefront. Branches usually have a negative effect on the execution time of a code. CPUs

sometimes use branch prediction algorithms to improve the performance. However, the problem of branches is more obvious in GPUs because of the divergencies.

The computational pattern of the srad requires many branches in order to separate the inner values from the values in the edges. The *srad_cuda_1* kernel also separates the inner points from the others creating a large number of branches. This means more divergence for the GPUs and more context switches for the CPUs and more instructions to be executed for both a CPU and a GPU.

Our first goal for the current kernel was to eliminate all the branches. A technique called padding is used for such algorithms. The padding is shown in *Figure 6.40*. At first, the suitable size of the matrix must be allocated. The new size depends on the original size of the matrix and the depth of the neighborhood. One important thing in padding is that the same number of the work-items with the original code must be kept.

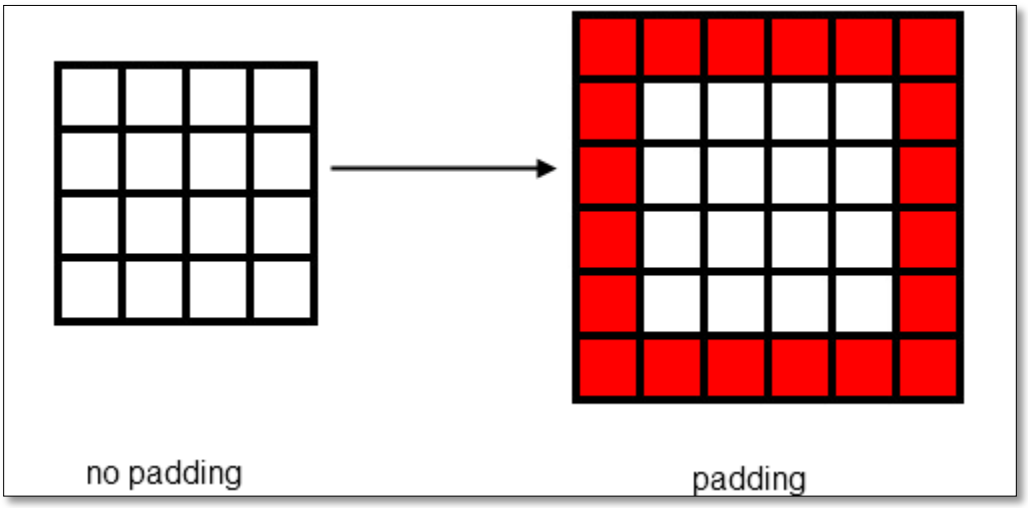


Figure 6.40: *The padding technique*

In our example we suppose that the neighborhood has depth 1. So the new size of the matrix is (size+2) x (size +2). The white elements in the matrix are the points of the initial matrix and the reds the points needed for the computation for those in the edges. Instead of having four more matrices the “north”, “south”, “east” and “west”, the values needed for the points in the edges are stored in the red area. Now, all the points are treated in the same way and there is no need for branches.

The elimination of the branches led to less instruction for execution by all the architectures. The instructions were decreased to half of the initial code. Moreover the cache hit rate was increased by 10%. The CPU also had fewer context switches. The GPUs stopped to suffer divergencies because of the branches overhead. The divergence is one of the biggest problems for a GPU because all the work-items of a warp or a wavefront must execute the same instructions in a given time. If halves of the work-items of a warp or a wavefront execute a different branch from the other halves of work-items then all these work-items cannot be executed concurrently and their halves have to stall. Moreover, the ALUPacking metric was increased from 40 to 54 for the Cayman.

Although the GPUs execute a code in the same way in general, there were different speedups for the NVIDIA and the AMD. This happens because of the different sizes of a wavefront and a warp. A wavefront has 64 work-items and a warp 32. Fewer work-items in a warp result usually in fewer

divergences per warp in a code. So, eliminating all the branches the AMD had a better gain than the NVIDIA.

Figure 6.41 and *Figure 6.42* depict the results for matrix dimension 4096 and 8192.

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	0.233 sec	0.036 sec	0.0056 sec
padding	0.067 sec	0.018 sec	0.0040 sec

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
padding	3.48	2	1.4

Figure 6.41: *Execution times and speedup of optimization 'Padding' for matrix dimension 4096*

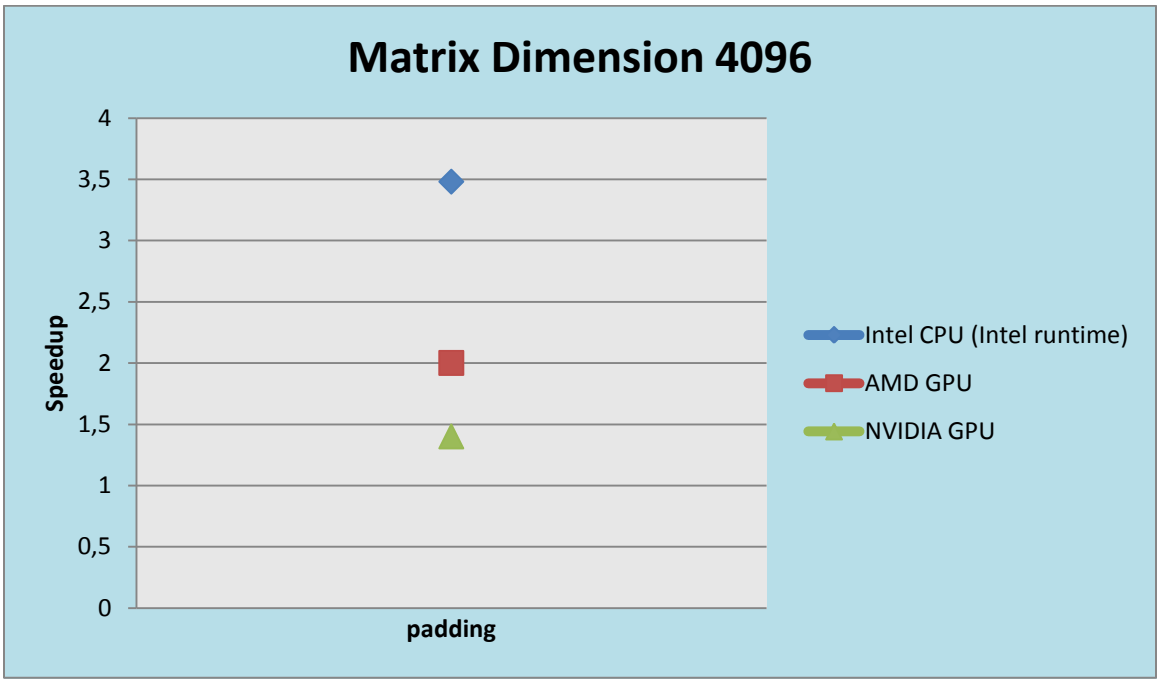


Figure 6.42: *Speedup graphical representation of optimization 'Padding' in srad_cuda_1 kernel for matrix dimension 4096.*

One more advantage of the padding in the current kernel is that there is no need of the matrices for the red elements. These matrices were allocated in the local memory and the use of this memory was decreased from 6144B to 2432B leading to fewer cache misses. Also more useful data can be stored in the local memory such as a larger padded matrix.

2) Execution Geometry

Geometry of work-items

In this section, we tried to find the optimal BS. The work-groups are 2-dimensional with BS size each dimension. The initial BS was 16, meaning that there were 256 work-items per work-group.

We first tested the CPU changing the BS from 16 to 32 having a total number of 1024 work-items per work-group ,which is the maximum number of work-items per work-group we can get for this architecture. The new execution time was slightly slower and maximizing the size of the work-group did not give any gain for this pattern. However, there was not a big loss either, meaning that the size of the work-group is not a significant point for the *srad*.

Next, we tested the NVIDIA GPU. The results were different from the CPU. NVIDIA allows up to 1024 work-items per work-group, but also each SM can have up to 1536 work-items. So, the number of the work-items per work-group must be an integer divisor of the maximum number of work-items per SM. Also the total number of the work-groups should not be more than 8 [15]. The result of these is that the optimal numbers of the work-items are 192, 256, 384, 512 and 768. The initial code had 256 work-items and changing the BS to 32 we had 1024 work-items, which is not an optimal number. That is why the result of the 1024 work-items is worse. One more factor that the BS 32 was worse is the number of the registers per work-item. There are 20 registers per work-item. Work-groups with 1024 work-items and 20 registers per work-item suffer a loss of performance.

Finally we tried to test the AMD GPU but this was not feasible. The Cayman supports up to 256 work-items per work-group and we were already at the maximum number of them.

Figure 6.43 and *Figure 6.44* depict the results for matrix dimension 4096 and 8192.

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	0.067 sec	0.019 sec	0.0040 sec
BS 16 to 32	0.074 sec	-----	0.0058 sec

Optimizations / Devices (4096)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
BS 16 to 32	0.9	-----	0.69

Figure 6.43: *Execution times and speedup of optimization 'Geometry' in srad_cuda_1 kernel for matrix dimension 4096*

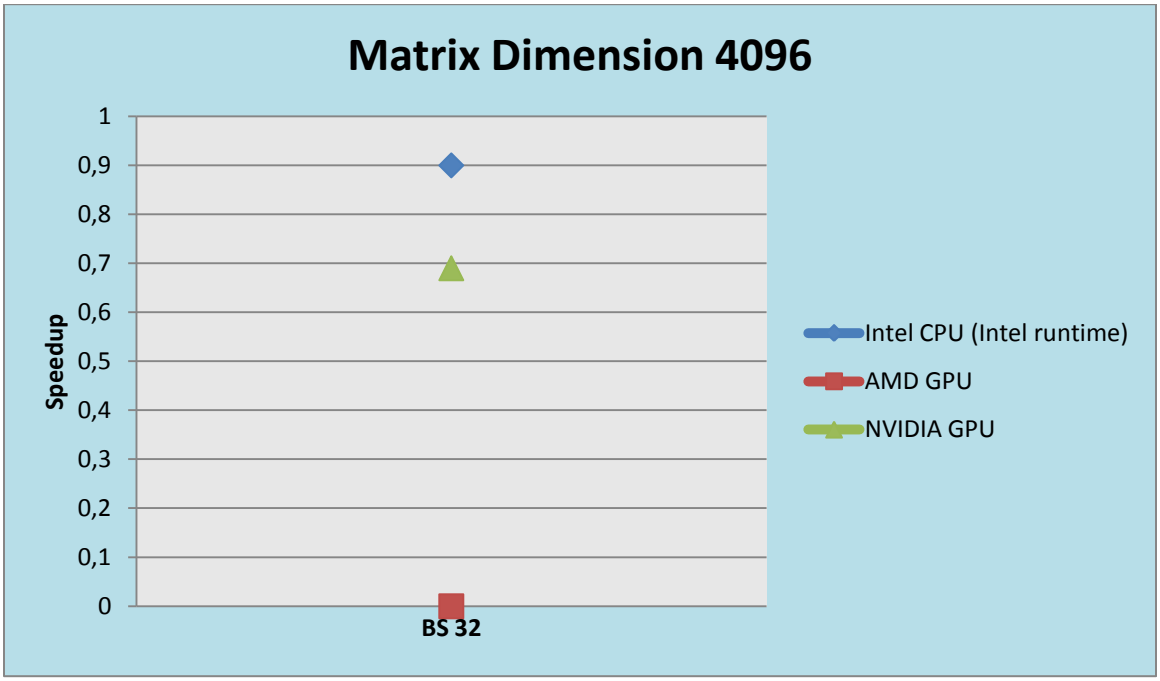


Figure 6.44: *Speedup graphical representation of optimization 'Geometry' in srad_cuda_1 kernel for matrix dimension 4096.*

6.4.1.3 Unfeasible Optimizations

The current kernel does not have any loops and optimizations such as *loop unrolling* and *loop fission* could not be examined.

6.5 bfs

The *bfs* belongs to *graph-traversal* applications which calculate the cost of the nodes of a graph. The inputs are the number of edges and the nodes that visit each node.

6.5.1 Analysis of *kernel1* kernel

6.5.1.1 Data Dependencies

The current application uses an one-dimensional array of structs to keep the data of each node, one more one-dimensional array to keep the edges of each node and one last one-dimensional to

keep the cost of each node. Also there are created as many work-items as the number of the nodes. Each work-item calculates the cost of each node. At first, a max number (*max*) of each node is calculated, which is the sum of the first visited (starting) node of this node and the number of its edges (*num_of_edges*). Then a loop of (max – starting) iterations is executed checking all the visiting nodes of the current node increasing their cost. For our experiments we used the *graph65536.txt* file. *Figure 6.45* depicts the data dependencies pattern of *bfs*.

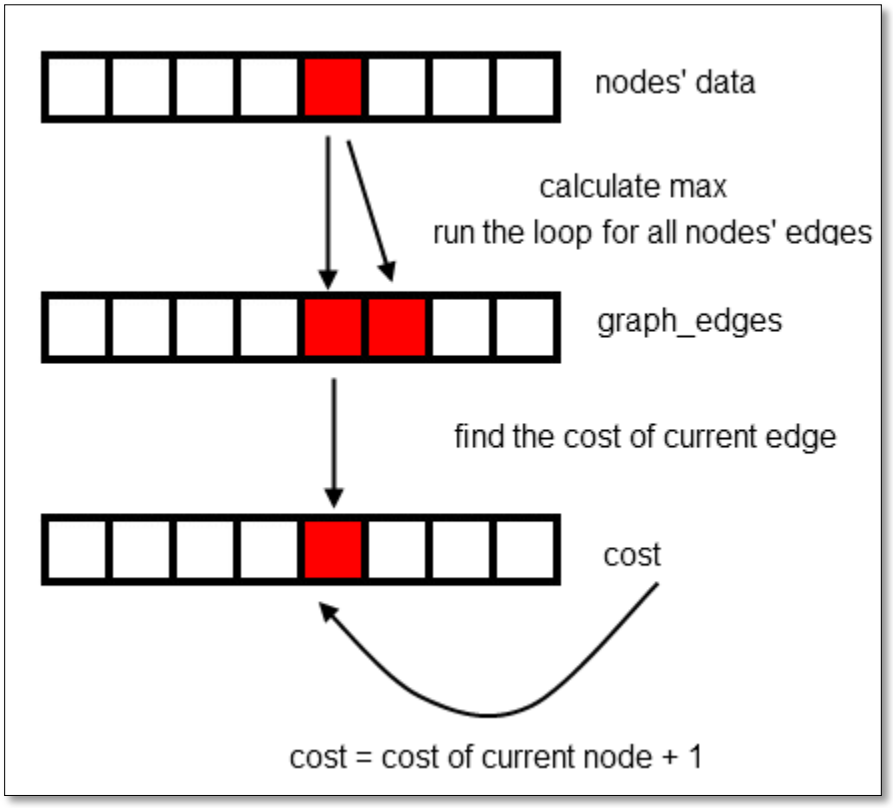


Figure 6.45: *Data dependencies pattern of 'bfs'*

6.5.1.2 Basic code segment

```
tid = get_global_id(0);
max = (g_graph_nodes[tid].no_of_edges + g_graph_nodes[tid].starting);

for(int i = g_graph_nodes[tid].starting; i < max; i++)
{
        id = g_graph_edges[i];
        if(!g_graph_visited[id])
        {
                g_cost[id] = g_cost[tid] + 1;
        }
}
```

6.5.1.3 Optimization Efforts and Results

This kernel uses a one-dimensional work-group. The number of the work-items per work-group depends on the number of the total nodes. If the total nodes are fewer than the maximum number of the work-items supported per work-group for the current architecture then the work-group size is equal with the number of the nodes, otherwise it is equal with the maximum number of the work-items per work-group. This is calculated by the following formula:

*clGetDeviceInfo(device_id,CL_DEVICE_MAX_WORK_ITEM_SIZES,sizeof(size_t)*3,&maxThreads, NULL);*

maxThreads[0] = no_of_nodes < maxThreads[0] ? no_of_nodes : maxThreads[0];
size_t localWorkSize[1] = {maxThreads[0]};

And the total work-items are calculated by:
size_t WorkSize[1] = {no_of_nodes + (no_of_nodes%maxThreads[0])};

a) *Granularity*

We tested two different cases for the coarse grain. The first case was to divide the number of the work-items per work-group by a factor of 2 and the second one was to divide the number of the work-groups by a factor of 2. In both cases we had the same number of the total work-items but the geometry between the 2 cases was totally different.

1st case:

In this case we divided the number of work-items per work-group by a factor of 2. The Cayman's performance was better than the performance of the initial code again. More specifically, the ALU instructions were increased almost twice compared to the initial implementation, but the total number of them was lower. For example, the total number of the work-items was 32768 while in the initial code was 65536. In the first kernel invocation there were 17 ALU instructions per work-item for this case and 9 for the initial code. So, we now had 564,592 instructions in total, while the initial code had 597,688. Also the total number of the wavefronts was divided by a factor of 2 from 1024 initially to 512.

On the other hand, the Intel CPU and the NVIDIA GPU did not get any gain from that and their execution times were very similar to the initial ones.

Figure 6.46 and *Figure 6.47* depict the results for 65536 nodes.

Optimizations / Devices (65536 nodes)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	0.024 sec	0.056 sec	0.000713 sec
Size of Work-groups/2	0.024 sec	0.041 sec	0.000721 sec

Optimizations / Devices (65536 nodes)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Size of Work-groups/2	1	1.37	0.98

Figure 6.46: *Execution times and speedup of optimization "Granularity" with size of work-groups divided by 2*

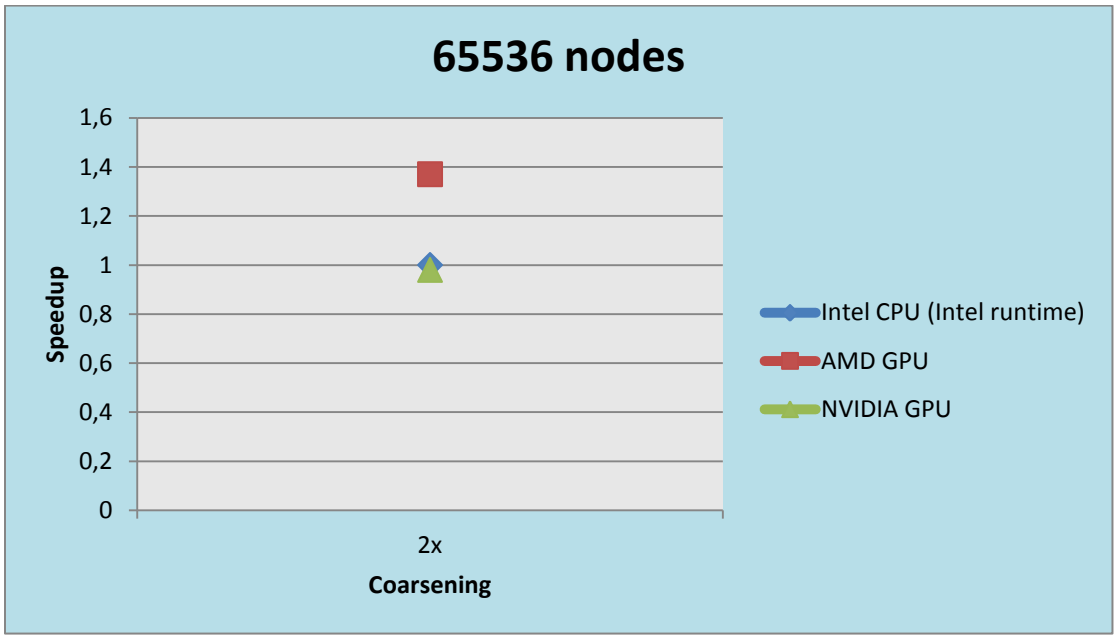


Figure 6.47: *Execution times and speedup of optimization 'Granularity' for bfs for 65536 nodes*

2nd case:

In this case we kept the same size of the work-group with the initial code but we had the halves of them. The results for this case were the same with the *1st case*. The low level metrics were too similar compared to the 1st case thus we had the same results.

Figure 6.48 and *Figure 6.49* depict the results for 65536 nodes.

Optimizations / Devices (65536 nodes)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Initial Code	0.024 sec	0.056 sec	0.000713 sec
Number of Work-groups/2	0.024 sec	0.041 sec	0.000721 sec

Optimizations / Devices (65536 nodes)	Intel CPU (Intel runtime)	AMD GPU	NVIDIA GPU
Number of Work-groups/2	1	1.37	0.98

Figure 6.48: Execution times and speedup of optimization "Granularity" with number of work-groups divided by 2

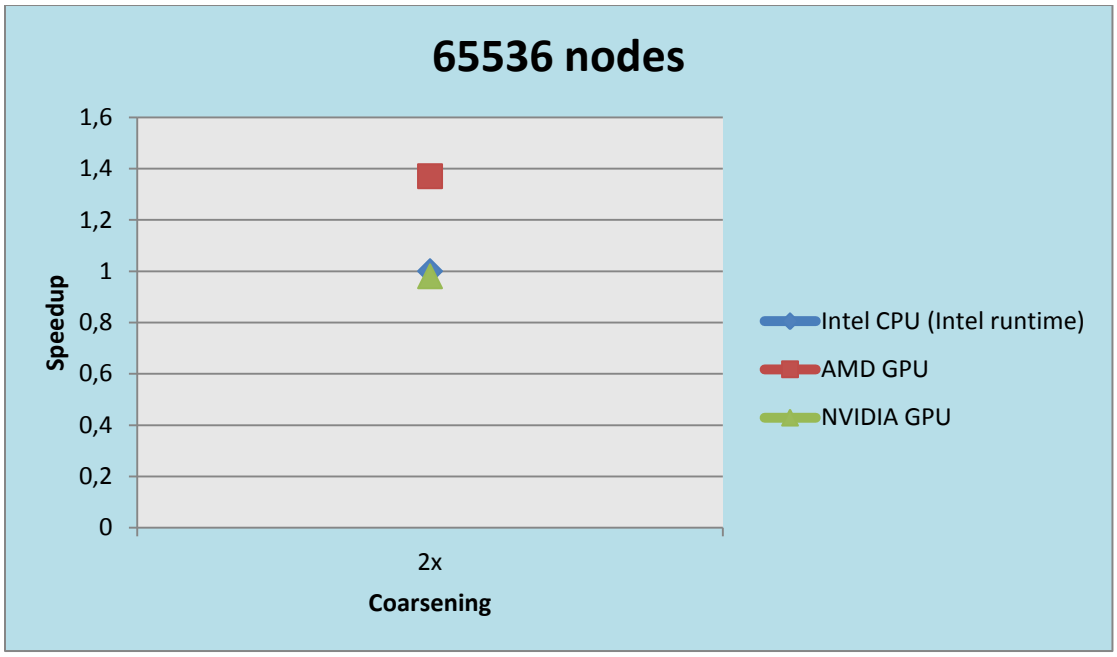


Figure 6.49: Execution times and speedup of optimization 'Granularity' for bfs for 65536 nodes

b) *Geometry of work-items*

In this section we tested different sizes of the work-group. The size of the work-groups was initialized to the maximum number of each architecture. So, for the Intel and the NVIDIA architectures the size of the work-groups was 1024 and for AMD 256. We tested the sizes of 128, 256, 512 and 1024. There was not a significant improvement for any of the architectures and the execution times were almost the same with the initial ones. There was not a difference in the low level metrics too.

Figure 6.50 and *Figure 6.51* depict the results for 65536 nodes.

Optimizations / Devices (65536 nodes)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
BS 128	0.023 sec	0.052 sec	0.000659 sec
BS 256	0.024 sec	0.056 sec	0.000698 sec
BS 512	0.022 sec	-----	0.000689 sec
BS 1024	0.023 sec	-----	0.000713 sec

Optimizations / Devices (65536 nodes)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
BS 256	0.95	1.07	0.94
BS 512	1.05	-----	0.96
BS 1024	1	-----	0.92

Figure 6.50: *Execution times and speedup of optimization "Geometry of work-items"*

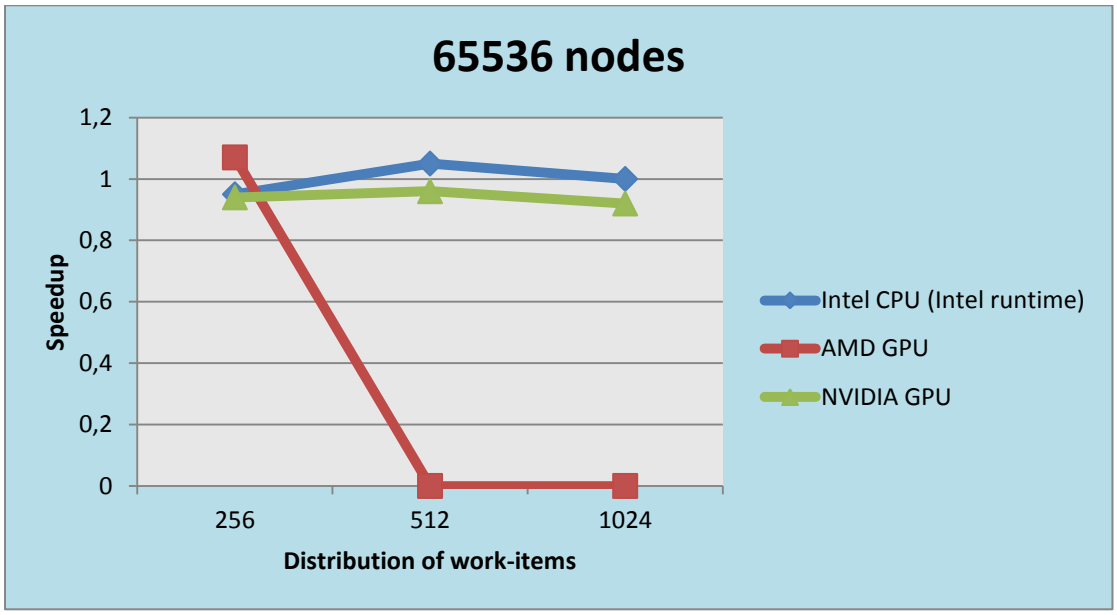


Figure 6.51: *Execution times and speedup of optimization 'Geometry of work-items' for bfs for 65536 nodes*

2) Loops

Loop Unrolling

The loop unrolling optimization was applied in this section, but this could not be done directly. As it was said previously the number of the iterations for each node is not constant for all the work-

items. What is more, we cannot know if the number of the iterations is an odd or an even number. So, we had to check all the possible cases, creating branches and increasing the complexity of the code. The branches resulted in more instructions for execution in all the architectures and in more divergences for the two GPUs. As a result, there was not any improvement for any of the architectures. We only tested the loop unrolling by a step of 2 because for bigger steps the code was even more complex.

Figure 6.52 and *Figure 6.53* depict the results for 65536 nodes.

Optimizations / Devices (65536 nodes)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Initial Code	0.024 sec	0.056 sec	0.000713 sec
Unrolling	0.031 sec	0.064 sec	0.000756 sec

Optimizations / Devices (65536 nodes)	**Intel CPU (Intel runtime)**	**AMD GPU**	**NVIDIA GPU**
Unrolling	0.77	0.86	0.94

Figure 6.52: *Execution times and speedup of optimization "Loop Unrolling"*

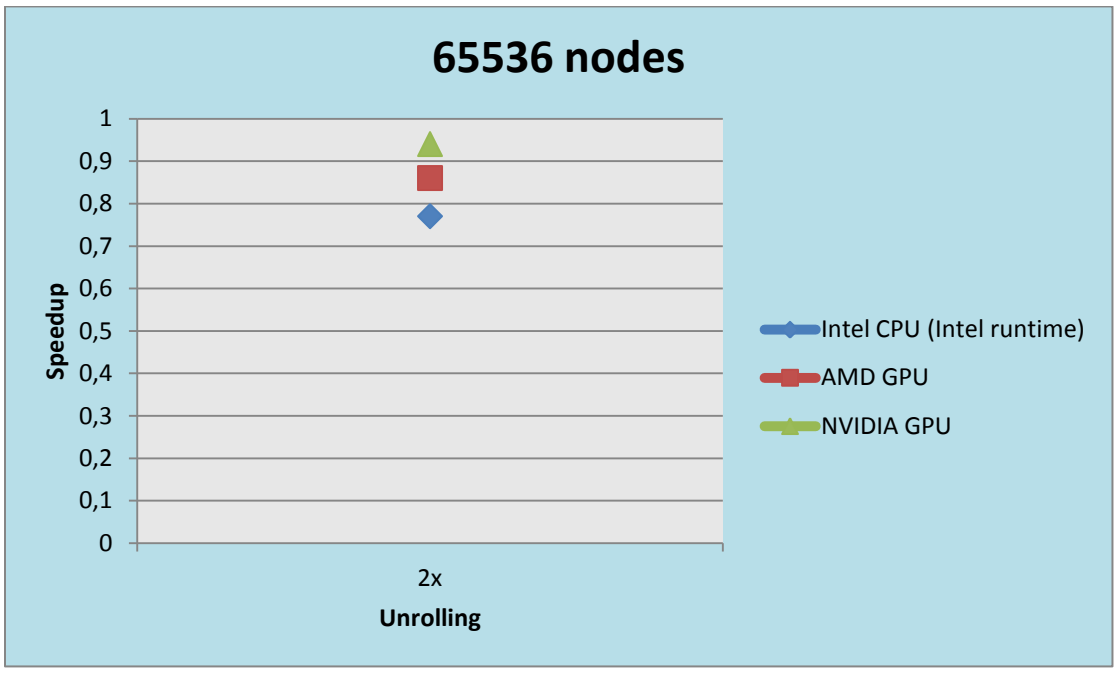


Figure 6.53: *Execution times and speedup of optimization 'loop unrolling' for bfs for 65536 nodes*

6.5.1.4 Unfeasible optimizations

We could not apply vectorization because of the if-statement that checks if the for-loop should be executed. In order to apply vectorization we should use the component of each vector unit and make the calculations in each component separately.

Chapter 7

Conclusion

The optimization categories that examined in this thesis are a) *Geometry*, b) *Vectorization*, c) *Loops* and d) *Branches*. The implementations of the Berkeley dwarfs on which they are applied are the applications *lud*, *crc*, *needle*, *srad* and *bfs*. The basic conclusions of our study are the following:

1) <u>Geometry</u>

Larger coarsening benefits significantly the Intel CPU, as the total number of work-groups reduces and this results to fewer context switches which cost much to CPUs (*lud_perimeter*). Instead, the afore-mentioned reduction degrades the performance of GPUs, as it causes great load imbalance within cores of warp/wavefront execution units of NVIDIA/AMD GPU respectively (*lud_perimeter*). Generally we can say that this results to fewer work-items per work-group, namely limited opportunities to hide memory latency), or fewer work-groups which, after a point, will result to inability to exploit all streaming multiprocessors of the GPU. Moreover, this load imbalance is bigger in AMD GPU, as its wavefronts comprise 64 work-items, while NVIDIA warps comprise 32 work-items (*lud_perimeter*, *crc*). Moreover, the reduction of context switches has greater impact on speedup gain for bigger data sizes on kernels that executed iteratively and their work-items execute little instructions (*lud_perimeter*). However, larger coarsening can help AMD GPU to perform better in cases, like *crc*, where can exploit the hardware FastPath instead of using its ineffective CompletePath.

In applications where each work-item has many instructions to execute (*lud_perimeter*, *needle*), an optimization that increases this workload degrades more the performance of GPUs rather than that of CPU. This happens, as Intel CPU work-items have higher clock frequency and can execute faster a heavy serial code. However, in some cases, like *crc*, *bfs*, heavy workload per work-item and few work-items per kernel are not bad for both CPUs and GPUs (especially for AMD GPU), as the product *total number of work-items per kernel * average number of ALU instructions executed per work-item* can prove beneficial. This means that although the ALU instructions increase, the number

of work-items decrease significantly and therefore the total number of ALU instructions executed per kernel is less.

Another interesting case is when the performance of Intel CPU remains steady (*crc*, *needle*), although context switches are reduced. This happens only in kernels that the overhead is defined by the many instructions that a work-item must execute and the overhead of context switches is minor compared to the computation overhead per work-item. Additionally, we also observed cases in iteratively called kernels, like *needle*, where the product *total kernel calls* * *execution time per kernel call* gets smaller as work-group size increases and this benefits the AMD GPU and degrades the performance of NVIDIA GPU, although the number of wavefront/warp divergencies remains constant during the work-group size change. This holds, since NVIDIA doesn't have vector processors to execute fast their work-item instructions and the overhead of execution time per kernel is higher than the one of reduced number of kernel calls that work-group size change offers. Instead, AMD GPU can take advantage of its VLIW architecture and the overhead of increase of ALU instructions which defines the execution time per kernel is minor compared to the one of reduced number of kernel calls.

Last but not least, it is observed that there should not be less work-groups than the number of the cores of the CPU, because some cores will be idle. Having more work-items in a work-group and a few work-groups increases the performance of the applications that use the local memory such as the *internal kernel* of the *lud*. More work-items sharing the same local memory means less cache misses because each core has a separate local memory and the work-items reuse the data in cache in such patterns. Each work-group is executed in one only core thus a few work-groups with many work-items offer less cache misses. In other applications such as *bfs* that did not give any gain but we can infer that it is a useful optimization because it did not worsen the performance in any application.

2) <u>Vectorization</u>

Vectorization is a technique that allows two or more adjacent elements in a consecutive memory area to be executed concurrently as it enables, using the vector types that each architecture provides, a group of them to be executed in the same clock cycle. The size of group depends from the degree of vectorization that implemented. Many architectures adapt this technique to increase their performance. The Intel CPU and the AMD GPU has vector type units to perform this action, while the NVIDIA GPU does not have any.

Therefore, in cases like *lud_perimeter* kernel of *lud*, the Intel CPU can make effective use of vectorization on float types and the implementation degrees of 4 and 8 yield the best performance. Generally, vectorization is one of the main optimizations used in high performance computing and can give a speedup in patterns like calculating the inner product of two vectors or adding the elements of two arrays, but in some cases it is difficult to use it in an application, like in *lud_internal* kernel, because it demands a consecutive memory area. The application may need a different treatment in order to apply the vectorization, killing the performance sometimes. Nevertheless, it is recommended for use when the application allows it, like *lud_perimeter* kernel as we said.

3) Loops

The optimization technique of loop unrolling was tested. It is useful to unroll the loops in order to exploit better the register file of each device, schedule more efficiently the instructions and utilize the pipelines that almost all the architectures offer, so as some instructions in the loop to be executed in parallel. However, the testing of the loop unrolling on the *internal kernel* of the *lud* and on the *bfs* doesn't give any speedup for any of the platforms, but there was not a loss of performance too. Unrolling a loop does not always give a gain but it can be applied on the code because we usually expect to have speedup.

The cases that exhibit the benefits of loop unrolling are the kernels *lud_perimeter* and *compute* of applications *lud* and *crc* respectively. In the first kernel loop unrolling creates serially dependent instructions and AMD GPU exploits its VLIW pipelines and this results to significant reduction of the number of ALU instructions that are executed per work-item. In our experiments performance improves as the degree of unrolling increases. In the second kernel, there are some independent instructions among dependent ones and AMD VLIW architecture can make a more effective use of its VLIW4 execution units.

Additionally, NVIDIA in the first case can exploit much better the vast register file that possesses, while in second kernel, the branches of if statements hinder it from utilizing effectively its cumulative register capacity. Instead, Intel CPU performs much better in the second case, as the presence of some independent instructions among dependent ones helps it to take advantage of ILP. This can't happen in the first kernel, since all the instructions that the latter has, after loop unrolling implementation, are sequentially dependent.

4) Branches

The branches were tested on the *srad* dwarf. The initial code had many branches. When the branches were eliminated there was a speedup for all the platforms. Eliminating them all, there was not any divergence for the GPUs, which is one of the biggest problems decreasing their performance significantly. The CPU had a better performance because there were fewer instructions to be executed and less context switches. The elimination of the branches led to less context switches among the work-groups. Generally, branches can slow down the execution time of an application. Moreover, in architectures like GPUs, it is preferable to have more computations than branches and in some cases this is preferable for a CPU too.

The future aim of this thesis is to apply more optimization techniques, such as fixed point arithmetic, texture memory interpolation, associative cache and others, on the afore-mentioned and on more computation and communication patterns (the rest of the 13 OpenCL Berkeley dwarfs). The basic goal is to identify which computational patterns better match which architectures via which optimizations so as the performance can be portable among different architectural devices and revealing which accelerator device is most suitable to execute a certain computational task. Through

this thesis, several very important aspects of the above themes have been illustrated in practice with experimental results and justifications, but in order one to be able to generalize the benefits that heterogeneous systems can offer, more applications and more optimizations are needed to be evaluated.

References

[1] Khronos OpenCL Working Group, “The OpenCL Specification (version 1.1)”, 2011

[2] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung, Dan Ginsburg, “OpenCL Programming Guide”, 2011

[3] M. Flynn, “Some Computer Organizations and Their Effectiveness”, in IEEE Trans. Comput., C-21:948+, 1972

[4] W. Feng, H. Lin, T. Scogland, J. Zhang, “OpenCL and the 13 Dwarfs: A Work in Progress”, in ICPE ’12 Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, 2012, pp 291-294

[5] Sean Rul, Hans Vandierendonck, Joris D’Haene, Koen De Bosschere, “An Experimental Study on Performance Portability of OpenCL Kernels”, in 2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC ’10), Knoxville, TM, USA, 2010

[6] Jarno van der Sanden, “Evaluating the Performance and Portability of OpenCL”, Master thesis under supervision of H. Corporaal and C.Nugteren, Electonic Systems Group-Faculty of Electrical Engineering, Eindhoven University of Technology, 2011

[7] Joo Hwan Lee, Kaushik Patel, Nimit Nigania, Hyojong Kim, Hyesoon Kim, “OpenCL Performance Evaluation on Modern Multi Core CPUs”, in Multicore and GPU Programming Models, Languages and Compilers Workshop (PLC 2013), 2013

[8] D. Grewe, M. F. P. O’Boyle, “A static task partitioning approach for heterogeneous systems using OpenCL”, CC 2011:286-305

[9] Intel Corporation, “Intel Xeon Processor 5600/5500 Series Platforms for Embedded Computing”, 2010

[10] http://ark.intel.com/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI

[11] NVIDIA Corporation, “Whitepaper: NVIDIA’s Next Generation CUDA Compute Architecture Fermi”, 2009

[12] David Kanter, “Real World Technologies: AMD’s Cayman GPU Architecture”, 2010

[13] Wiki Answers, “What is dark silicon”, http://wiki.answers.com/Q/What_is_dark_silicon

[14] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Pattrerson, William Lester Plishker, John Shalf, Samuel Webb Williams, Katherine A. Yelick, “The Landscape of Parallel Computing Research: A View from Berkeley”, EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2006-183, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html, 2006

[15] Yuri Torres, Arturo Gonzalez-Escribano, Diego R. Llanos, “Using Fermi architecture knowledge to speed up CUDA and OpenCL programs, in Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10_{th} International Symposium on, pp 617-624

[16] Intel Corporation, “Writing Optimal OpenCL Code with Intel OpenCL SDK”, http://software.intel.com/file/37171