Francisco Paulo de Aboim

Strategies of Assembly of Finite Element Sparse Matrices with GPUs and OpenCL

**Dissertação de Mestrado**

Dissertação apresentada ao Programa de Pós-graduação em Engeharia Civil da PUC-Rio como requisito parcial para

Tecgraf...

Rio de Janeiro, Junho de 2012

Todos os direitos reservados. É proivida a reprodução total ou parcial do trabalho sem autorização do autor, do oridentador e da universidade.

Francisco Paulo de Aboim

Graduou-se em Engenharia Civil na PUC-Rio em 2010...

Agradecimentos

Tecgraf...

Abstract

Aboim, Francisco Paulo de; Martha, Luiz Fernando; Sotelino, Elisa. **A Finite Element Implementation with OpenCL**. Rio de Janeiro, 2012. Master's Dissertation - Department of Civil Engineering, Pontifícia Universidade Católica do Rio de Janeiro.

The paper investigates the implementation of the finite element method using the OpenCL (Open Computing Language). GPU computing is becoming increasingly popular because of its ability to efficiently perform highly parallel computations, such as vector and matrix operations. Current GPUs have approximately 100 times more potential computing power than a CPU but achieving this potential requires an understanding of the hardware and how it affects the programming model. The massive multicore programming model is also especially well suited for problems of a parallel nature. On the other hand, GPUs are less flexible and require more care and patience in their programming. Furthermore, until recently knowing different programming frameworks was necessary for developing for different hardware vendors. OpenCL was thus made to function as the standard for heterogeneous platforms, using a single framework. Its development is supported by a consortium of major vendors such as Apple, Nvidia, AMD, Intel, ARM, and others. The finite element method is the most widely used method to solve complex computational mechanics and engineering problems. It involves matrix and vector operations in many of its phases and therefore is an excellent candidate for GPU computing. In this investigation, the use of OpenCL is explored on the different stages of the method. Both code implementation and its optimization are discussed. Comparisons with an optimized implementation using a classical parallel processing implementation using OpenMP on a multiprocessor machine are provided.

**Keywords:** GPU computing, OpenCL, High Performance Computing, Parallel Finite Element Analysis

Sumário

List of Figures

1. Background
   1. The CPU Architectures and the Road to Massively Multicore

The motivations of researching new programming models targeted at heterogeneous multicore systems are inherently tied to the motivations that led to the development of multicore systems. Moore's law is well known inside and outside the computing community for having been able to predict with remarkable accuracy back in 1965, in the celebrated and often cited article "Cramming more components onto integrated circuits" [G. Moore, 1965], that transistors on integrated circuits would double roughly every two years. This claim has gained such notoriety because, regardless of silicon price fluctuations, technological breakthroughs and investment in research and development it has held to this day. Not only did Moore predict this trend, but he also was able to envision what would be one of the greatest barriers processor manufacturers would face to keep up with the trend and postpone diminishing returns: heat generation. In the same article Moore poses: "Will it be possible to remove the heat generated by tens of thousands of components in a single silicon chip?”, a question that lingered for decades.

While the growing problem was addressed by cooling solutions which got progressively more complex and expensive, the inherent problem behind heat generation remained. In a few decades passive heat sinks evolved to active heat sinks (including fans to increase the heat exchange), copper heat tubes and a variety of fan designs to optimize air flow (fig).

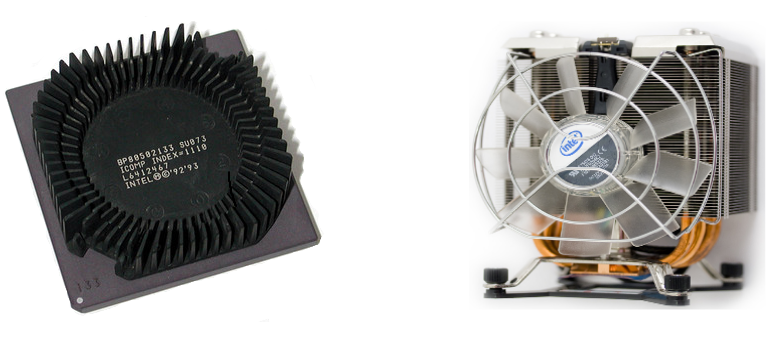


Fig x

We have now reached a point where current processor gates are around 20 nm wide (As in Intel's 22nm Ivy Bridge) or about 74 atoms of silicon wide (with prototypes being even smaller). When dealing with these reduced dimensions, transistor gates have become increasingly susceptible to an effect defined in quantum physics as quantum tunneling. Quantum tunneling tells us that the smaller the barrier is, the greater the probability that an electron will be found on the other side of the barrier. Recently gates have gotten so small that this probability has risen dramatically and the loss of energy due to electrons tunneling through the insulator has become a barrier to further miniaturization, since energy loss translates into an even higher production of heat. The loss of energy due to these effects is commonly named *leakage*. During decades, the processor industry was able to increase processing power by miniaturization and increasing clock frequency. During the last years, however, due to Leakage becoming an increasingly important factor, the industry has hit a frequency wall where further gains in processing power by increasing clock speeds are leading to a higher marginal cost in terms of power consumption (fig). To deal with this, many techniques have been proposed, all of which lead to an increase in complexity, be it in the processor itself, compilers or shifted to the programmer[]. The era of easy gains in algorithm speed coming from advances in the semiconductor industry has come to an end. As Microsoft researcher Herb Sutter put it: “The free lunch is over”. In the burgeoning mobile industry, where efficient use of processing power is paramount and heat dissipation is one of the main concerns. Industry players such as ARM, which have for decades researched how to build efficient processors, have gained a significant amount of market share due to the difficulty other vendors are having in dealing with these setbacks, becoming the largest producers of microchips in the world.

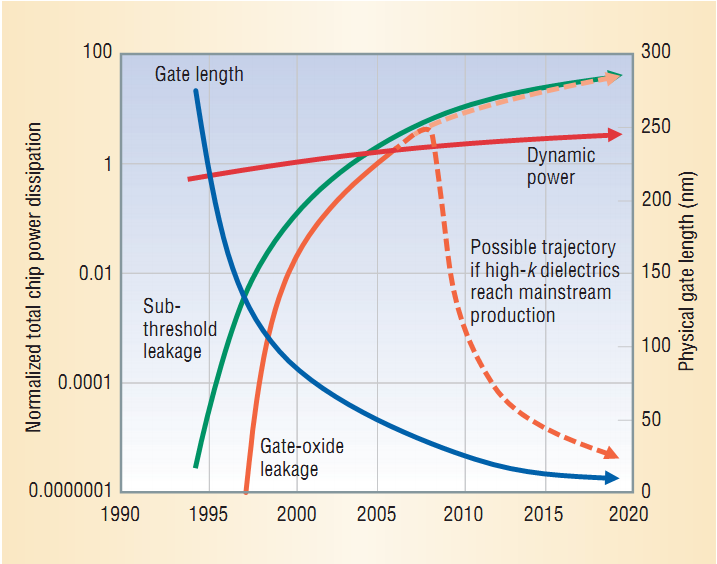


Fig x (Sung Kim, N. et Al.)

While the long awaited replacement for the common variants of the MOSFET (metal-oxide semiconductor field-effect transistor) technology do not arrive and contenders such as the graphene transistor, carbon nanotube transistor and other incipient exotic technologies are still being researched - albeit a long way from mass production - other solutions have been proposed. One of the solutions found to overcome these limitations was to simply pack more cores into a single processor which set the trend of the multicore era. This solution, although obvious, was not implemented at earlier stages because of one problem: almost all production code was designed with the serial programming model in mind and would gain no immediate benefit from the availability of multiple cores.

Such a fundamental hardware change meant that programmers now had to redesign their hardware and algorithms with a different mentality. The burden of having increasing computing performance from a piece of code was now partially passed on to the programmer, who could no longer depend on better compilers and faster processors to gain increased performance. Not only would serial code cease to run faster on newer generations of processors, but the code could actually see its performance degrade, since clock rates stagnated or even fell to be able to accommodate the extra heat produced by the addition of more cores. Still, clock speeds and transistor counts are not the only factors which affect processing power, and to keep up with the growing demand for performance every possible avenue was explored.

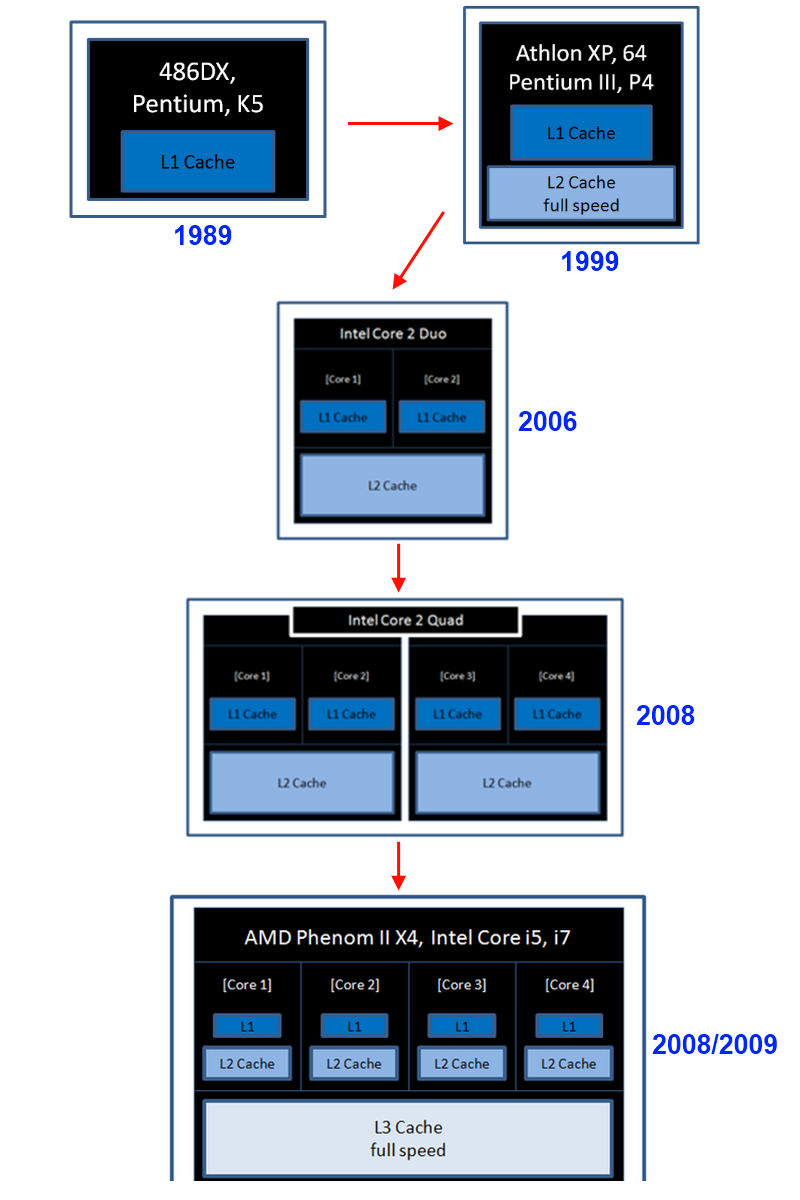
One of the main factors influencing overall performance is the number of instructions per cycle which translates roughly to the amount of work which is done per clock cycle. Unfortunately, the number of instructions per clock has hit its own wall [Stamatis Vassiliadis et al.] and has remained fairly constant in the past years. Not being able to achieve more performance out of instruction level parallelism (ILP), once an oasis for squeezing more performance out of processors, has led to focusing efforts in other areas.

Another imperative factor for driving performance is memory speed. While processing power has kept up with Moore's law as described, memory speed has risen at a slower pace. This has made the spread between processing speed and memory access speed grow larger with time and no new technology was developed which significantly increased memory access speed in a cost effective manner. Moreover, with the addition of more cores to processors communication between cores has increased compared with previous models [Psota, J. et Al]. Due to wire delay, this trend has obviated the necessity of exploring data locality, particularly in more recent years with the rapid increase in numbers of cores. Because of this, many of today's computational algorithms are not actually limited by processing capability but by memory bandwidth and speed.

Before processors started using cache as means to speedup processing, faster types of memory did exist and were available but at a much higher cost which made it impossible for them to hit mainstream production. The solution Intel first implemented on the i386 chip was to allow an extra space on the motherboard where a small amount of high speed memory could be connected to provide the processor with high speed memory access for whomever would be willing to pay for it. With compilers getting smarter and adapting to these changes, engineers saw that adding a small amount of cache to a processor could greatly increase the processing speed since all data was not used equally. As an example, while performing a task such as multiplying an n-sized vector by a scalar, the scalar is used n times while the vector is used only once. If the scalar were to be cached, the reading time would be an order of magnitude faster without the need to use high speed memory to store the whole problem's data. One would simply have to have space for a single float or double, normally a measly 4 or 8 bytes, a realization which prompted Intel to create the cache in the first place, given the high marginal gain of using a small portion of costly high speed memory. This architectural change was such a success that on the next family of processors, the i486, cache started to be included on the same wafer as the processor.

Thus, in modern processors the traditional architecture with one memory space was replaced by a hierarchy of different memory levels. AMD's Athlon XP and 64 and Intel's Pentium 3 and 4 added one more layer to the memory hierarchy with the addition of a larger but cheaper L2 cache. With the advent of the multicore architectures Intel's Core 2 Duo created an L2 cache which was common to both cores and therefore permitted sharing of cache memory between cores while the L1 cache was faster but core specific(for counters and such) which further impacted programmability. Current processors such as the Phenom X4 and the Intel core i7 have yet another added level of memory with cores having their own L1 and L2 caches, and sharing an L3 cache (fig).

One unintended effect was that the complexity of the compilers increased dramatically with this new type of memory. Compilers have to decide what to put in the cache, when to put it there - since even changing the order of execution of independent events can impact performance - and where (at what cache level) to write the information. The sharing of information on certain caches also created new possibilities for optimizing code by allowing parallel computing within the cache memory level. While compilers have evolved significantly over the last decades in terms of optimizing code, they are limited in what they know of the task at hand. If one were to have an algorithm with deterministic memory access it would be easy to outperform a compiler by planning ahead and manually decide which variables to store in the different levels of cache and by tweaking memory use to maximize bandwidth by sharing information at high speeds through the common memory cache. In this manner, many linear algebra libraries can achieve performance much higher than would be expected by running a naïve version of the routines and letting the compiler do the work. Nevertheless the growing hierarchical nature of memory increases the possibilities of information storage and, therefore, difficulty for the programmer to define and optimal strategy.



* 1. Vector Processors
     1. The Development of Vector Processors

The SIMD (Single Instruction Multiple Data) paradigm as described by Flynn’s taxonomy [Flynn, 1972] describes an execution model where the same instruction is executed on multiple units of data (words). A vector processor is an implementation of this model where, contrary to a scalar processor which operates on a single word, the same instruction is executed for N words, which can also be viewed as executing on a N-dimensional array of words (which gives insight to why the technology was named a "vector processor"). Immediately, we can identify that the instruction bandwidth will be much smaller since only one instruction read will be required for operating on N words and, therefore, the instruction read overhead will be (1/N)th of that of a scalar processor. Nevertheless, this will only happen if all N processors are actually processing useful information. Since vector processors execute in lock-step, i.e., all individual processors are required to concurrently perform the same operation, if the operation is not data-parallel or barely so, only one or a few of the processors will be used and unused processors simply operate on any data and have the result discarded, resulting in a less than optimal efficiency. It becomes clear that vector processors are, in essence, engineered to solve problems which are predominantly data parallel.

Given that vector architectures are only suited for a specific type of problem, widespread adoption only occurred in the 70’s with the infamous Cray-1 supercomputer. Besides being the first successful supercomputer to employ vector architecture, the Cray-1 was also ahead of time in that it employed individual registers for each vector processor and used a pipeline model[], both of which are ubiquitous in today’s computers. One of the reasons for its success was that, like modern GPU’s, the Cray-1 could deliver a much greater performance compared to other supercomputers of similar price – for problems that were inherently data parallel[]. In this sense the cray supercomputer brought the vector processing model to the mainstream, which at the time was limited to few large and expensive computers mainly used for research.

While modern CPUs are not vector processors *per se*, vendors have added instruction sets over the years that make use of vectorization for added performance for specific operations with examples such as Intel's SSE/MMX and 3DNOW!. These processors are, rather, a myriad of processing models developed over decades with different companies (or even divisions within companies) focusing on different processing models based on the target market's necessities.

* + 1. SSE and CPU Instruction Sets

A characteristic of the x86 architecture is that one of the ways vendors can deliver additional performance is by using wafer space for adding instruction sets. On the other hand, wafer space is limited and there is a tradeoff between adding instructions or cache, for example. Adding instruction sets also increases chip complexity, but when they can be used, large gains in performance can be obtained. While the work can be done on the software side, complex instructions transfer the load to the hardware and usually accomplish the task in a fraction of the time required. Instruction sets exist for a diverse number of applications, from multimedia to 3D rendering (which recently has been dropped given that nowadays it is rare to utilize a CPU to process graphics).

One of the most useful, particularly for scientific computing, is the Streaming SIMD Extensions (SSE) family of instructions which was born to address the shortcomings of the first SIMD instructions proposed by Intel, MMX. It has since then been expanded four times since the inception and is still in the process of being extended in new chips, as with the addition of the new AVX (Advanced Vector Extensions) instruction set. The SSE family, as the name implies, is a SIMD extension which permits the processor to perform as if it were a vector processor, being able to achieve much better performance on parallel operations. This is accomplished by storing data in registers which possess faster memory access and by operating on these registers by using a set of instructions which can each operate on a whole vector of data in one clock cycle. Thus, it is possible to compress into one cycle the computation that would take multiple cycles if done without vector extensions. It is also valid to point out that many times the way an algorithm is programmed makes is difficult to expose the vectorization potential to the compiler, which will result in suboptimal code. To trust the compiler for performance code is tricky and not straightforward, as sometimes even unrolling loops, aligning access and performing separate load/store operations as to expose vectorization to the compiler will not be sufficient to guarantee the code will utilize the appropriate vectorization instructions.

Although considerable gains are possible using extended instruction sets, one counterpoint is that instruction utilization depends not only on the hardware where it is run but is also compiler specific. That means roughly doubling the lines of code if you want software that can be built on a windows compiler and GCC, for example. Using a framework which compiles to multiple platforms such as OpenCL(Open Computing Language) makes the code capable of vectorization and compiler independent. For small functions which get called many times, such as in linear algebra libraries for CPUs, the overhead which OpenCL entails makes the code better suited for utilizing instructions directly. Although the resulting code may not be the most elegant, with conditional compilation to account for the platform and the necessity to port from one compiler to the other, for applications where performance is paramount, it is the best option.

* 1. GPU Architecture and the Development of General Purpose GPU Computing

With the advent of the programmable vertex and pixel shaders, developers were given the flexibility of creating their own code for customizing the rendering process to meet their needs. While most developers used this new technology for its original purpose, i.e. creating custom graphics effects or adding realism, it became clear that while the shader was made to operate on a vertex or pixel, any data could be treated as such and thus the GPU (Graphics Processing Unit) could be programmed to be used for any general purpose computation. This formed the field of GPGPU (general purpose GPU) computing. With the rapid growth of the processing power of GPUs which quickly surpassed that of CPUs (central processing units) pushed by the highly competitive gaming industry, research began to appear that took advantage of the great and untapped pool of GPU resources.

To process hundreds of vector and matrix required by shader algorithms, state of the art GPUs possess hundreds of “small vector processors”, also called SIMD engines (In AMD nomenclature) each of which contains a few ALUs (Arithmetic Logic Units) (fig). ALUs can perform single precision or integer operations. To perform double precision operations, some architectures group ALUs together which yielding double precision performance which in inversely proportional to the number of ALUs grouped. This means that even though GPUs, in general, operate with a smaller clock rate with regard to CPUs, current hardware is able to perform *thousands* of arithmetic operations per cycle, and frequently hundreds of special functions (sine, cosine, square root, etc.). Since special function operations, also called transcendental operations, occur less frequently, there is usually only one special function processing element per engine. Another design groups ALUs, potentially sacrificing ALU power to perform a transcendental operation. The loss in processing power is offset by gained wafer which can be used for memory, control or processing. For this reason, algorithms that rely heavily on such operations such as Fourier transforms and random number generators are one of the prime beneficiaries of GPU computing.

It is valid to take a detour here and observe that comparisons regarding "processing power" are used rather loosely and can be taken to mean peak throughput, and this liberty was taken for the sake of comparison. In reality, achieving peak throughput or anything close to it is highly algorithm dependent which makes it difficult to speak of processing power in general terms. Such figures such as gigaflops and teraflops should thus be read as a comparison between the potential of rivaling technologies with the addendum that GPU potential is naturally harder to extract, given the many layers of parallelism present .

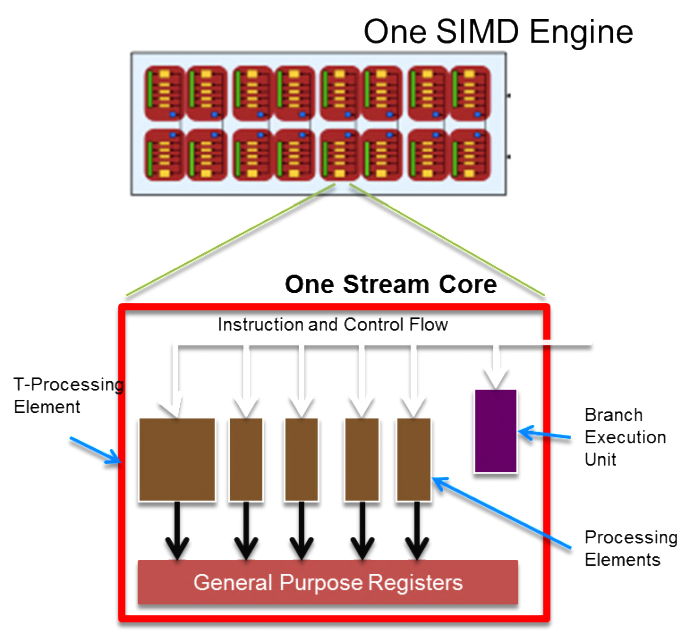


Fig.

With the advent of the first programmable shaders, although the potential gains were high, so was the initial investment in learning to program for general purpose computation on a GPU. Initially only programmable in assembly, GLSL (OpenGL Shading Language) greatly increased the ease of programming shaders by raising the level of abstraction and offering the developer the ability to program his own kernels in C variant. GLSL was also a standard to which each vendor had his own implementation. This meant that programmers did not need to learn vendor specific assembly code to write their own custom shaders. While this was more practical than dealing with instruction language, GLSL was still a language written for shaders and not for performing general purpose computations on a GPU. This introduced some awkwardness in the process of programming because of the necessity of treating any piece of data which was operated upon as a pixel or vertex, making code unreadable to a layman of the world of shader programming. General purpose computation with GLSL left an overarching feeling of being more a "hack" than something usable for serious development in the field of general purpose computing.

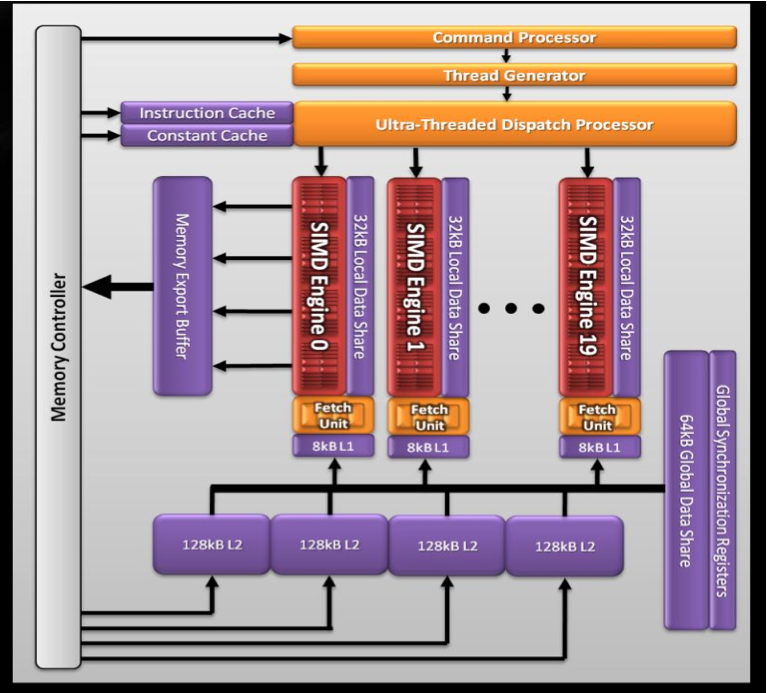
Nevertheless, the ease of being able to program a shader in C, which is heavily used by the scientific community and known to be performant in HPC made this new technology gain much wider appeal. Furthermore, for many programmers familiar with the OpenGL API and the intricacies of managing buffers and the interface in general, the transition was facilitated, since the primary focus of the language was to extend OpenGL's capabilities and this was, after all, their targeted audience. A boom in research ensued, which was primarily targeted at computer graphics but extended to other areas as well with the realization that many of the benefits reaped by this specific audience could be used by a wider family of algorithms.

While the hardware continued to evolve, hardware vendors realized that vertex and pixel shaders had many overlapping functions. By joining the two shaders into a unified shader architecture, manufacturers were able to save wafer area and pack more transistors into the chip, while also being able to balance the computational load when one form or the other was being more heavily required - further increasing processing power. Also, this required that a new level of abstraction be introduced, since shader was to be used for vertices and pixels, leading to a further increase in the programmability of the GPU.

Nvidia and ATI, the only two large GPU manufacturers who managed to survive the intense competition, recognized the potential of this new model and developed their own high level languages to facilitate the task of creating a program to run on a shader, commonly known as a kernel. Of the two languages created by Nvidia and ATI (AMD), respectively CUDA and Brook+, CUDA enjoyed the most hype and was embraced as the standard for GPU computing because of a gentler learning curve and higher level of abstraction which made the it more accessible, as well as Nvidia's capacity to quickly develop a functional and mature programming language for the GPU. The fallback was that CUDA and Brook+ were still vendor specific languages and therefore allowed no flexibility for porting code to other platforms. Meanwhile, IBM also found it’s way into the general purpose computing scene by creating the acclaimed CELL BE processor for the Playstation 3. Promptly recognized as a viable alternative to GPUs for massively parallel computationally intensive tasks, the Playstation 3 spawned the creation of a number of beowulf type clusters because the low cost per FLOP (Floating Point Operation), and the added benefit of networking and storage in a cheap setup. The cell architecture itself is an interesting compromise between traditional CPUs and the first programmable GPUs. The added flexibility of the CELL BEs multiple cores with DMA controllers and with a threading model in mind made it a powerful processor for general purpose computing. With it IBM added yet another language to the growing list of languages tailored for the massively multicore era.

OpenCL was therefore created in an industry effort to standardize the programming for these different platforms. While each vendor would have his own implementation, since there was a standard they would have to adhere to, the portability to different platforms would be guaranteed. This was made possible because while the underlying hardware was completely different when dealing with a DSP (digital signal processor), CPU or GPU, all of these have hardware that can benefit from exposing vectorization, memory hierarchies and parallelism with a greater detail and control. In essence, what was needed was an abstraction to deal with programming for multiple cores, possibly hundreds, and to be able to manually manage the different memory hierarchies, mapping the hardware abstraction to a programming model (fig).

OpenCL was conceived with this purpose, and has been constantly growing in adoption, posing itself as a potential new industry standard for heterogeneous programming.



The standard is supported by most of the large processor vendors, including the largest growing market segment of the industry, that of mobile processors for smartphones and tablets which utilize architectures which can greatly benefit from such a programming paradigm. It is also valid to note that by creating a high level language that gives the developer explicit management of the different levels of the memory hierarchy also greatly benefits the CPUs, which have taken a road similar to that of GPUs to be able to keep up with Moore's law: add cores and layers of memory (cache) to increase performance. The fact that OpenCL and other frameworks address this problem is what makes it, in part, possible to extract more performance out of such a wide range of platforms.

1. OpenCL
   1. Introduction

OpenCL is a standard defined by the Khronos Group, which is formed by a consortium of major players in the technological industry including hardware manufacturers, software developers and middleware vendors with the purpose of creating an open and portable API for accessing the capabilities of multicore systems. OpenCL provides a level of abstraction which enables the developer to write code for heterogeneous platforms consisting of CPUs, GPUs, APUs (Accelerated Processing Units), DSPs (Digital Signal Processors) and Accelerators (e.g. IBM Cell BE processor) within one unified programming model. Through the OpenCL programming language it is possible to write code based on the ISO C99 standard [ISO/IEC 9899:1999] extended to write functions, called kernels, which execute on the device. Through the abstraction of the hierarchical memory present in modern GPUs and CPU with multiple levels of cache, OpenCL provides the capability of creating code which will take advantage of greater bandwidth provided by some levels of memory to create optimized code without the need resort to vendor specific instruction language. Although programming in instruction language provides a greater level of control and performance, using OpenCL in to program in C99 like language makes the code comparatively more readable and maintainable, besides being a cross-platform solution. A large number of resources for programming with OpenCL are available online. A good reference on utilizing the framework, describing the programming language and the functionality available through the API is [Benedict R. Gaster et Al., Heterogeneous Computing with OpenCL], and I would also point to [Aaftab Munshi et Al., OpenCL Programming Guide] for a broader understanding of programming with OpenCL. This chapter will offer only a small overview of the framework necessary for the comprehension of posteriorly cited optimization strategies.

* 1. Execution Model

OpenCL works on the abstraction of a GPU composed of multiple compute units. The compute units are, in turn, composed of multiple processing elements which work as vector processors as shown in fig. The processing elements contain multiple ALUs and, in some cases, a SFU (special function unit) which some vendors call “cores”. Since the underlying hardware differs from that of a CPU, certain concepts such as that of a CPU “core” do not map directly between both architectures and the term “core” must be taken with a grain of salt. For that reason, the ALUs are called *processing elements* within the context of OpenCL (fig x).

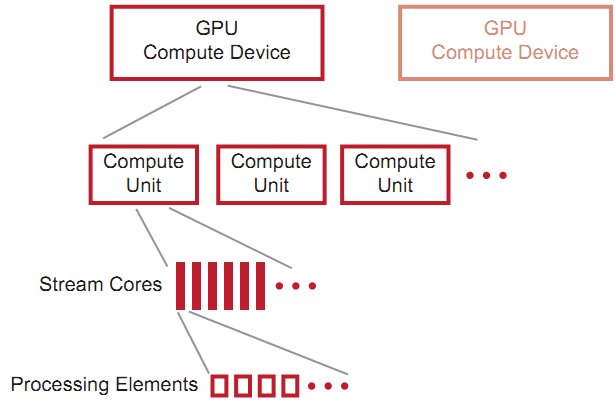


Fig x (AMD)

A kernel, as previously described, is basically and C function with some syntax limitations, particularly with regard to memory management. Pointers do not work in the same way as C and there are vector data types built in, but aside from a few differences a C programmer can easily transition to writing OpenCL kernels. OpenCL also has many built in functions which, given native hardware support, can be used to perform more complex instructions in a single cycle such as MAD (fused multiply and add), 2D and 3D vector operations such as scaling, addition, dot product, sine, cosine and random number generation, and various others. Many of these functions would require dealing with assembly code to perform or depending on the compiler.

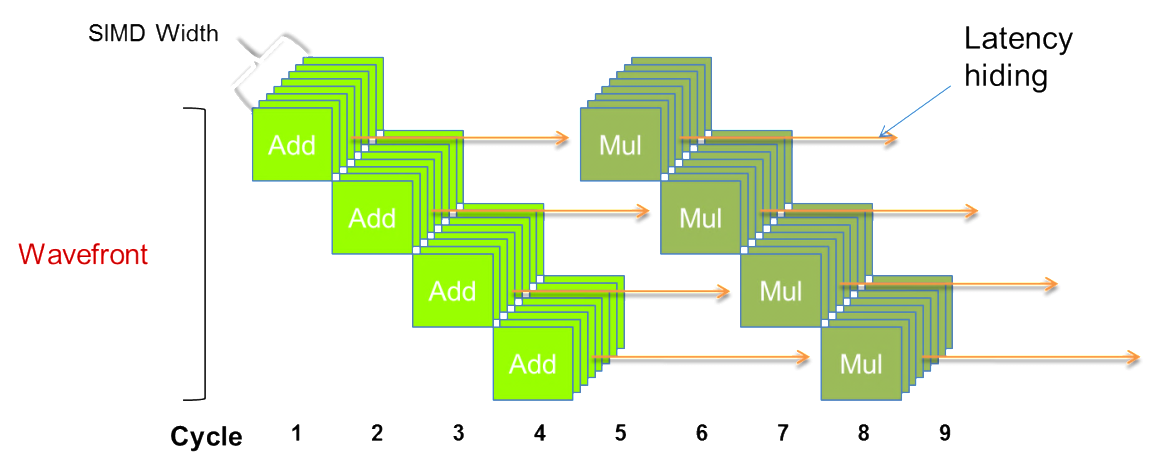
Memory management within the kernel is in large part deferred to the OpenCL library running on the host and cannot be dynamically managed from with a running kernel, which is one of the shortcomings of programming for the GPU. Memory is managed through the creation of buffers which indicate the amount of memory which will be allocated and queues which can be used to schedule the asynchronous or synchronous transfer of data to the GPU. Optionally, the GPU can operate directly on the computers data stored in RAM, which is not our case.

Each compute unit runs an instance of this kernel, called a *work item* which is somewhat analogous to a CPU thread. Within a compute unit, all work items execute the same instruction. In a graphics card, hundreds of work items can be generated and because of the common use with computer graphics and models, these work items are arranged in and N-dimensional grid, called *ND-Range.* Utilizing a 2D range for images, where each work item corresponds to a pixel, and 3D range for a 3D model, where each work item corresponds to a voxel (3-dimensional pixel) to deal with images or 3-dimensional objects makes it easier for the programmer to create algorithms around these objects, around which graphics applications are typically centered. Using this indexing has its own benefits such as increasing cache hits by exploring data locality for certain algorithms, but in the context of finite elements utilizing a 1D range, i.e. a vector, suffices.

Work items are divided into groups called *workgroups.* A workgroup has access to common shared memory and synchronization directives which it can use to perform operations such as parallel reductions. The task of determining the size of the workgroups is deferred to the programmer, and the size chosen determines how local much local memory will be reserved for each group. The determination of work group size, therefore, has a large impact on the overall performance of the algorithm. Determining the best workgroup size is still something of an art and oftentimes counterintuitive, since there are always a number of conflicting interests e.g. larger numbers of work groups are generally good for filling wavefronts which on the other hand implies on less local memory per workgroup to perform parallel reductions.

Work items are executed in fixed groups of work items, which depend on the hardware, called wavefronts. While wavefronts are determined by hardware, grouping is determined by software. For the hardware used in the benchmarks, the wavefront size is 64. This is because there are 320 “stream cores” (fig) divided into 20 compute units yielding 16 cores per compute unit. Without going into much detail, the VLIW (very long instruction word) architecture permits the compute unit to issues one big instruction for 4 cycles at once. To have the hardware completely utilized it would be necessary to have a group size of, at least, 64(one work-item per core times cycles necessary) and be a multiple thereof, since any factionary number would leave a wavefront only partially full [ocl prog guide]. Besides having the compiler extract instruction level parallelism, another reason to which also benefits more recent GPU architectures is to diminish the latency in memory access.

Due to characteristics of the GPU model, memory latency is preponderant factor in GPUs. To minimize the latency conundrum, GPUs pipeline work item execution so that while one work item fetches information from memory, other work items are executing. Up to four work items of the same wavefront can be pipelined on the same stream core. Performance is therefore dependent on keeping the pipeline full so that latency does not affect performance, a technique called latency hiding. An example of the pipelined execution can be seen in fig.



Fig

As a concrete example, one of the GPUs used for the test, the ATI Radeon™ HD 5870 HD has 20 compute units which each contain 16 stream cores. These cores each contain 5 processing elements adding up to a whopping 1600 processing elements. The wavefront size is 64 work items. While there is an immense added complexity, the theoretical peak throughput of this already outdated graphics card is 2.72 teraflops in single precision and 544 gigaflops in double precision (processing elements are linked together to perform double precision arithmetic leading to lower throughput). Nevertheless, as has been noted many times, achieving anything close to maximum throughput requires a special set of problems that can be parallelized into hundreds of different portions, preferably without the need of communication between these portions, and without branching. It is clear that the GPU evolved with a very specific problem in mind.

On more recent models, the choice has been made to sacrifice peak throughput in favor of a model more suitable for general purpose computing. The more recent models offer greater flexibility for wavefront execution and have shifted the job of scheduling wavefronts from the compiler to the hardware simplifying the compiler’s job at the cost of occupying wafer space. On the other hand this allows more flexibility for wavefront executing regarding branching and exploring thread level parallelism. On most SIMD execution models, branching takes a heavy toll since lockstep execution does not allow diversion of execution paths and therefore should be avoided. This deficiency is partially mitigated by the more recent developments at the cost of some computational potential.

* 1. Memory Model

OpenCL contains four main memory domains: private, local, global and constant memory (fig.). Private memory pertains only to a work item and is not visible by other work items. It is the smallest level of granularity at which memory is divided on the GPU. Local memory is a level above, and can be shared between work items contained in a common workgroup and provides synchronization directives. Global memory is visible by every work item and is at the highest level of granularity, being one monolithic chunk of memory every work item has access to but without possibility of synchronization between work items. As in the different cache hierarchies in a CPU, there is progressively more memory on chip moving up towards global memory but memory access speed also decreases by roughly an order of magnitude for each level.

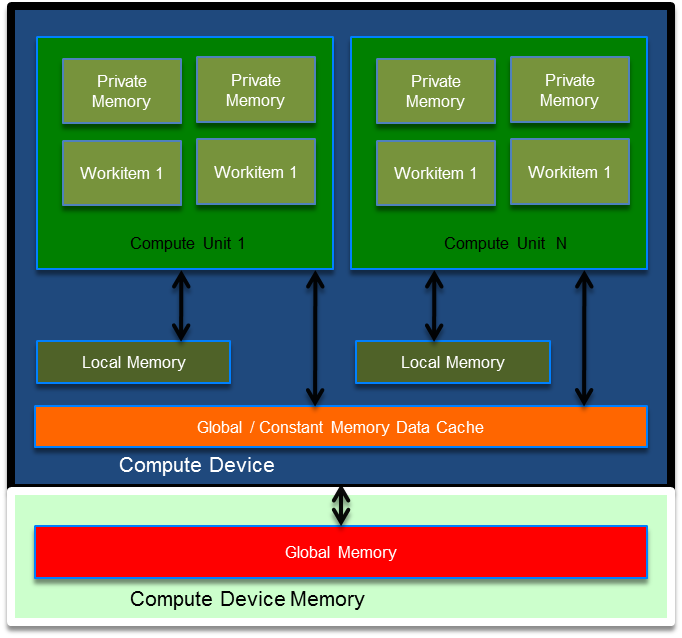


Fig.

The many layers the programming model contains, be it in execution or in memory management, are what make programming in OpenCL challenging. Wrapping one’s head around such concepts takes time to get used to, even for experienced programmers, particularly because the programming model adds many dimensions to be managed, far beyond what programmers are generally used to.

1. Sparse Matrix Data Structures

The formats used to store sparse matrices are central to the performance of algorithms which rely heavily on matrix-vector or matrix-matrix operations. Contrary to a dense matrix which stores all the information in a matrix directly, a sparse matrix storage format usually stores only nonzeros or at least tries to store the smallest number of nonzeros possible. This makes it possible to operate on very large matrices which under a dense storage format would consume orders of magnitude more memory. The format determines how the data of the matrix will be accessed and stored, impacting on time to read or write a matrix entry and the size of the stored sparse matrix. Most modern formats are relatively cheap to store and with very little variability between format storage sizes to make a significant difference.

As observed in some operations in linear algebra sometimes there is some reuse of rows or scalars which makes dividing a matrix into rows or columns a natural choice. Depending on the way a matrix is divided, indexed and stored, these decisions can make a format better suited to one particular type of matrix or to solve one particular problem. Matrix operations generally take up a considerable amount of the computational and are therefore sections which provide considerable impact upon performance. Thorough discussions on the difference between data structures and which structures are more adequate for given problems and hardware architectures, particularly vector architectures, have thus been widely discussed (P. T. Stathis, also Amer Wafai) and a large body of comprehensive studies are available. Only a succinct description of each format employed follows will be provided with brief implementation remarks (Code for all sparse matrix data structures can be found on appendix A)

* 1. Diagonal Format

The diagonal sparse matrix format, also called compressed diagonal, stores sparse matrices as diagonals, which are stored if they contain a nonzero. The main advantage of the diagonal sparse matrix format is that given a regular band, this format can have a very small memory overhead, especially for stiffness matrices which have gone through a decent nodal reordering algorithm. If a matrix were to have a full row, this format would be worse than storing a dense matrix. It enjoyed popularity within the finite element community for structures because of the regular band densely packed about the diagonal, making the engineers inclined to choose it. Unfortunately the small gain in memory consumption is offset by poor performance for the required operations, as the format does not offer an efficient way to access data for operations such as matrix vector multiplications (one must access all diagonals to perform a matrix vector operation) and for random insertion which is important on the assembly stage the format performs as well as the other formats. The format requires storing the number of diagonals n, and a vector with all the stored diagonals, and a n-sized vector to indicate the offset from center diagonal of each diagonal. Positions are defined by convention. The general C data structure would include these fields:

int numdiagonals;

fem\_float\* diagdata;

int\* diagoffset;

* 1. Compressed Sparse Row

This particular implementation of the CSR format may have a few differing points from other implementation with regard to memory allocation, namely that each row has its own pointer which adds a memory allocation overhead but at the same time permits independent memory reallocations. In the context of finite elements this decision allows performance gains at the assembly stage when used in conjunction with mesh coloring, since independent memory management of rows to be resized in parallel, avoiding the issues of race conditions other formats have to consider. The strategy used was to create a data structure to hold each compressed row's data as follows:

typedef struct ROWdata {

fem\_float\* values;

int\* col;

int numNZ;

int maxSz; // Used to control reallocations

} ROWdata;

The matrix is then represented by a vector of these structures which represent a compressed row. The advantage of utilizing this format for finite elements is that it allows rows to be reallocated in parallel without needing to move a chunk of memory, which provides more flexible filling, compared other data structures which group all non-zeroes into one vector at the cost of slower deallocation. Also, since dynamically created pointers to pointers are not supported by the GPU, the data is not directly transferrable to the GPU.

* 1. Ellpack Format

While the Ellpack format lost some ground with the demise of most of the vector supercomputers, with the return of the vector processing paradigm into scientific computing with GPU programming, other formats such as Ellpack-R (F. Vásquez et Al.) based on the Ellpack format have been devised to take advantage of vector processing and address prerequisites such as data alignment. The traditional Ellpack format is also especially well-suited for finite element problems in structures, particularly because as was previously mentioned in general the problems have a somewhat regular banding. This makes storing padding for rows less common and the storage scheme more efficient. For problems with more variable row width, formats such as described in the former reference would be a better option, adding the storage of number of nonzeros in each row as additional information. To store a matrix in the Ellpack format the matrix has all the nonzeros shifted left and the matrix is padded to length of the row with maximum number of nonzeros n. The entries are then scanned column-wise to a matrix. A matrix of the same size is stored keeping an integer with the column value for the corresponding entry. The row can be inferred since they are scanned sequentially when building the matrix, restarting at the first row after the first column is entirely scanned. The data necessary to store the information is therefore minimally:

fem\_float\* data;

int\* colindex;

int maxrownnz;

* 1. Eigen Format

While not a separate format in itself, the Eigen sparse matrix library was added to the sparse matrix interface to have a library that performs well to compare to. Documentation suggests the use of a variant of the CSC (Compressed Sparse Column) format. While our formats were tailored for our problem and our needs, Eigen which is a general format performed well, particularly the dynamic format which has a memory overhead which is a bit larger than the regular format but has better performance regarding random insertion which is where most of the time is spent assembling a stiffness matrix. Another advantage of using the Eigen library is that since it is a header only library, it can be compiled with the code utilizing the same compilation optimization level as the other formats.

* 1. Searching index arrays

Sorted arrays are commonly used to store indices and thus getting or setting information in the sparse matrix passes necessarily through these operations. When assembling the stiffness matrix, entries have to be read and updated multiple times per element and the speed at which memory can be retrieved and stored impacts the performance greatly at this stage. Efficiently searching for the correct index is therefore a very important part in having a sparse structure that performs well. While understanding of Big O notation is necessary to understand the performance of an algorithm, relying solely on the limiting behavior of the algorithm as an optimization choice can yield counter-intuitive results if the nature of the problem is not taken into consideration. For problems with small N, for example, although in the long run a certain algorithm may yield better performance, setup costs may make the initial overhead trump the gains of a theoretically faster algorithm. As Donald Knuth points out “People often abuse O-notation by assuming that it gives as exact order of growth; they use it as if it specifies a lower bound as well as an upper bound” [Knuth, D; The Art Of Computer Programming vol1]. It is valid to point out that one could theoretically take Big Omega to be able to find a lower bound and from a theoretical perspective select the appropriate algorithm. Nevertheless, other factors such as caching, memory and bandwidth boundaries as well as the distinct performance of multiple memory hierarchies which are paramount for optimization are not captured by these models.

In the context of searching within a row of a matrix generated by the finite element method, row size is a function of mesh structure. The belief that selecting the “best” algorithm according to Big O notation in hope of, at least, building a system that will scale well is in this case erroneous, since the algorithm will scale according to mesh structure and not problem size. Considering that elements connected to many other elements (e.g. “star” structure) are the exception rather than the rule, it would be wiser to program for the most common connectivity of each element. For each element type analysis was performed in order to find the impact of searching algorithms on sparse matrix performance. This analysis does not include OpenCL searching within the GPU, which has its own particularities and is discussed in the appropriate section. Differences between the different searching techniques have been discussed at length here: [<http://schani.wordpress.com/2010/04/30/linear-vs-binary-search/>].

* + 1. Linear Search

The traditional linear search algorithm was used as a basis to compare with other approaches used for searching an ordered array. The pseudocode for finding the correct place to insert a value in an array which returns and index is:

**FOR** position = 1 to length of vector

**IF** vector[position] >= value **THEN**

**RETURN** position

**END IF**

**END FOR**

We can then check if the value exists to discover if the value has to be inserted or modified or simply return that it does not exist in the case of a query. For small vectors, or in the case of sparse matrices for narrow bands, linearly searching across the vectors usually is faster than performing a binary search.

* + 1. Binary Search

The implementation for binary search was also used as a basis for posterior comparison with other versions. The pseudocode for the implementation is as follows:

**STORE** variable min set to zero, variable max set to length

**WHILE** (min < max)

**STORE** middle is average of min and max

**IF** (searched value > vector[middle]) **THEN**

**STORE** min is middle + 1

**ELSE**

**STORE** max is middle

**END IF**

**END WHILE**

**RETURN** min

* + 1. Linear Search with SSE

Using the SSE instruction set it is possible to speed up our linear search algorithm to search more positions per clock cycle. This can be done because utilizing special high speed CPU registers we are able to process multiple operations per clock cycle and, therefore, reduce the number of clock cycles necessary to perform the required operation. One of the downfalls is that aligned memory access is required to transfer the data efficiently to the registers, which impacts the requirements of the underlying data structure. Besides this, since up to 16 comparisons can be fitted into the registers, larger array sizes benefit the most from utilizing SSE linear searches. Second to all this, while the primary objective of writing optimal code is performance, utilizing SSE instructions makes the code harder to read, debug and maintain, especially considering that the code is compiler dependent and thus compiling for GCC would require extending the code with compiler directives. The SSE code utilized and compiled with MSVC was the following (description of the code was added as comments):

// Value to be compared is copied into 4 vector positions

\_\_declspec(**align**(16)) int v4[4] = {val, val, val, val};

const \_\_m128i\* v4ptr = (\_\_m128i\*)v4;

// Loads comparison values into registry

\_\_m128i key4 = \_mm\_load\_si128(v4ptr);

int i = 0;

\_\_declspec(**align**(16)) unsigned short res;

// Does search 16 elements at a time

for (i = 0; i <= len; i += 16) {

//

const \_\_m128i\* in0ptr = (\_\_m128i\*)&intvector[i ];

const \_\_m128i\* in1ptr = (\_\_m128i\*)&intvector[i + 4];

const \_\_m128i\* in2ptr = (\_\_m128i\*)&intvector[i + 8];

const \_\_m128i\* in3ptr = (\_\_m128i\*)&intvector[i + 12];

\_\_m128i in0 = \_mm\_load\_si128(in0ptr);

\_\_m128i in1 = \_mm\_load\_si128(in1ptr);

\_\_m128i in2 = \_mm\_load\_si128(in2ptr);

\_\_m128i in3 = \_mm\_load\_si128(in3ptr);

\_\_m128i cmp0 = \_mm\_cmpgt\_epi32(key4, in0);

\_\_m128i cmp1 = \_mm\_cmpgt\_epi32(key4, in1);

\_\_m128i cmp2 = \_mm\_cmpgt\_epi32(key4, in2);

\_\_m128i cmp3 = \_mm\_cmpgt\_epi32(key4, in3);

\_\_m128i pack01 = \_mm\_packs\_epi32(cmp0, cmp1);

\_\_m128i pack23 = \_mm\_packs\_epi32(cmp2, cmp3);

\_\_m128i pack0123 = \_mm\_packs\_epi16(pack01, pack23);

// Result is saved as a mask (fits more comparisons)

res = \_mm\_movemask\_epi8(pack0123);

if (res != 0xffff)

break;

}

int count = 0;

// To find position scans the bits of the resulting mask

if (res) {

unsigned long rb = 0;

\_BitScanForward(&rb, (unsigned long)~res);

count = rb;

}

return i + count;

1. Implementation of The Finite Element Method
   1. Determining the Global Stiffness Matrix

Determining the stiffness matrix is one of the main steps in performing analysis by the finite element method. Our version utilizes isoparametric finite elements, the most effective method for most practical analyses [bathe]. The global stiffness matrix is utilized to solve the matrix equation:

with which we can then insert boundary conditions to solve for the displacements. To first determine the global stiffness matrix, we first determine the local stiffness matrices of each element in the model and then assemble to a global stiffness matrix which represents all of the model's degrees of freedom. The element stiffness matrix can be evaluated as:

The work of calculating the integral is facilitated by using normalized coordinates, i.e. isoparametric elements, because when we perform a Gaussian numerical integration, we are always integrating in the same domain. To transform from natural coordinates to the general coordinate system, me must calculate:

Our corresponding pseudocode for the algorithm is, therefore:

**FOR** element = 1 to number of elements

**STORE** element Coordinates from node coordinates and element connectivity matrix

**FOR** gauss point = 1 to number of gauss points squared if in 2D, cubed if in 3D

**STORE** r,s,t natural coordinate values given the current gauss point

**COMPUTE** derivative of shape functions for current element type and r,s,t position

**COMPUTE** jacobian matrix given the element coordinates and derivative of shape functions

**COMPUTE** determinant of jacobian matrix

**COMPUTE** the inverse of the jacobian matrix given itself and its determinant

**COMPUTE** the derivative of shape function matrix in cartesian coordinates by multiplying the inverse of jacobian matrix by the derivative of shape functions in natural coordinates

**COMPUTE** B matrix by utilizing data in computed derived shape function matrix in natural coordinates

**COMPUTE** B transposed by transposing elements in B

**COMPUTE** CxB matrix by multiplying constitutive matrix C (given by constitutive model adopted) by strain-displacement transformation matrix B

**COMPUTE** temporary stiffness matrix by multiplying prior result, CxB, by B transposed

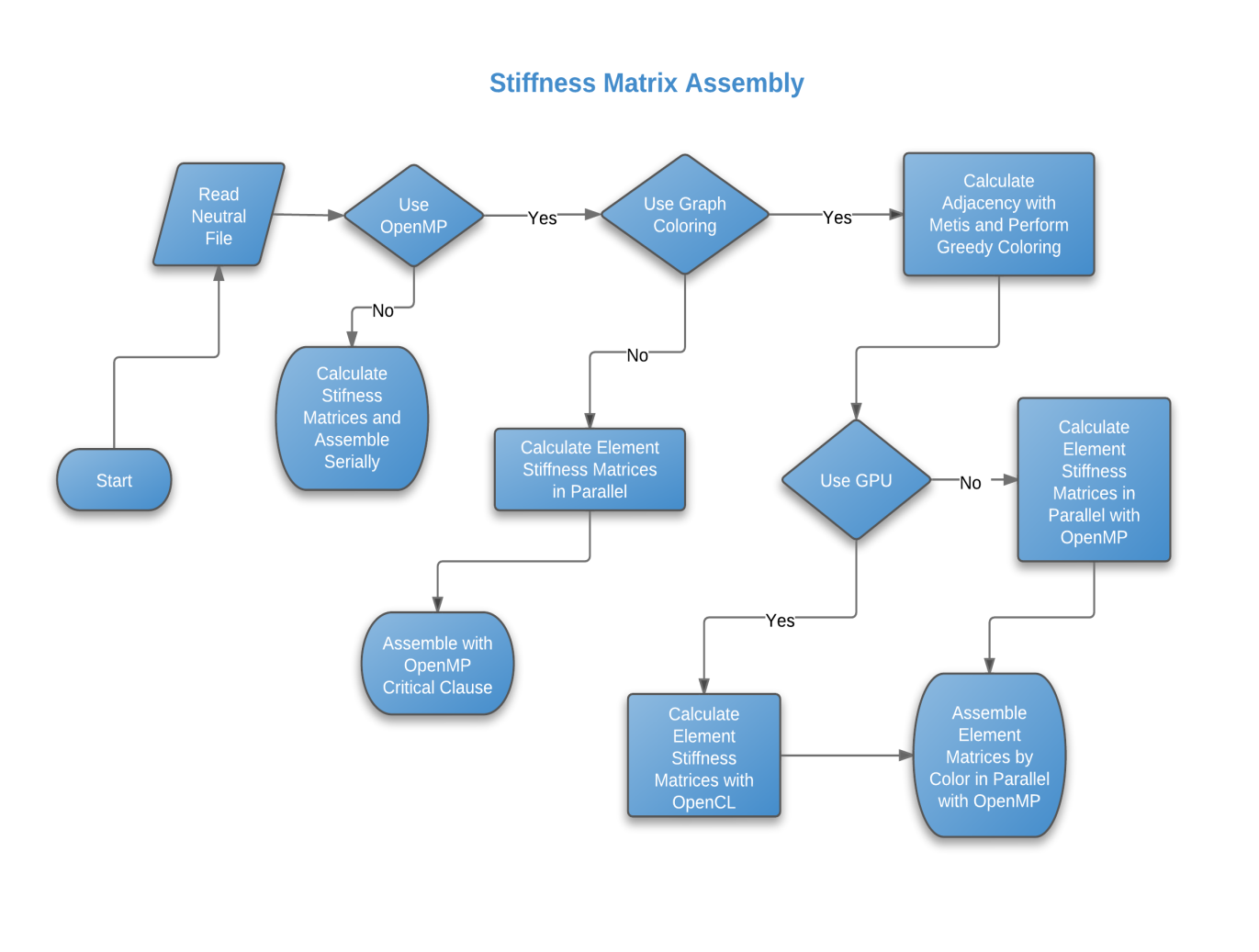
**STORE** in local stiffness matrix values calculating by adding the partical results of the numerical integration

**END FOR**

**STORE** in global stiffness matrix values of local stiffness matrix by assembling matrix into corresponding global degrees of freedom

**END FOR**

Since the in assembly process degrees the corresponding degrees of freedom are modified more than once, to parallelize this portion it is necessary to ascertain that concurrent threads are not simultaneously accessing the same element, which could lead to a race condition and yield erroneous results, e.g. two threads read the same value, increment the value by the amount that will be assembled, and then write to the same location irrespective of the work the other thread has done. To sidestep this problem it is possible to tell the compiler through OpenMP directives that a certain portion of the code should not be parallelized as covered in the OpenMP implementation section. Another possibility that maintains parallelism would be to calculate where these race conditions would arise, i.e. where elements have corresponding degrees of freedom, and “color” these elements with different colors. Therefore, while elements of different colors must be assembled serially, elements of matching color can be assembled without need for worrying about race conditions. Yet another option which is not covered in this text but which has been extensively discussed and which also yields good results would be to parallelize in the domain of the nonzeros(). The different paths possible which were implemented are all shown in the following workflow graph():



* 1. Parallel implementation using OpenMP

By utilizing OpenMP it is possible to obtain a considerable speedup in multicore systems without having to refactor a great quantity of code when using C or C++. It enables us to delegate and distribute work to different processors, while having control over locks, atomic decisions, load distributions and other features of parallel processing while maintaining a simple API.

One straightforward way to parallelize our code as referred to in [] would be to compute the local stiffness matrices in parallel and then assemble to the global stiffness matrix. Care must be taken that race conditions do not affect the result, for while calculating the local stiffness matrices are independent calculations, when assembling the results race conditions may appear of two processors modify the same global degree of freedom at the same time. It is therefore necessary to utilize the CRITICAL construct from OpenMP to assert that only one thread performs this operation at a time. The resulting pseudocode would therefore be:

#pragma omp parallel for

**FOR** element = 1 to number of elements

**STORE** element Coordinates from node coordinates and element connectivity matrix

**FOR** gauss point = 1 to number of gauss points squared if in 2D, cubed if in 3D

Calculates local stiffness matrix

**END FOR**

#pragma omp critical

**STORE** in global stiffness matrix values of local stiffness matrix by assembling matrix into corresponding global degrees of freedom

**END FOR**

* 1. Parallel CPU implementation using graph coloring

By utilizing metis, a third party library to obtain a k = 1 graph coloring of the elements of the mesh by giving metis the elements connectivity. Metis then returns a colored graph which can be used to perform a quick greedy coloring and for a vector with all the elements for a given color. It is proven that there exists a coloring where the whole mesh can be colored in parallel in no more than n + 1 colors(), where n is the largest connectivity of all elements. This means that the mesh can be colored in no more than n + 1 steps, where each step may perform its calculations in parallel. Meshes with elements connected to many other elements are also a problem for other balancing methods and are usually discouraged during modeling. Nevertheless the presence of a few of these will not sufficiently impact the performance gain since for each of the added steps of an added connectivity there are usually hundreds of elements to be executed in each step which dilutes the added time of having to force a few sequential steps. All elements of a given color are executed in parallel for a given step, since elements that were given the same color have no degrees of freedom in common. This way it is possible to perform parallel loops over the elements without having to worry about race conditions in accessing the same element, as would happen with a dense matrix.

With sparse matrices certain particularities arise with regard to memory management since adding an entry sometimes requires reallocating an array which holds all nonzero values and therefore requires that the operation be performed by a single thread. This is a bottleneck of many sparse implementations. The compressed sparse row implementation, as described in the sparse matrix formats section, permits parallel access to rows and with graph coloring we are guaranteed that no two shared degrees of freedom are accessed concurrently. This way we can modify values and reallocate sparse compressed row matrices without having to worry about race conditions. The pseudocode for the parallelized version using graph coloring is as follows:

**FOR** color = 1 to number of colors

**#pragma omp parallel for**

**FOR** auxelem = 1 to number of elements with this color

**STORE** find current element based on elements for current color and auxelem index

**STORE** element Coordinates from node coordinates and element connectivity matrix

**FOR** gauss point = 1 to number of gauss points squared if in 2D, cubed if in 3D

Calculates local stiffness matrix

**END FOR**

**STORE** in global stiffness matrix values of local stiffness matrix by assembling matrix into corresponding global degrees of freedom

**END FOR**

**END FOR**

* 1. Parallel implementation in OpenCL

The parallelization in OpenCL was made calculating each element’s local stiffness matrix in parallel and assembling on the host device with the CPU utilizing OpenMP and graph coloring. The only difference between the assembly of the local matrices calculated by the GPU and the that of section 3.3 relative to the CPU is that while in section 3.3 the assembly was performed after each local matrix calculation the GPU code passed back a vector with all of the sparse matrices calculated and subsequently performed the assembly in the same manner. This implementation can be scaled to larger problem sizes/distributed systems easily by assigning the calculation of local stiffness matrices to different GPUs and performing the assembly of sub-blocks of local matrices by the host. Furthermore, for a given color multiple hosts can perform assembly in parallel and while GPUs are busy, given that proper care is taken to avoid race conditions regarding memory allocation (providing critical sections or preallocating matrices by counting nonzeros which can be done by third party libraries such as METIS). Different implementations exist such as utilizing a degree of freedom, i.e. a row/column of the sparse matrix as the domain of parallelization, or by nonzero which were very well documented and discussed (Cecka, C. et Al.).

* 1. Setting Boundary Conditions

Given the focus on performance and optimization, boundary conditions were enforced using the penalty method and heuristics such as the “Square Root Rule” [Felippa, Carlos; Introduction to The Finite Element Method]. While it is clearly not the best option in terms of numerical accuracy, this step does significantly affect the overall running time, and thus is not a primary concern regarding performance.

1. Solving for Displacements

The global stiffness matrix resulting from assembling the system of the previous chapter is a positive definite matrix. Between the solvers used for positive definite matrices, the conjugate gradient method is one of the most widely employed. Iterative solvers such as the conjugate gradient method are especially suited for solving large linear systems of sparse matrices, where traditional factorization methods would destroy matrix sparsity. The foremost reference regarding the conjugate gradient method used for this section is Shewchuk's seminal, aptly named paper "An Introduction to the Conjugate Gradient Method Without Agonizing Pain". For a more detailed explanation I would urge the reader to refer to this reference, and most material exposed in this section should pertain to particularities of implementation and also regarding the finite element method.

Many implementations of the CG (conjugate gradient) method are available, and the implementation of the algorithm is relatively straightforward, especially considering the vast available literature on the subject. Being that in a traditional finite element CPU computation, the largest part of the computational time is spent on the solver makes it a great target for exemplifying some OpenCL optimization techniques. These techniques can be used in a variety of different situations when programming for vector processors, increasing the efficiency of the GPU use. Within the CG method, the operation which, by far, consumes most of the attention of the processor is the matrix-vector multiplication. This operation is the basis of almost all iterative solvers, particularly Krylov subspace methods. For this reason there is a considerable amount of academic research devoted to this simple linear algebra operation, making it an important benchmark for determining performance of different sparse matrix formats. Given an Ellpack format as described in section [], A naïve implementation of the matrix vector multiplication operation using OpenCL is:

uint gid = get\_global\_id(0);

if (gid < matDim) {

uint nnz = rowNnz[gid];

float sum = 0;

for (int i = 0; i < nnz; i++) {

int index = i \* matDim + gid;

int col = colIdx[index];

float aval = matData[index];

float xval = vector\_x[col];

sum += aval \* xval;

}

vector\_y[gid] = sum;

}

This implementation does not take into consideration local memory or grouping. The approach displayed simply assigns one thread to each row and calculates the dot product between the vector and a matrix row using a private register as an accumulator. It is valid to point out that although no workgroup indication appears in the kernel, being that no local memory was used nor were workgroup specific calculations were executed, when executing a workgroup size must be defined and different sizes can still influence performance. That is because the workgroup size defines the number of wavefronts to execute, memory allocation and other factors which carry weight in determining performance.

Another technique used to extract parallelism out of the GPU is to expose vectorization by using built in vector data types. This allows the compiler to more easily identify opportunities for using vectorization capabilities, oftentimes performing multiple arithmetic operations in one cycle. Code readability does suffer by treating variables as vectors which in practice do not possess a vector representation:

uint gid = get\_global\_id(0);

if (gid < matDim) {

uint nnz = rowNnz[gid];

float4 sum = 0;

for (int i = 0; i < nnz; i += 4) {

int4 index, col;

float4 aval, xval;

index.s0 = i \* matDim + gid;

index.s1 = (i + 1) \* matDim + gid;

index.s2 = (i + 2) \* matDim + gid;

index.s3 = (i + 3) \* matDim + gid;

col.s0 = colIdx[index.s0];

col.s1 = colIdx[index.s1];

col.s2 = colIdx[index.s2];

col.s3 = colIdx[index.s3];

aval.s0 = matData[index.s0];

aval.s1 = matData[index.s1];

aval.s2 = matData[index.s2];

aval.s3 = matData[index.s3];

xval.s0 = vector\_x[col.s0];

xval.s1 = vector\_x[col.s1];

xval.s2 = vector\_x[col.s2];

xval.s3 = vector\_x[col.s3];

sum.s0 += aval.s0 \* xval.s0;

sum.s1 += aval.s1 \* xval.s1;

sum.s2 += aval.s2 \* xval.s2;

sum.s3 += aval.s3 \* xval.s3;

}

vector\_y[gid] = sum.s0 + sum.s1 + sum.s2 + sum.s3;

}

One possibility of taking advantage of local data and synchronization primitives within a local workgroup is to assign a workgroup to a row. This way, each workgroup is responsible for a dot product operation and this operation can utilize a parallel reduction. While it seems this approach would be faster, the large number of work items per workgroup means that each work item will also do less ALU operations, which can negatively affect performance. Additionally, although fewer cycles are necessary in a parallel reduction, memory latency is larger than the naïve case which uses registers and the potential performance gain also depends the number of nonzeros per column, i.e. the vector length:

uint grpid = get\_group\_id(0);

uint locid = get\_local\_id(0);

uint loclen = get\_local\_size(0);

uint numblocks = ELLwidth / loclen;

uint blockstride = loclen \* matDim;

uint slaboffset = grpid + (locid \* matDim);

if (grpid < matDim) {

auxShared[locid] = 0;

for (int i = 0; i < numblocks; i++) {

// vertical slab + row + x\_pos

int index = (i \* blockstride) + slaboffset;

int col = colIdx[index];

float aval = matData[index];

float xval = vector\_x[col];

float val = aval \* xval;

auxShared[locid] += val;

}

barrier(CLK\_LOCAL\_MEM\_FENCE);

if (get\_local\_id(0) == 0) {

float sum = 0;

for (int i = 0; i < loclen; i++) {

sum += auxShared[i];

}

vector\_y[grpid] = sum;

}

}

While sharing inside a row did not work, another alternative which commonly yields good results is to divide the problem into a tiled format. By dividing the matrix into a slab of rows and looping over these groups of rows by a fixed width, it is possible to compute the matrix vector product by visualizing these blocks as tiles. This approach increases the number of operations performed by a workgroup increasing the percentage of time the ALU is busy, an important indicator of performance, besides permitting parallel reductions:

uint gloidy = get\_global\_id(1);

uint locidx = get\_local\_id(0);

uint locidy = get\_local\_id(1);

uint loclenx = get\_local\_size(0);

// Zero out local memory

auxShared[loclenx \* locidy + locidx] = 0;

uint slabsize = matDim \* loclenx;

uint NumBlocksX = ELLwidth / loclenx;

float sum = 0;

// Loops the vertical "slab" over ELLwidth

for (int i = 0; i < NumBlocksX; ++i) {

int index = i \* slabsize + gloidy + (matDim \* locidx);

int col = colIdx[index];

float aval = matData[index];

float xval = vector\_x[col];

sum += aval \* xval;

}

auxShared[locidy \* loclenx + locidx] = sum;

// Only one thread per row reduces

if (locidx == 0) {

barrier(CLK\_LOCAL\_MEM\_FENCE);

sum = 0;

uint localbase = locidy \* loclenx;

for (int i = 0; i < loclenx; ++i) {

sum += auxShared[localbase + i];

}

barrier(CLK\_LOCAL\_MEM\_FENCE);

vector\_y[gloidy] = sum;

}

Applying the same techniques of using vector data types it is also possible to increase the efficiency of our tile based approach. Using the built in vector data types it is possible to speed up both the vector-row multiplications as the parallel reductions performed on local memory:

uint gloidy = get\_global\_id(1);

uint locidx = get\_local\_id(0);

uint locidy = get\_local\_id(1);

uint loclenx = get\_local\_size(0);

// Zero out local memory

auxShared[loclenx \* locidy + locidx] = 0;

uint slabsize = matDim \* loclenx;

uint NumBlocksX = ELLwidth / loclenx;

// Loops the vertical "slab" over ELLwidth

float4 sum4 = 0;

for (int i = 0; i < NumBlocksX; i += 4) {

int4 index, col;

float4 aval, xval;

index.x = i \* slabsize + gloidy + (matDim \* locidx);

index.y = (i + 1) \* slabsize + gloidy + (matDim \* locidx);

index.z = (i + 2) \* slabsize + gloidy + (matDim \* locidx);

index.w = (i + 3) \* slabsize + gloidy + (matDim \* locidx);

col.x = colIdx[index.x];

col.y = colIdx[index.y];

col.z = colIdx[index.z];

col.w = colIdx[index.w];

aval.x = matData[index.x];

aval.y = matData[index.y];

aval.z = matData[index.z];

aval.w = matData[index.w];

xval.x = vector\_x[col.x];

xval.y = vector\_x[col.y];

xval.z = vector\_x[col.z];

xval.w = vector\_x[col.w];

sum4 += aval \* xval;

}

auxShared[locidy \* loclenx + locidx] = sum4.x + sum4.y + sum4.z + sum4.w;

// Only one thread per row reduces

if (locidx == 0) {

barrier(CLK\_LOCAL\_MEM\_FENCE);

sum4.x = 0; sum4.y = 0; sum4.z = 0; sum4.w = 0;

uint localbase = locidy \* loclenx;

for (int i = 0; i < loclenx; i += 4) {

sum4.x += auxShared[localbase + i];

sum4.y += auxShared[localbase + i + 1];

sum4.z += auxShared[localbase + i + 2];

sum4.w += auxShared[localbase + i + 3];

}

barrier(CLK\_LOCAL\_MEM\_FENCE);

vector\_y[gloidy] = sum4.x + sum4.y + sum4.z + sum4.w;

}

1. Results

The first graphs displayed [] are relative to computing local stiffness matrices and assembling the matrices into the global stiffness matrix. Results are shown for using graph coloring to parallelize assembly in the element domain and also without coloring, enforcing that only one thread but maintaining parallelism in calculating local stiffness matrices. For assembling the global stiffness using the GPU, the GPU handled calculating local element stiffness matrices while the task of performing parallel assembly was performed by OpenMP. Since the assembly of the global stiffness matrix, also in this stage called the scattering of local stiffness matrices, is a memory intensive operation with few arithmetic operations, adding more threads quickly makes the problem memory bound and diminishes gains. If more memory operations were carried out while the GPU was still calculating part of the stiffness matrix, further gains would be potentially possible at this stage. This would also be a solution that would permit scaling to other GPUs and calculating matrices larger then GPU memory, which would be a natural development. Nevertheless, it is possible to see some gain by utilizing coloring, which was seen to have a negligible cost of calculating.

Comparing the gains in performance with the time dedicated to optimizing code, OpenMP offers much quicker gains with far fewer modifications and a greater ease maintaining. Conversely, in cases where performance is truly paramount, with OpenCL it is possible to extract the most out of the hardware at this level of abstraction. Although with OpenCL it is possible to write code that will run on a variety of different platforms, some tuning is still required given that many options on what strategy to use for parallelizing depend on the underlying hardware. Still, this extra work can be mitigated by introducing some form of auto tuning or heuristics, and much less work then rewriting for a different framework.

Much of the discussion of the validity of using GPUs for finite element analysis stem from having a problem which is not a perfect fit for the original hardware as other problems, and hence with less potential gains. The current trend towards GPUs with a greater focus on general purpose computing makes newer hardware better suited for these problems, making the switch even more appealing in the long run. Moreover, there are still many experiments as to which format or strategy can produce the highest gains, and the potential gains from developing new procedures are higher for GPUs.

One interesting result and that come out of these studies was that 3D elements, with more degrees of freedom and higher connectivity, gain more from utilizing GPUs. This happens because the number of degrees of freedom of the elements used in the mesh directly influences the matrix’s band size. Clearly, elements with more degrees of freedom with have more connections to other elements. The greater gains in elements with more degrees of freedom make the decision to use 3D, or even more complex elements within 3D, more attractive. Using more gauss points is another factor which correlates directly to a greater speedup in GPUs. Both of these factor increase arithmetic intensity for calculating the local stiffness matrices which increases the ratio of ALU operations to memory operations, which is especially important on a memory bound problem such as ours. Consequently, the marginal cost of using more complicated elements on GPUs is smaller when compared to CPUs.