This is the author's version of the work. The definitive work was published in *Proceedings of ACACES 2009 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems, pp. 229-232, 2009.*

Optimization Flow for Algorithm Mapping on Graphics Cards

Richard Membarth[1], Frank Hannig,
Hritam Dutta, and Jürgen Teich

Hardware/Software Co-Design, Department of Computer Science
University of Erlangen-Nuremberg, Germany

ABSTRACT

Graphics card architectures provide an optimal platform for parallel execution of many number crunching loop programs, ranging from fields like image processing to linear algebra. However, it is hard to efficiently map such algorithms to the graphics hardware, even with detailed insight into the architecture. This paper presents an optimization flow for mapping algorithms to the graphics hardware. Using state of the art programming models, like CUDA for graphics cards, parallel algorithms as found in image processing can be significantly accelerated, and a high speedup can be achieved.

KEYWORDS: GPU; CUDA; Optimization

1 Introduction

Modern many-core architectures like the Cell processor and graphics processing units (GPUs) open fascinating new possibilities to boost the performance of numerically intensive algorithms in fields like image processing, linear algebra, and numerical simulation. Graphics cards manufacturers like NVIDIA and AMD discovered the potential of commodity graphics hardware as accelerators, and released high-level languages and programming models to leverage this tremendous processing power of GPUs. Programming models like CUDA, Stream Computing, or OpenCL abstract from the underlying graphics hardware architecture and allow even programmers not familiar with details of the traditional graphics hardware pipeline to write programs for the GPU.

However, acceleration on such architectures is not trivial and requires careful consideration of the architecture in order to achieve high performance. Algorithms have to be investigated and alternative approaches to solve a certain problem in parallel have to be considered. This makes in particular mapping of existing applications to graphics cards challenging when different approaches need to be taken. Often, only a fraction of the peak performance of the hardware can be achieved due to constraints of the hardware, and the nature of the algorithms. Therefore, it is crucial to follow a clear path when mapping applications to the graphics hardware. We present in this paper an optimization flow for algorithm

[1]Email: richard.membarth@cs.fau.de

mapping on graphics cards, which has been used to accelerate imaging algorithms [**?**]. This has been done using CUDA [**?**] on GPUs from NVIDIA. This optimization flow considers the capabilities of current graphics hardware and comprises measures to achieve best performance depending on the nature of the tasks to be performed.

2 Optimization Flow

Nowadays, graphics cards comprise hundreds of lightweight streaming processors that execute programs, also called *kernels*, on multiple data elements simultaneously. Several of these streaming processors are grouped together and execute commands in a SIMD way (Single Instruction, Multiple Data). In particular, data parallel problems that match this constraint are solved on graphics cards. Since the global memory has a latency of several hundred clock cycles, the streaming processors execute more than one thread at a time to hide this high latency. This requires a fine-granular partitioning of tasks in independent sub-problems that can be processed by groups of streaming processors. The scheduling of these sub-problems is done by the underlying hardware and can not be influenced. Thereby, the most critical issue is the memory access in order to feed the processors steadily with data.

To map algorithms and applications efficiently to graphics hardware, we distinguish between two types of kernels executed on the graphics card. These are *compute-bound* and *memory-bound* kernels. For each type a different optimization strategy applies. While the execution time of compute-bound kernels is determined by the speed of the processors, for memory-bound kernels the limiting factor is the memory bandwidth. However, there are measures to achieve a high throughput and good execution times for both kernel types. A flowchart of the used approach is shown in Figure 1.

First, for each task of the input application corresponding kernels are created. Afterwards, the memory access of these kernels is optimized, and the kernels are added either to a compute-bound or memory-bound kernel set. Optimizations are applied to both kernel sets and the memory access pattern of the resulting kernels is again checked. Finally, the optimized kernels are obtained and the best configuration for each kernel is determined by a configuration space exploration.

Before a kernel can be classified as memory-bound or compute-bound, the memory access has to be optimized. In particular uncoalesced memory accesses have to be eliminated, where processors of a SIMD-group access non-contiguous data elements. This may have a performance penalty in the order of one magnitude. Information about uncoalesced memory accesses can be retrieved, for instance, from hardware counters on graphics cards from NVIDIA. To determine afterwards whether a kernel is memory-bound or compute-bound, several possibilities exist. In our approach, we count the instructions for memory and computational operations and calculate the resulting memory bandwidth and the achieved GFLOPS. Comparing these numbers with the corresponding peak performance numbers, it can be determined if the kernel is memory-bound or compute-bound.

For compute-bound kernels the limiting factor is the speed of the streaming processors. Therefore, either the instruction count can be decreased, or the time required by the instructions can be reduced in order to achieve better performance. This can be done by applying techniques like invariant code motion in order to replace computations by lookup table accesses or using faster intrinsic functions for extensive operations like exponentiations or divisions. In contrast, for memory-bound kernels, the available memory bandwidth constrains

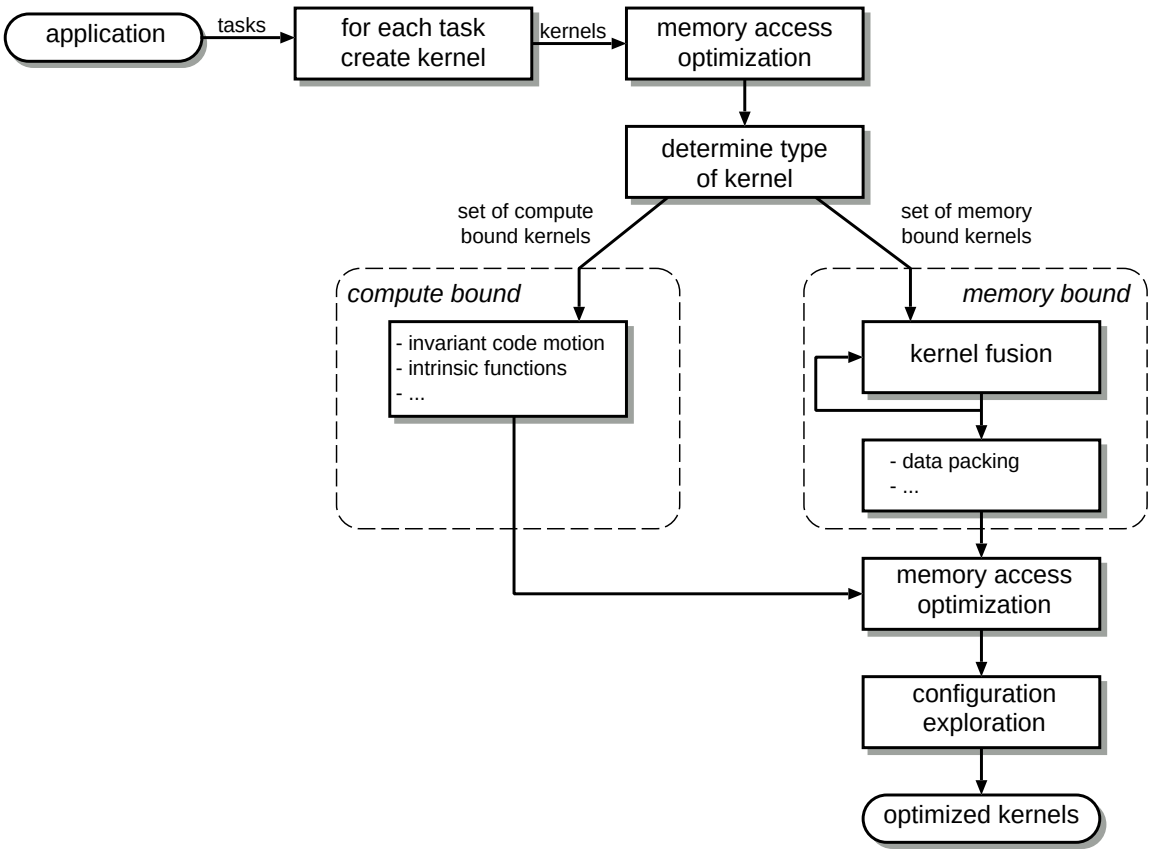

Figure 1: Flowchart of mapping strategy.

the performance. Therefore, a higher ratio of arithmetic instructions to memory accesses can reduce the required memory bandwidth. Techniques like loop fusion can be used to merge several kernels, as long as data dependencies are met. This reduces the global memory accesses, since for separate kernels the data has to be written to global memory at the end of the first kernel, and read again at the beginning of the second kernel. For certain applications like image processing, data packing can also be used to reduce the memory bandwidth. Short data types may be used to store and load data, while for the computation integer data types are used. This often provides sufficient accuracy.

After the kernels have been optimized, we explore the configuration for the kernels, which determines the tiling of the problem into smaller, independent sub-problems. The size as well as the shape of the blocks used for tiling have impact on the execution time of a kernel. Therefore, we explore all configurations with a coalesced memory access pattern to get the best execution configuration for each kernel.

3 Experimental Results

In this section, we show the results that can be achieved using the presented optimization flow for a multiresolution image filtering application with an computationally intensive filter kernel (cf. [**?**]). Table 1 shows the best speedup achieved for different measures compared to the GPU implementation without that optimization. In particular, texture memory which allows cached access to the global memory without coalescing constraints and loop fusion

have huge impact for memory-bound kernels. For compute-bound kernels the usage of intrinsics and lookup tables reduces the execution time significantly. The configuration space exploration was compared against a previously as optimal assumed configuration.

Altogether, the multiresolution application is accelerated by a factor of up to 33 on a Tesla C870, compared to parallelized baseline implementation on a Xeon Quad Core.

Table 1: Achieved speedup applying optimization for memory-bound and compute-bound kernels.

measure	bound by	speedup
texture memory	memory	3.26
loop fusion	memory	2.35
intrinsics	compute	1.51
lookup table	compute	1.39
configuration space exploration	–	1.10
loop unrolling	compute	1.02

4 Conclusion

In this paper, an optimization flow for algorithm mapping on graphics cards has been presented. For each individual task a corresponding kernel is implemented. These kernels are classified as compute-bound or memory-bound kernels, and appropriate optimizations are applied to each of these kernel sets. This results in significant speedups for the GPU implementation of up to 33 times for the considered imaging application. An implementation of this multiresolution filter as gimp plugin is available online[2], showing the impressive speedup compared to conventional CPUs.

[2] http://www12.cs.fau.de/people/membarth/cuda/