Abstract

In a perfect world, the code should be written once, run on different devices with reasonable efficiency and the execution time should be inversely proportional to the compute power of newly released hardware. The time should be spend time on thinking about the algorithms, not on implementing them. That used to be the case in the era of frequency scaling on a single core. However, parallel programming has become necessary to observe any performance gains nowadays due to energy consumption limitations. Parallel architectures differ, usually require specialized knowledge, sometimes reimplemention and fine tuning for a specific device. This time-consuming task can cause a situation when most of the time is spent on reimplementing rather than devising new algorithms. The main goals of our research are to find new programming techniques increasing productivity, maintaining high performance and providing abstraction hiding unnecessary time-consuming tasks from a programmer. However, this usually comes at high performance degradation costs. This paper investigates current approaches to portable accelerator programming seeking for an answer whether it is possible to combine high efficiency with algorithm abstraction or not. If not, where would the best abstraction/performance tradeoff point be?

1 Introduction

In the world of modern hardware, parallel computing is a necessity. It allows to take full advantage of the computational power as well as reduce the overall energy consumption. The general trend is clear, architectures become more parallel, the performance of a single processing unit remains the same or even decreases due to lowering the frequency. Therefore, to simplify, it can be stated that the code parallelization should be proportional to the hardware parallelization. The important thing is the way these architectures execute the parallel code as it affects the programming techniques. With the advent of multi-core CPUs it was evident that the old, sequential code can run only faster if the work needed to be done is split between the cores and multithreaded programming model is adopted. However, an important observation is that the old code still could be compiled and executed and vice-versa. The single most important benefit of the multithreaded approach is that if the number of threads is sufficient, arbitrary number of CPU core can be utilized. Running such a program on an older CPU (more threads than cores) would actually not harm the performance significantly. So, it is safe to assume that this approach can lead to high performance portable code.

Of course, currently the commodity CPU parallelism has gone even deeper, into vector processing such as SSE or AVX, but then, using the same method, over-parallelized code will be serialized on older hardware. This leads to a conclusion that if the code written in year 2000 would have been designed in the way that it executed one hundred threads at once to complete a task, we could observe scaling and speedups even now. Recent OS releases (i.e. Windows 8, Mac OS 10.8) show a similar approach in order to take advantage of future CPUs by running hundreds of threads simultaneously. Nowadays, we are facing a similar issue. The problem is however more complex as it involves so called Accelerated Processing Units (APUs) such as FPGAs, GPUs, CELL and other coprocessors. The architectures of each of those devices are very different and so is programming them. Referring to the aforementioned example of multi-core CPUs, at this moment it is not entirely clear how to program them in way that the code compiles and runs on future accelerators without any problems. The first reason is surely the uncertainty and the other one the programming language barrier. Similarly to the multi-core CPU example, throughout the next 5 or 10 years in the perfect case one would wish to simply run previously coded algorithm on newer accelerators and see
speedup proportional to the hardware development. That would allow spending time on thinking about the algorithms rather than reimplementing them every time when new, better and possibly different hardware is released. Parallel programming is becoming complicated even on multi-core CPUs. An prediction can be even made that CPUs will eventually be more like current APUs, so the general multi-threaded approach will not work any more so efficiently as the parallelism will have to be exploited in a different way. Therefore, this problem is more general.

Furthermore, next problem that can be formulated is the ability to use heterogenous systems comprising multiple different types of processors simultaneously. By definition, APUs accelerate one or more types of computations outside of a CPU creating the first obstacle which is the APU-CPU communication and memory addressing. Currently, this kind of computation such as multi-GPU or GPU + CPU execution has to be expressed explicitly. This is especially difficult, since it extends the concept of uniform programming even further. Another emerging problem is an efficient load balancing scheme as every device can have different throughput and latency limitations. Yet another possibility might be a situation when a whole cluster of devices either physically or virtually separated is to be addressed, requiring some sort of an inter-process communication such as MPI. We are unsure about the future, so this level of parallelism cannot be excluded as future PCs might be similar more to current supercomputers.

Future compute devices can have millions of cores, compute nodes can accommodate hundreds of heterogenous devices. Currently, parallel programming is already hard enough, but the situation will not improve. In the perfect world the code should be written once, run on different devices with reasonable efficiency and the execution time on newer hardware should be inversely proportional to the compute power. Time should be focused on devising algorithms, not reimplementing nor tuning them. The main goals are to increase productivity, maintain high performance and provide an abstraction hiding unnecessary time-consuming tasks from a programmer. Some predictions can be made, such as increasing data movement cost, free compute power.

2 Related works

2.1 Halide

The first approach [1] demonstrates that both the performance and the abstraction can be achieved by separating the algorithm description, which defines the storage, and the schedule, meaning the order of computation. While maintaining the same algorithm, the program can behave differently on different devices only by altering the execution scheme. This is a very elegant method which makes the code portable and modular. The performance achieved on multiple architectures ranging from ARM to GPUs is the same as the hand tuned assembly code utilizing SIMD instructions. The only requirement for a program to be executed on new hardware is to be recompiled. However there is a very significant matter that has to be pointed out. The domain is restricted to image processing, a task which is very parallel in nature. Furthermore, such a limitation allows to tune the compiler accordingly. We find the approach very interesting and promising, the authors implemented a very impressive compiler, but this poses other questions. Can abstraction and performance be achieved at the same time only at a cost of domain restriction? Is this kind of a priori knowledge about the underlying algorithm and data structures required? If the answer is no, then what is the cost of sacrificing or more of the above? If yes, what would be the cost of implementing such a compiler for other types of problems?

2.2 Phalanx

The next way of solving the problem has been presented by Garland et al.[2]. Phalanx is an architecture-aware C++ like programming language. It addresses large-scale parallelism, heterogenous devices and complicated memory hierarchy problems. The background of this project is the Echelon (NVIDIA’s Extreme-Scale Computing Project) Processor concept introduced by NVIDIA whose architecture is more similar to current supercomputer nodes comprising both throughput (like GPU) as well as latency(like CPU) optimized cores. Such a design is being developed in order to reduce energy consumption by putting both into one chip and reducing data moving cost. On the other hand, the problem of programming such a processor arises. This research might be a clue that future commodity hardware will be more similar to current supercomputers whose programming is not easy. Therefore this task needs to be simplified. The main goals of Phalanx are the unified model for heterogeneous parallel machines, which means that there is a single notation for all processors and that is being designed for current and future machines. Another interesting feature is that it is based on C++. It is able to obtain information about the underlying hardware and memory hierarchy and execute the proper kernel accordingly. The backends includes CUDA for addressing NVIDIA GPUs, OpenMP for CPUs and GASNet for multi-node execution. The preliminary results are very promising, algorithms such as MatMul and FFT scale extremely well on clusters based on CPUs (Cray XE6, IBM Blue Gene/P) and GPUs (Cray XK6). This con-
cept almost entirely fits all the required features that we would expect from a programming language for future accelerators. We find 2 issues however, not mentioning that the results are very preliminary and the implementation details are not known. The first one is the CUDA backend, which would mean being restricted to NVIDIA GPUs. The other one is OpenMP which might not be the best choice in order to achieve very good performance. Overall, if it was replace or increase the number of backends to improve portability, that would be our language of choice.

2.3 AMP C++

C++ Accelerated Massive Parallelism (C++ AMP) is a library implemented on DirectX 11 and an open specification from Microsoft for implementing data parallelism directly in C++[3]. The C++ AMP programming model includes multidimensional arrays, indexing, memory transfer, tiling, and a mathematical function library. The single most powerful feature of C++ AMP is hiding all the low-level code from a programmer. Based heavily on C++ and abstraction an accelerator is represented by a single class and the high-level code remains the same regardless of the hardware. Vendors are responsible for proving proper implementations of the interfaces. _parallel for each_ function encapsulates parallel code making the program as short as possible. No more than a few lines of code are needed to execute a simple parallel code, making it similar to Halide, but domain independent. The schedule, or execution scheme is specified as the parameters of _parallel for each_ function (tiling). Contrary to Halide, AMP’s performance is far from perfect, suffering especially from high latencies. Our experiments have shown that although the peak performance of devices such as Radeon or NVIDIA GPUs can be achieved, there is a substantial time needed to run a simple small task, making AMP impractical in case of small problems. Other problems include Another drawback is being limited to Windows OS and Visual Studio 2012, which does not make the code platform-independent as well as early development stage causing unexpected problems such as lack of CPU support.

2.4 OpenCL

Open Computing Language (OpenCL)[4] is a well known parallel programming framework for heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), DSPs and other processors. It is based on C99 which makes it very portable. It is supported by almost any operating system. Code, once written in a parallel manner using OpenCL can be executed on a variety of devices ranging from relying by so called platforms provided by hardware vendors such as AMD, NVIDIA, Intel or IBM. The way the code is compiled and executed for a specific device is handled by the manufacturer’s low-level implementation. OpenCL organizes the programs in workgroups which run threads. Similarly to CUDA, the threads are very lightweight. Also recent research has shown[5] that if the OpenCL implementation is correctly tweaked to suit the target architecture, it performs no worse than CUDA. Our results show that the performance gap is approximately 20%-30% mainly due to loop unrolling limitations. A definite advantage of OpenCL on the other hand is its ability of execution on CPUs. We have observed speedups coming from multithreaded execution as well as auto-vectorization using Intel’s OpenCL platform. OpenCL determines what device is available and selected for execution at runtime and the code is compiled when needed or offline. This is single most powerful feature of OpenCL. Being a really open and free standard makes it very portable and available on nearly all operating systems. OpenCL offers full control over the code execution, similarly to CUDA, many calls are low-level in nature, making the code design slightly cumbersome at times. An example of Xeon Phi clearly demonstrates that new devices do and most likely will support OpenCL, while a programmer can keep the same code and enjoy speedups. This is a perfect example of portability.

2.5 Shevlin Park

Another project called Shevlin Park is being developed by Intel. It augments CLANG and LLVM[6] with C++ AMP, and uses OpenCL as its underlying compute backend. The goal of this project is to combine C++ AMP’s elegant code design and minimalism with OpenCL’s portability and relatively higher performance. The way Shevlin Park accomplishes this task is by translating AMP’s source code into OpenCL using Clang frontend and LLVM back end. As a result the program can be implemented using AMP in a very abstract way and the resulting code is in OpenCL’s form. The preliminary performance comparisons show a significant improvement over C++ AMP, but slightly (around 10% on average) worse efficiency compared to native OpenCL. It is an interesting initiative and possibly code translation is one of the ways of achieving the goal of keeping the performance at a reasonable level while maintaining the code abstraction and leaving just the minimal description of the algorithm as in C++ AMP. It is important to point out that Shevlin Park is just a concept project and that Intel has not announced any release plans for the Shevlin Park technology yet.
2.6 Pattern Language

The last concept we would like to describe briefly is the Pattern Language being developed at UC Berkeley[7]. The two main concepts is to identify key computations and structural patterns[8]. Computational patterns (Key computations) include problems such as N-Body, Dense Matrix Algebra or Dynamic Programming. Structural patterns identify the software structure such as MapReduce or Pipe-and-Filter. Identifying those patterns and learning how to solve each of them efficiently leads to a modular design possibly resulting in specialized libraries or methods which can be quickly applied to solve a problem. This way to approaching the problem is slightly different than the previous ones. It is assumed that a single programming language is not able to perform well. The main idea is that key to the design of parallel programs is software architecture, and the key to efficient software implementation is frameworks.

3 Proposed method

As we presented in the previous section, there are many ongoing projects at the moment. The key question that has to be asked is what would a programmer want or what kind of tool would improve the productivity to the maximum. This is of course done with the assumption that three conditions can be satisfied at the same time: the algorithm is abstract, code is always fast (not necessarily at peak performance), the domain is not restricted. In our opinion, OpenCL is the best choice at the moment as a base. (excluding Phalanx, which seems to be the best solution overall) The low level implementation is hidden from a user, provided by a vendor. Therefore it is already easy to achieve the first of the stated goal, meaning code portability on variety of devices. However, in order to achieve best performance on all types of devices, code has to exploit parallelism at all levels. For example, our experiments show that by writing code which is explicitly vectorized (works on vector data types in OpenCL) and the number of thread is sufficient, peak performance can be observed on CPUs, Intel Xeon Phi as well as NVIDIA and AMD GPUs running exactly the same code.

What currently poses the greatest challenge is utilizing all of the devices at once. Is it possible to make it more productive? Currently the only method is to distribute work explicitly across the available OpenCL devices. Our idea is to be able to use a Virtual Node Device[1], accessible like any other OpenCL hardware, but hiding the task distribution from the programmer.

Normally seen as:

Device 0: CPU(s)
Device 1: NVIDIA GPU
Device 2: AMD GPU
Device 3: Something else

Goal:

Device 0: Node

![Virtual Device (Node)](image)

There are many problems with such a concept. One of them is the load imbalance. This has been researched by other groups, for example Chen et al.[9] or de La Lama et al.[10] who achieve good speedups on multi-GPU systems. Our goal is to extend this to heterogeneous environment. One of the ways of accomplishing this task is by predicting the expected performance of a device. Regardless of the architecture, we believe that certain things will remain constant and specific data concerning a device can be obtained. This includes:

- Number of processors
- Number of cores per processor
- Global memory (seen by all the processors)
- Local memory (within one processor or thread)
- Cache - size and hierarchy
- Memory hierarchy - Clock frequency

This kind of information can be used to model the work distribute in a smart way. Additional problems arise unfortunately. There are usually 3 stages of execution - (a) Copying data to the device, (b) Kernel execution, (c)
Copying data from the device. In our opinion this requires a sort of a virtual address space, but it is unclear to us if it should be handled at the OS or OpenCL level. The Virtual Node device should take other OpenCL platforms as its backends, hiding this from the user. It should be programmable as any other OpenCL device. How good can the performance be? We expect that being heavily dependent on the task distribution model.

4 Acknowledgments

This work was supported by Core Research of Evolutional Science and Technology (CREST) project of Japan Science and Technology Agency (JST) and Grant-in-Aid for Scientific Research of MEXT Japan.

5 Conclusion

This paper investigated current approaches to portable accelerator programming seeking for an answer whether it is possible to combine high efficiency with algorithm abstraction or not. In our opinion, C++ AMP presents the perfect approach to abstraction, Halide’s performance is unbeatable, Phalanx is both fast and addresses the problem of heterogenous simultaneous code execution on variety of devices using multiple nodes. The overall performance is however unknown. OpenCL is the perfect example of code portability, relying on hardware vendors providing proper OpenCL platforms. We also introduced our concept of a Virtual OpenCL Node. However, the main question (How to get abstraction and performance at the same time?) remains unanswered and in addition to that new ones can be formed - Should we be domain restricted? (i.e. image processing, graph traversal). Can the compilation time be increased to get better performance? What would a typical programmer want? What should be tuned at the compilation stage and what later? If we cannot get both the performance and abstraction, where should the tradeoff point be located (see Figure 2)? Should specialized libraries be used rather than a programming language? Auto tuning - what should be tuned at the compilation time and what during the execution? How to implement the Virtual Node OpenCL device efficiently?

References