Compiler optimizations for directive-based programming for accelerators

Matias Piispanen

Abstract—Parallel programming is difficult. For regular computation on central processing units application programming interfaces such as OpenMP, which augment normal sequential programs with preprocessor directives to achieve parallelism, have proven to be easy for programmers and they provide good multithreaded performance. OpenACC is a fork of the OpenMP project, which aims to provide a similar interface for heterogeneous multiproducting as an alternative for low-level programming models such as CUDA and OpenCL.

In this paper the general structure of existing compilers for directive-based programming for GPU’s is presented and some used program analysis and optimization methods are explained. Just like with low-level programming models, optimizing the parallel regions for efficient memory usage is the most important task to achieve good performance.

I. INTRODUCTION

In recent years the development of the performance of computer processors has stopped to follow the so called Moore’s Law due to the laws of physics limiting how far single processor performance can be pushed. Lately the trend has been to add more cores to the processors instead of pushing single-threaded performance to the limits. To fully utilize the performance of multi-core processors, programmers face the difficult task of writing multi-threaded programs. Various application programming interfaces have been developed to ease the task of writing correct and efficient parallel code for multi-core processors.

The affordability, high computational performance and energy efficiency have made graphics processing units (GPU) a very interesting and powerful platform for computation of highly parallel tasks. Very first general-purpose computing on graphics processing units (GPGPU) programs were written as computer graphics shader programs than manipulated textures that held the data of the computation. Since then many software frameworks for GPGPU have emerged, such as CUDA and OpenCL, that are better suited for writing parallel code for GPU’s. However, writing efficient parallel programs using these frameworks is probably even more difficult than writing regular multi-threaded code.

An extension to the OpenMP API has been proposed to support computation on accelerators such as GPU’s. This paper gives an introduction to what kind of program analysis and techniques are needed to build compilers for such a programming model. First the GPU hardware and directive-based programming models such as OpenMP and OpenACC are introduced. In the next chapter the general structure of a existing directive-based programming for GPU compilers is explained. In the following chapters some examples of required program analysis and compiler optimization techniques are covered.

II. GPGPU COMPUTATION MODEL [1]

There is a number of differences between GPU’s and CPU’s that need to be taken into account in order to perform efficient computation. Figure 1 shows the general architecture of Nvidia GPU’s. Other manufacturers, and even future Nvidia GPU’s, may have different architectures but in the context of analyzing the nature of GPU computation and its challenges, it is sufficient to limit us to the currently popular Nvidia Fermi architecture.

The first thing worth noting is that the GPU and CPU are connected through a PCI Express bus. This makes it important to minimize the memory transfers between host memory and GPU memory. Data must be transferred explicitly between device memories using programmed DMA.

The power of GPGPU computation follows from the sheer number of processing elements on the GPU. For example the GF100, the first Nvidia Fermi architecture GPU, consists of 16 multiprocessors and each multiprocessor consists of 32 stream processors which are the actual processing elements. In the computation model, which Nvidia calls single instruction,
multiple data (SIMT), the threads are grouped together into *warp* s of 32 threads which are computed simultaneously on the same multiprocessor. Each streaming processor executes one thread of the warp sequentially. Each thread in a warp must be executing the same code. This code is generally called a kernel function.

The threads are also grouped into *blocks*. Threads of the same block are guaranteed to be executed on the same multiprocessor. Threads in the same block can synchronize and share memory using the local software cache.

There are three major factors that should be exploited to achieve high performance from computation on GPU’s. First of all there should always be a large number of threads ready to be executed to achieve full utilization of the streaming processors. This means there should be a large number of threads in each thread-block. On the other hand the threads should utilize the local software cache as much as possible. The more software cache the threads use, the smaller the thread-blocks can be, so there’s a trade-off between cache usage and processor utilization. Finally current GPU’s can fetch 16 consecutive 32-bit words from memory in one cycle. This means the memory accesses of consecutive threads in a warp should not be strided.

### III. DIRECTIVE BASED PROGRAMMING

Open Multi-Processing (OpenMP) is an API for shared memory multiprocessing. It consists of a set of preprocessor directives that are augmented into the code to achieve parallelism. The typical usage is to mark which loops in the code can and should be executed in parallel if possible. The pragmas can also be used to specify data sharing and synchronization. The idea behind using directives is that programmers can easily parallelize the computationally heavy parts of their code and the same code can still be run sequentially if OpenMP functionality is not available.

OpenMP follows a so called Fork-Join model which is illustrated in figure 2. The master thread uses a pool of worker threads which are launched to compute the blocks of code in parallel at the beginning of the parallel region and the computation joins and synchronizes at the end of the parallel region. This is very similar to the GPGPU computation model where kernel functions are launched to be computed on the GPU and the computation synchronizes when all threads have finished. This is the motivation for extending the OpenMP programming model to accelerators.

OpenMP has proven to be an effective and easy to write programming model. OpenACC is a fork of the OpenMP project which has the aim of providing similar high-level parallel programming model for accelerators such as GPU’s. OpenACC uses similar preprocessor directives to OpenMP, which define accelerator specific parameters, such as the thread-block sizes, usage of software cache and memory transfers. The goal is to be within 10% in performance compared to programs written in low-level languages such as CUDA or OpenCL [2].

### IV. STRUCTURE OF OPENACC COMPILERS [1]

PGI Accelerator is one of the first compilers to implement OpenACC functionality. The compiler works roughly like a source-to-source compiler from C code to host code and CUDA or OpenCL kernel code would. The main steps the compiler takes are the following:

1) Determine which computations and data should be moved to the accelerator, allocate memory and copy data to the accelerator.
2) Map loop parallelism to hardware parallelism. Essentially divide loop instances into thread-blocks and represent loop indices as block and thread indices.
3) Generate kernel code and optimize for underlying hardware.

The following subsections present some examples of how the above phases have been implemented in existing compilers and what kind of analysis and techniques can be used.

#### A. Determining parallelizable parts and data transfers

In current implementations it is the programmers duty to indicate which parts of the code can be executed in parallel with directives. Incorrect markings will result in incorrectly working code.

Data-flow analysis must be performed to minimize memory transfers between host memory and accelerator memory. The first step is to find the *Use-Set* of data that is read inside kernel regions and the *Def-Set* of data that is written in the kernel regions. Data-flow analysis combined with reaching analysis can be used to reduce the amount of needed memory transfers. If a variable is in the *Use-Set* of a kernel region and the kernel region is reached, data needs to be copied into device memory. If the variable is in the *Def-Set* of a kernel region and it is used in host code afterwards, the data should be copied back to the host memory. [4]

One way to simplify kernel reaching analysis is to divide the functions of the host code into four categories [5]. This makes sure the analysis works inter-procedurally and the rest of the analysis can be done locally within functions. The four classes are the following:

- **K-functions** are the functions that contain a kernel region.
- **MK-functions** are those that do not contain a kernel region but a K-function can be reached through a chain of function calls
- **IK-functions** are the functions that can be reached through a chain of functions starting from a K-function
- **N-functions** are the rest.

This classification simplifies the inter-procedural analysis and limits the analysis space of local analysis since only functions of certain class need to be checked.

The data copying choices made by the compiler can be overridden using preprocessor directives. While it is possible to reduce memory transfers significantly using analysis, in most real world applications the programmer is required to explicitly control copying of data in order to reach maximum performance.
B. Optimizing kernel functions [6]

There are many ways to perform optimization for the generated kernel functions. Many approach the optimization by modifying the loop-nest structure. Other approaches such as polyhedral optimization methods abandon the loop-nest structure and try to modify the loop iteration order some other way. In the end most optimizations that are currently in use end up doing loop unrolling, loop reordering and loop tiling even if they do not actually function by modifying the loop nest structure.

The aim of the kernel function optimization is to end up in a situation where threads are able to use the local cache efficiently, the minimum amount of memory fetches are performed and the streaming processors are fully utilized. As stated previously, the efficient software cache usage and processor utilization can be difficult to achieve simultaneously. In the end it is a search problem to find the proper balance between them.

C. Memory coalescing

The key to high computation performance in GPGPU computation is to minimize memory accesses and utilize the fast software cache as much as possible. The first thing to do is check whether or not the memory accesses of consecutive threads can be coalesced, meaning they access data at consecutive memory locations. In such cases the GPU hardware can fetch the data of 16 threads to the L1 cache in one cycle. In case of arrays the checking needs to be done by observing the array indices. There are essentially four different types of array indices:

1) Constant index is a constant value used as an array index.
2) Predefined index is usually the absolute or relative thread id.
3) Loop index is a loop iterator that is used as an array index
4) Unresolved index is an index that cannot be resolved statically, so they are skipped during checking.

Non-coalesced memory accesses can be converted to coalesced ones in some cases. This can be done for multi-dimensional arrays if the order of the dimensions of the array can be switched so that only the lowest dimension index has a different value for 16 consecutive threads. This is the same thing as performing loop re-ordering on the sequential loop nest [4].

Non-coalesced memory accesses can also be converted into coalesced ones by manipulating the kernel function code so that the data is loaded into an array in the shared memory which locates on the L1 cache for faster access. For this method to be effective, it should be checked that the data will be used more than once during the computation.

D. Thread-block merging

After doing the checks for coalesced memory accesses and doing the conversions when possible, it is easy to check whether or not threads in different blocks are accessing the same data. It is enough to check if the address ranges of the coalesced segments overlap. In real world situations this usually happens between neighboring blocks in the X and Y directions, so it is enough to check if they are sharing any data. If data sharing occurs, thread-blocks can be merged and the kernel functions modified so that data can be loaded once into the software cache and all threads in the thread-block can use the same array in the shared memory to access the data. This minimizes the amount of memory accesses significantly. This optimization method essentially does the same thing as loop tiling.
E. Other optimizations

There are various different optimizations that can be applied to improve the performance of the kernel functions. Threads can be merged so that one thread computes the values of multiple data items which allows the reuse of registers, data can be prefetched to local memory for example when there are loops in the kernel functions and eliminating partition camping. These are only some of the possible optimizations.

In practice choosing which optimizations to apply and figuring out the optimal parameters for them becomes a search problem where the usage of local memory should be maximized, but at the same time the thread-blocks should remain large enough so that the processors can be fully utilized. The problem can be solved by using decision trees, some kind of a heuristic or empirically testing different parameters to find the optimal solution.

V. Summary

Existing application programming interfaces such as CUDA and OpenCL have offered programmers a rather low level approach to GPGPU computing. OpenACC, which is a fork of the existing OpenMP API, offers a simple alternative for GPGPU computing where normal sequential programs can be augmented with preprocessor directives that guide compilers by hinting which parts of the code can be parallelized and when data needs to be copied between host and accelerator memories. The aim is that this simple programming model could eventually reach nearly the same performance as OpenCL and CUDA programs.

The directives define the parallelizable regions and they can help with reducing required data copying between host and accelerator memory. Just like with lower level GPGPU programming models, the difficult part is optimizing the kernel regions for efficient memory usage. There are various different ways to optimize code and there is no silver bullet. Time will tell whether or not OpenACC compilers will reach their target of being able to compile programs that are within 10% of programs implemented with OpenCL or CUDA in performance.

REFERENCES