INGENIERÍA Y DESARROLLO

ARTÍCULO CIENTÍFICO / RESEARCH ARTICLE

Impact analysis of conditional and loop statements for the NVIDIA G80 architecture

Análisis del impacto de sentencias condicionales y repetitivas en la arquitectura NVIDIA G80

Snaider Carrillo*
University of Ulster (Londonderry - UK)
Jakob Siegel**
Xiaoming Li***
University of Delaware (Newark - USA)

* Beng in Electronics Engineering from Universidad del Norte, Barranquilla (Colombia). MSc in Electronics Engineering from Pontificia Universidad Javeriana, Bogotá (Colombia). PhD Student in Computer Science at University of Ulster (Londonderry - UK). *snaidercl@ieee.org*
Correspondencia: Carrera 15 N.° 27C-16, Barranquilla (Colombia).

** BEng in Software Engineering from HS Essligen: University of Applied Science (Essligen - Germany). MSc in Electrical and Computer Engineering from University of Delaware (Newark - USA). PhD Student in Electrical and Computer Engineering at University of Delaware (Newark - USA). *jsiegel@udel.edu*

*** BSc and MSc in Computer Science from Nanjing University (Shanghai - China), and PhD in Computer Science at the University of Illinois at Urbana-Champaign (Urbana-USA). Assistant Professor in the Department of Electrical and Computer Engineering at University of Delaware (Newark - USA). *xili@udel.edu*

Número 27
Enero-junio, 2010
ISSN: 0122-3461

Abstract

In this paper, we propose two novel techniques to transform control statements so they can be executed efficiently on the NVIDIA G80 architecture. Our techniques called *loop splitting* and *branch splitting* smartly increase code redundancy, which might be deemed as "de-optimization" for CPU; but for a GPU framework these techniques improve the occupancy of a program on the GPU device and therefore improve its performance. We demonstrate our optimizations on an artificial benchmark and the results show that these techniques are very efficient and, depending on the problem layout, can lead to an increase in occupancy and a drastic improvement in performance compared to *non-spli*t version of the same algorithm.

Keywords*:* Branch splitting, instruction level optimization, loop splitting, NVIDIA G80 architecture

Resumen

En este artículo se proponen y evalúan dos nuevas técnicas de optimización a nivel de instrucciones enfocadas a hacer un mejor uso de los recursos de tipo *hardware* en la arquitectura NVDIA G80. Estas técnicas llamadas *loop splitting* and *branch splitting* incrementan de forma controlada la redundancia de código, lo cual puede ser considerado como "no óptimo" en una arquitectura convencional como la CPU; sin embargo, en la arquitectura multiprocesador NVIDIA G80, dicha redundancia se ve reflejada en el incremento de la ocupación de sus multiprocesadores y en un aumento del paralelismo de los programas ejecutados en este tipo de arquitectura. Los resultados obtenidos a partir de los bancos de pruebas aleatorios y no aleatorios realizados en esta investigación muestran que estas técnicas incrementan la ocupación y el paralelismo de la arquitectura NVIDIA G80 comparado con la ejecución de la versión *non-splitting* del mismo algoritmo.

Palabras clave: Arquitectura NVIDIA G80¸ *branch-splitting*, *loop-spliting*, Optimización a nivel de instrucciones.

Fecha de recepción: 4 de agosto de 2009
Fecha de aceptación: 11 de febrero de 2010

1. INTRODUCTION

Scientific computations have always been in need of all the computational power they can get. However, high computational power is traditionally very expensive and, if available, its full usage requires extensive programming efforts. The limited availability of computational power to the High-Performance Computing (HPC) community has changed recently. Off-the-shelf hardware such as IBM Cell processors in video game consoles

(e.g. PlayStation III), and Graphic Processing Units (GPUs) in consumer PCs provides equivalent or even higher computational power than what is offered by the traditional CPU-centered high performance computing solutions (see figure 1). Until recently, it was a challenging task to implement an algorithm to run on a GPU, especially because of the functionality of such a device was plainly geared toward graphics acceleration, and did not offer an interface to perform non graphics related operations. Hence, scientific applications had to be implemented using functions and APIs such as OpenGL that are intended only for graphic tasks. In the past, this fact and the lack of downward compatibility of newer generations of graphic hardware greatly restrained the usage of GPUs for scientific computing [1], [2].

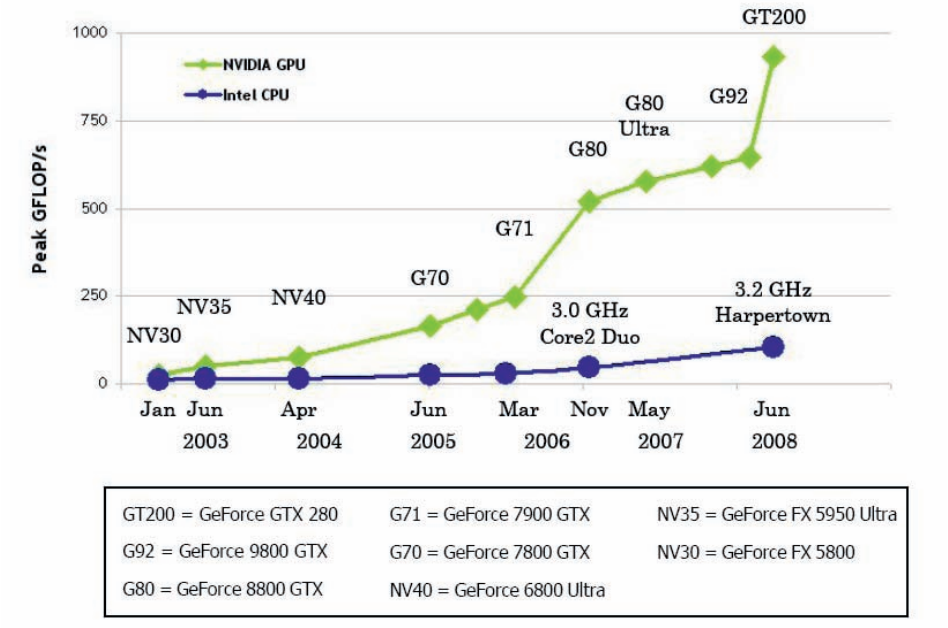

Figure 1. Comparison of Floating-Point Operations per Second for the CPU and GPU [3]

Nowadays, the introduction of the NVIDIA G80 architecture and the accompanying Compute Unified Device Architecture (CUDA) driver and C language extension [3] make this computational power of GPUs easier to utilize, taking the General-Purpose Computing on Graphic Processor Units (GPGPUs) programming to a higher level (see figure 2) [2], [3]. The CUDA driver and C language extension simplify the usage of the GPU as

a co-processing device. However, even though the problem of writing a program that can *wor*k on a GPU seems to have been solved, the question of how to tune a program to make it *wor*k *well* on a GPU is only rudimentary understood and insufficiently investigated. Most notably, the program optimization for GPUs faces two major challenges:

- The GPU memory hierarchy is organized in a radically different way from the organization of the CPU memory hierarchy. Using a GPU, the programmer has to manipulate the access patterns to increase the applications performance. It is the programmer's responsibility to maximize the throughput by optimizing the memory layout, as well as, removing bank conflct in shared memory.

- The classical instruction level optimization must be re-evaluated in the context of the GPU. Instruction level optimizations for the CPU usually assume a program can occupy all CPU resources such as registers. However, the main focus of instruction level optimization for a CUDA program is to conserve hardware resources in order to allow a higher occupancy of the available hardware for all *threads*.

Within the instruction level optimization, *loop* and *branch* statements are of special interest since most of the algorithms that qualify to be implemented in CUDA are *flow control* based. Furthermore, control statements in a program such as *loops* and *branches* pose serious challenges for the efficient usage of the GPGPU resources because those control statements will lead to the serialization of *threads* and consequently ruin the parallelism and occupancy of the GPU. Even though scientific computation programs typically have few control statements, those can easily become the performance bottleneck of a whole program.

Unlike traditional vector processing units that are inside a general purpose processor, a GPU cannot leave the control statements to the CPU, because a GPU is generally much farther away from the CPU than the vector units, hence fine-grain statement scheduling between a GPU and a CPU is impossible [4]. On the basis of this, we need an effective method to handle the control statements "just-in-place" on the GPUs.

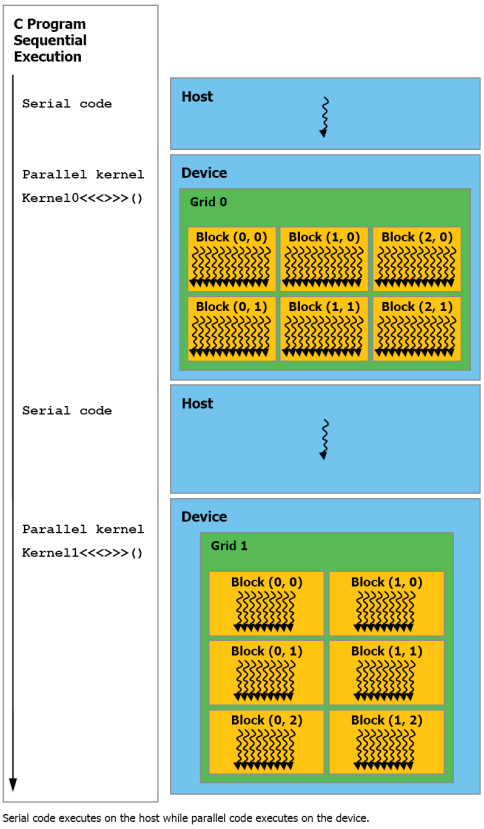

Figure 2. CUDA programming paradigm: serial code executes on the host (CPU) while parallel code executes on the device (GPU) [3]

In this paper we present and evaluate two novel instruction level optimizations that help to make better use of the hardware resources of the NVIDIA G80 architecture. These techniques called *loop splitting* and *branch splitting* smartly increase code redundancy, which might be deemed as "de-optimization" for CPU; but for a GPU framework these techniques improve the occupancy of a program on the GPU device and therefore improve its performance. We demonstrate our optimizations on an artificial benchmark and our results show that these techniques are very efficient and, depending on the problem layout, can lead to an increase in occupancy and a drastic improvement in performance compared to *non-spli*t version of the same algorithm.

The rest of the paper is organized as followed: In Section 2 we present a brief discussion about the related work. In Section 3 we give an overview of CUDA. In Section 4 we present the proposed approach to improve the occupancy and parallelism for the NVIDIA G80 architecture. In section 5 we evaluate the performance of our optimizations on an artificial benchmark. The conclusions are given in Section 6.

2. RELATED WORK

Since NVIDIA published CUDA in 2006, there has been a great interest in many different areas to apply and optimize techniques to use CUDA in an efficient way. Over the past few years the understanding and acceptance for GPGPU programming grew steadily. As a result, we can see many research projects that design new optimization techniques for GPGPUs or study the application of CPU-based optimization to the new context of GPGPUs. Limited by space, here we discuss only a small set of those projects.

Optimization for memory hierarchy is probably most extensively studied in the context of GPGPUs. For instance, *loop tiling* is probably one of the most important program transformations to improve the cache locality of a program; it has been studied under two different contexts: for the CPU and the GPGPU. The *loop tiling* for CPU is extensively studied in [5]-[9]. Studies of *loop tiling* and *unrolling* also include those that analyze the *benefit/overhead* ratio and predict the best *unrolling* factors [10]. On the GPU side, Ryoo et al. [11] study the basic principles of optimization for GPGPUs including *loop-level* optimization techniques. Moreover, Ryoo et al in [11] present how to define the overall optimization space for GPGPUs by using heuristic. Their technique is shown to be effectively to find the best optimizations for the matrix multiplication on NVIDIA CUDA.

Another important category of GPU-based research projects is the application of the high performance GPUs in real world problems. In [12] the authors give an introduction to a few representative projects of such applications. Finally, Stratton et al in [13] present a unified framework that enable a kernel to be developed only once for one GPGPU architecture but can be automatically ported to other GPGPUs and multi-core architectures [14], [15].

3. CUDA OVERVIEW

CUDA is an NVIDIA's programming model that uses GPUs for general purpose computing. It allows the programmer to write programs in C with a few extensions, which are specific to the CUDA hardware. These extensions allow the programmer to directly access the different levels of the memory hierarchy, which is quite different from the common and well known CPU *memory/cache model* (see figure 3). In this framework the programmer is responsible to take full advantage of several memory layout techniques, which will manage all levels of the GPU's explicit memory hierarchy.

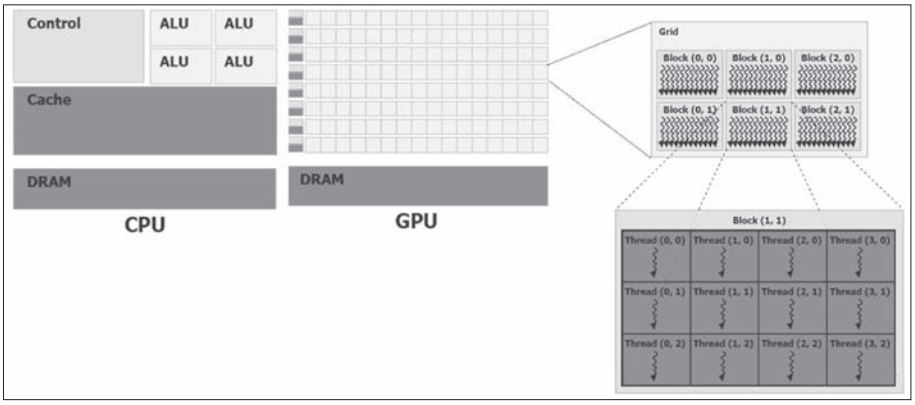

Figure 3. Comparison between a conventional CPU architecture and NVIDIA G80 architecture [3]

Occupancy

The NVIDIA G80 GPUs presents a new architecture called SIMT (single-instruction, multiple-thread) [3], allowing the multiprocessor maps each *thread* to one scalar processor core, and each scalar *thread* is executed independently with its own instruction address and register state (see figure 4). This is a cost-effective hardware model where groups of up to eight *threads* will execute the same instruction in a *thread* processing array for exploiting data parallelism. However, it can be ineffective for algorithms that require diverging control flow decisions, such as those generated from *if* and *switch* statements that can significantly impact the instruction throughput if *threads* within the same *warp* follow different *branches* [16].

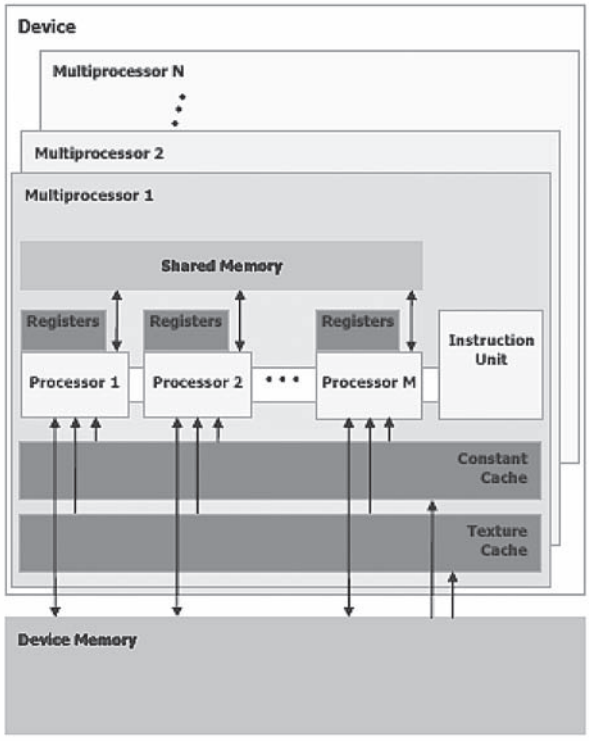

Figure 4. NVIDIA G80 hardware model: a set of multiprocessor with on-chip shared memory [3]

In some algorithms, *threads* can be reorganized to avoid divergent control flow, and logical *threads* within a block can have independent control flow throughout the program. However, for good performance on the GPU hardware, each *thread* should follow the same control flow or execution trace throughout the kernel code. The NVIDIA G80 GPU architecture executes logical *threads* in SIMT bundles called *warps*, but allows for divergence of *thread* execution using a stack-based re-convergence algorithm with masked execution [16], [17]. Therefore, logical *threads* with highly irregular control flow execute with greatly reduced efficiency compared to a group of logical threads with identical control flow.

In addition, another important challenge of The NVIDIA G80 GPU architecture is to find the optimal numbers of *threads* and *blocks* that will keep the GPU fully utilized. However, the percentage of utilization depend of

different factor such as the size of the global data set, the maximum amount of local data that *blocks* of *threads* can share, the number of *thread* processors in the GPU, and the sizes of the on-chip local memories [3]. In order to help developers perform this analysis, NVIDIA provides an Occupancy Calculator, which considers all these factors to suggest the best way to distribute the data along the kernels [2], [18].

To reach the maximum possible number of the 12,288 *threads* in 128-processor NVIDIA G80 architecture, the compiler should not assign more than about 10 registers per *thread*. However, in practice the compiler assigns 20 to 32 registers per *thread*, which limits the practical degree of concurrency, and decreases the occupancy of the kernel, which is defined as the ratio of the number of active *warps* per multiprocessor to the maximum number of active *warps*.

Table 1. Summary of the technical specification and features associated to the NVIDIA G80 GPU architecture:

Resources available	Number of Maximum Resources
Number of multiprocessors	16
Maximum number of thread per block	512
Warp size	32
Register per multiprocessor	8192
Shared memory per multiprocessor	16KB
Maximum number of active block per multiprocessor	8
Maximum number of active warp per multiprocessor	24
Maximum number of active thread per multiprocessor	768

4. PROPOSED LOOP AND BRANCH OPTIMIZATION TECHNIQUES

Modifying CUDA kernels in a way to guarantee the highest possible utilization of available computational resources is a major task for the GPGPU programmers. Having a Kernel that runs at 100% occupancy allows for

better usage of computational resources and improves the power consumption, is more efficient in hiding memory latencies and therefore gives a better performance. Two well known control structures, such as: *loops* and *branches*, can have a major effect on occupancy and the performance of a kernel which hase to be optimized [19].

There are many well known *loop* optimizations that can be applied to CPU programs as well as to CUDA Kernels. For example *loop unrolling*, to reduce the overall number of instructions needed and to allow instruction reordering, *loop interchange* and in some cases even *loop tiling* might help to improve access to the manually managed shared memory of the device. On the other hand, *branches* are one of the major factors that bring down the performance of a CUDA kernel if they are used without considering the negative effects on the SIMT architecture. If only one *thread* in a *warp* steps through the other *branch* then all the *threads* of the *warp* have to step through the instructions of both *branches* as it is the case for a common SIMD architecture [20].

In many cases the programmer can optimize the *branching* in a way that the additional number of instructions is kept low or can organize the input data in a way to guarantee that every *thread* in a *warp* has to execute the same *branch*. In this section we propose two novel techniques that can help to improve the performance of *loops* that utilize excessive memory operations and *branches* with different complexity.

Loop Splitting

A *loop splitting* is a pretty simple optimization to reduce the register pressure of a kernel. Therefore this optimization is limited to kernels that do not reach 100% occupancy because of register usage. If the kernel contains a *loop* where in the *loop* body multiple operations are performed and each operation relies on inputs that are stored in different registers (e.g. memory addresses) and at least some of those operations are independent this optimization can be applied. The *loops splitting* described here is not the same as *loop peeling* [21] which is also sometimes referred to as *loop splitting* but is a compiler optimization to remove dependencies or simplified *loops*.

The purpose of our *loop splitting* technique is to give the programmer a tool to increase the occupancy of a CUDA kernel and by doing so the overall performance [19]. In figure 5, we can split the *loop* in two *loops*, by doing so we only have to keep the parameter values *ptr1* and *ptr2* in registers for the first *loop* and *ptr3* and *ptr4* for the second *loop*. This can be done because all the pointers are parameters passed to the kernel and if we only use those parameters in the *loop* body, we don't have to load them into registers before. Therefore *ptr1* and *ptr2* are getting loaded into registers when the first *loop* is executed and *ptr3* and *ptr4* are loaded when the second *loop* is executed. This new transformation frees at least 2 registers which can in many cases give an increase in occupancy of up to 33%.

```
1  kernel (ptr1 , ptr2 , ptr3 , ptr4 , ptr_result){
       float x;
3      float y;
       …
5      for i=0 to N
         x += ptr1 [i] * ptr2 [i];
7        y += ptr3 [i] / ptr4 [i];
       end
9      …
       ptr_result = x%y;
11 }
```

```
1  kernel (ptr1 , ptr2 , ptr3 , ptr4 , ptr_result){
       float x;
3      float y;
       …
5      for i=0 to N
         x += ptr1 [i] * ptr2 [i];
7      end
       for i=0 to N
9        y += ptr3 [i] / ptr4 [i];
       end
11     …
       ptr_result = x%y;
13 }
```

Figure 5. (a) Pseudo code for a kernel that qualifies for *loop splitting* without any major changes. In the *loop* body multiple independent memory operations are performed and the pointer addresses that were passed as parameters to the kernel have to be kept in registers throughout all iterations of the *loop*. **(b)** Pseudo code for the kernel after *loops splitting* was applied.

Branch Splitting

As *loop splitting*, the general idea behind *branch splitting* is to reduce the usage of hardware resources such as registers and shared memory of a kernel or at least part of the kernel. *Branch splitting* can be applied for any kernel that does not run with 100% occupancy, works on independent data and contains *branching* where the *branches* differ in complexity and therefore in the usage of hardware resources, especially registers or shared memory. This means that if one *branch* makes excessive usage of registers or shared memory so that the occupancy drops below 100%, the whole kernel will always run with that minimal occupancy even if the *branch* that leads to the lower occupancy is never executed.

The idea is to split the *branches* of the initial kernel into two kernels, where one kernel executes only the *if-branch* and the other kernel only executes the *else-branch*. The benefit of a two kernel version is that even we have a little overhead from the additional kernel invocation we get an increase in performance since we could increase the occupancy for at least part of the initial kernel [19].

```
1  branchedkenrnel (){
       load decision mask
3      load input data used by both branches
       if decision mask [tid] == 0
5          load input data for if branch
           perform some calculations
7          using fewer registers than else -branch
           (overrall max 6 registers at the same time)
9          store result
       else if decision mask [tid] == 1
11         load input data for else branch
           perform some calculations
13         using more registers than if -branch
           (overall max. 13 registers at the same time)
15         estore result
       end if
17 }
```

```
1  ifkenrnel (){
       load decision mask
3      if decision mask [tid] == 0
           load all input data
5          perform some \label{tab : blabla}calculations
           using fewer registers than else -kernel
7          (overrall max 6 registers at the same time)
           store result
9      end if
   }
11
   elsekenrnel (){
13     load decision mask
       if decision mask [tid] == 1
15         load all input data
           perform some calculations
17         using more registers than if -kernel
           (overall max. 13 registers at the same time)
19         estore result
       end if
21 }
```

Figure 6. (a) Pseudo code for the kernel used in the single kernel version of the benchmark. **(b)** Pseudo code for the *if-branch* kernel and the *else-branch* kernel in the split kernel version of the benchmark, where the *if-branch* kernel can run at 100% occupancy compared to the *else-branch* kernel running with 67% occupancy.

The worst case scenario for using the single kernel approach is when at least one *thread* per *warp* steps through another *branch* as the rest of the *threads*, because of the SIMT architecture acts in such as way that in this case every *thread* of a *warp* has to step through the instructions of all *branches* and the device can only be utilized to the minimum occupancy defined by the *branch* with the highest usage of hardware resources. As an example we can see the figure 6a, where the arithmetic calculations are chosen so that the *if-branch* uses fewer registers than the *else-branch*. In the split version shown in *Figure 6b* the *if-kernel* uses an overall number of 6 registers compared to 13 for the *else-kernel*. This results in occupancy of 100% for the *if-branch* and 67% for the *else-branch*.

Theoretical Analysis of Branch and Loop Splitting

Experiments as the benchmark discussed in the next section have shown that this transformation in many cases can drastically improve performance.

To get an idea of the theoretical speedup for the worst case the following formula can be used:

$$speedup = \frac{T}{\sum_{i=1}^{n} \frac{t_i \times \rho_{\min}}{\rho_i} + \sigma} \quad (1)$$

Where *T*, is defined as the runtime for the worst case of the *branch*-version when the instructions of all *n–th branches* are executed. In ideal conditions, neglecting all optimizations that are applied at hardware level, this *T* can roughly be expected to be $T = \sum_{i=1}^{n} t_i \times \rho_i$. Where ρ_i is defined as the occupancy for *i–th branch* when it runs on its own, $\rho_{\min}$ is the occupancy when the *branched* version gets executed, σ is the invocation overhead produced every time a kernel is called, and t_i is the runtime of the single *branch* before the splitting.

The calculated speedup just gives an idea of what theoretic speedup can be expected if the kernel does not get limited by other factors (e.g. the memory bank conflicts). There are some more factors that might reduce the speedup or prevent this transformation of being applied. As said before, the kernel in its original setup might already have saturated the memory bandwidth where maybe the increased occupancy might help to hide part of the memory latency but as an overall the performance increase for this case might be marginal. The runtime of the single branches also plays a major role, if the kernel that might run with 100% occupancy has a runtime that is much lower than the kernel running with 67%, then the additional occupancy might not outperform the overhead added. As a guideline we can consider the following conditions for a kernel to be considered for a branch splitting [19]:

- A kernel that does not run at 100% occupancy
- A kernel that contains two or more major branches
- A kernel where the branches utilizing a different amount of hardware resources and the branches can be easily separated

5. ARTIFICIAL BENCHMARK

Benchmark Design

The benchmark performs arithmetic calculation for every single element in a fixed data set. By creating a decision mask that defines for every single data element which *branch* is taken, we have full control over the distribution of the *if-else-branch* kernel executions throughout a run. Every run of the benchmark executes the single-kernel version of *Figure 6a* and then the *split* two-kernel version of *Figure 6b* on a fixed data set of 4 million elements. We are measuring the overall runtime starting from the kernel invocation till the kernel is done for the single kernel version and compare it to the overall runtime starting from the *if-branch* kernel invocation till the *else-branch* kernel is completed. This way the measurements also include the host side (CPU) overhead for the additional kernel invocation in order to give a better picture of the overall change in performance.

To show how the distribution and density of which branch is taken effect, we run the benchmark with 2 different layouts of the decision mask: a linear decision mask in *Figure* 7, which will be explained in the sub-section 5.2, and a random decision mask *Figure 9*, which will be explained in the sub-section 5.3. Finally, the benchmark runs on an Intel dual core 2.8 GHz with 2GB of ram and two GeForce 8800 GTX GPUs where only one was used for this benchmark.

Layout 1: two section decision mask

The first run is performed with a decision mask that gets filled up from one side. Therefore we have only two sections in the mask: a growing *else-branch* section and a decreasing *if-branch* section (see figure 7). For the first iteration not one single *thread* will execute the *else-branch* kernel. For the last iteration, every *thread* executes the *else-branch* kernel.

The change in performance throughout the iterations is pretty much a connection between the two extreme cases for 0% and 100% *else-branch* executions in the first and last iteration of the benchmark (see figure 8). This can be explained by looking at what the *threads* in each single *warp* are doing.

There is only one *warp* throughout the whole problem where *threads* have to take different *branches*, which is the *warp* that contains the *threads* that are working on the boundary between the two sections of the decision mask. All the other *warps* only execute the *if-branch* or the *else-branch* kernel.

The performance differences between the extreme cases for the first and last iteration can be explained by looking at what is being executed and the level of occupancy at this moment. For the first iteration, that is 0% *else-branch* executions, every *thread* in the *branch*-version executes only the *if-branch* with 67% occupancy because the device has no way to know if all or how many *threads* in all blocks running in a multiprocessor only take the *if-branch*. For that reason the device has to consider that there are *threads* in all blocks that might take the *else-branch* and therefore the maximum possible occupancy is the one for the worst case. The *split* version drops out of the *else-branch* kernel immediately when there is only *if-branch* kernel executions needed and in this case also only executes the *if-branch* kernel for every single *thread* but each multiprocessor is running at 100% occupancy.

For the last iteration, both benchmark versions: *branched* and split, execute only the *else-branch* part with 67% occupancy.

Around this point we have the only problem layout where the overhead of the additional kernel invocation and the additional loads needed by the two kernels lead to a lower performance of the *split* version than the *branched* version, but since this is only the case for roughly 5% of this very special problem layout, the split version mostly outperforms the *branch* version of the benchmark.

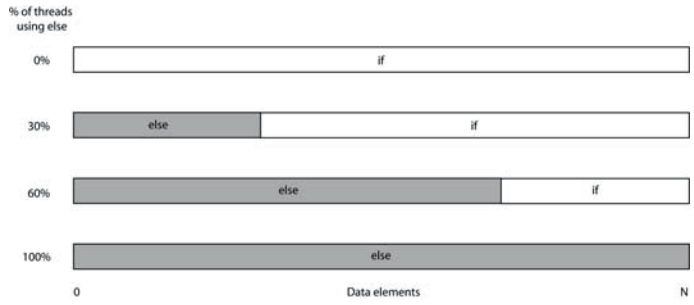

Figure 7. Linear decision mask, starting at the first iteration from no *else-branch* executions for any data element to only *else-branch* executions for the last iteration

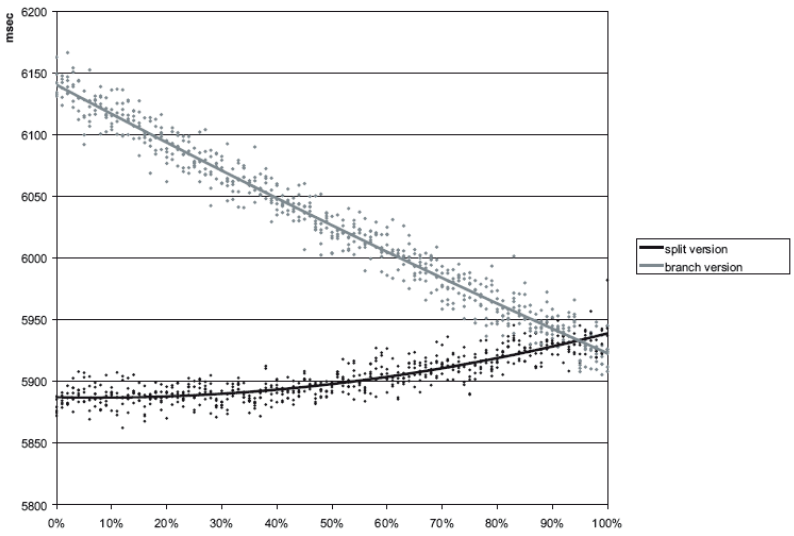

Figure 8. Change in runtime over the percentage of *else-branches* executed for the split two kernel versions and the *branch*, single kernel version of the benchmark with a decision mask.

It should be mentioned that in a case where the programmer knows that the problem will *branch* with such conditions there are better ways to improve the performance. E.g. by using two kernels with reduced input sets, so that not for every single data element has to be checked if one or the other *branch* has to be executed and only one kernel works on each section. The next section will discuss a more realistic decision mask with a random distribution.

Layout 2: random decision mask

In most common problems where *branching* is used, the condition that defines which *branch* is taken has a more randomized distribution compared to what is discussed in the previous section. To get an idea how *branch splitting* affects a problem like that, we generated a random decision mask for every possible condition in the *branches* (see figure 9). The mask is randomly initialized with more values that lead to an execution of the *else-branch* instead of the *if-branch*, starting from 0% *else-branch* executions to 100% *else-branch* executions. As can be seen, the results for the extreme cases in

figure 10 of 0% *else-branch* executions and 100% *else-branch* executions are the same as in the previous experiment, but as soon as we step away from those extreme cases we see a drastic drop in performance.

The reason for that is the earlier explained SIMT architecture of CUDA. where as soon as one *thread* in a *warp* has to step through the other *branch* all *threads* in the *warp* will execute all the instructions for both *branches*. In the previous benchmark run, only one *warp* in the entire system had to do that, the *warp* that handles the data segment where the decision mask switched form *if* to *else*. On the other hand, with this benchmark setup we have an entire different picture. For a data set of 2^{22} (4 million) elements we have $2^{22}/32 = 131072$ *warps*, only 1% of the data (41944 elements) has to be handled by the *else-branch* and in the worst case every element is placed in a different *warp*, that is 32% of the *warps* have to step through the instructions of both *branches*. This happens at both ends of the 0% to 100% run of the benchmark which explains the drastic drop in performance for the first iterations.

For the *branched* version of the benchmark the performance decreases further the farther we step away from the starting point till we reach the worst case scenarios between 8% and 16%. In these areas we have the highest probability that every *warp* at least contains one *thread* that executes the *else-branch*, forcing every single *thread* to step through the instructions of both *branches*. The split version on the other hand has a similar overall behavior except that the initial drop in performance is way not as drastic as for the *branched* version and throughout the run of the benchmark the performance of the *split* version does not change as much as for the *branch* version.

The important part here is that the execution time for the *split* version, especially for the worst case scenarios, is 14% faster than for the *branch* version (see figure 8 and figure 10). The explanation for the better performance of the *split* version again is the SIMT architecture of CUDA in combination with the different needs of resources of the *branches*. As explained earlier in this section, the *if-branch* uses fewer registers as the *else-branch*. In the *branched* version the kernel can only utilize part of the device since the kernel simultaneously uses up to 13 registers. Therefore a block size of 256 *threads* per *block* limits the occupancy of the multiprocessors to 67%. In the *split* version the *else-branch* kernel still uses 13 registers and runs at 67% occu-

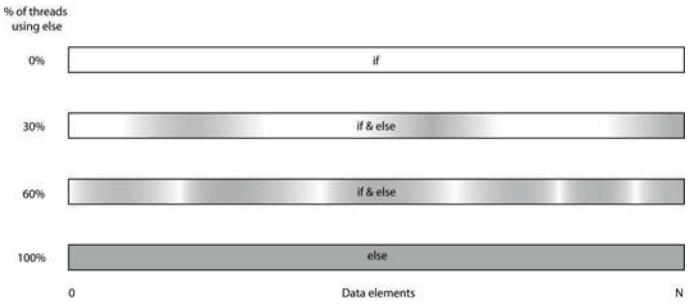

Figure 9. Symbolic figure of the random decision mask, starting at the first iteration with no or 0% *else-branch* executions for any data element to only or 100% *else-branch* executions for the last iteration. The mask is filled with a random number generator.

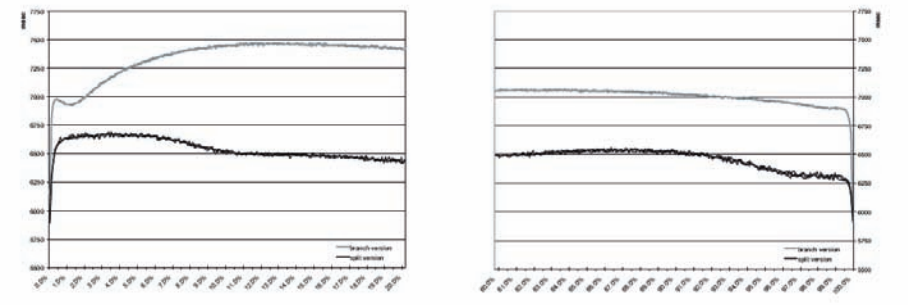

Figure 10. Benchmark results for a random decision mask showing the runtime in milliseconds over the probability of the *else-branch* taken. The worst case scenario for the initial branched version is located in the area between 8% and 16% else branch executions. In this area it is pretty much guaranteed that at least one *thread* per *warp* executes the *else-branch*. The overall performance of the *branched* version increases slightly the more else branches get executed because of the reduced number of *if-branch* executions at the lower occupancy. Still it can be seen that throughout 98% of the benchmark the split version outperforms the *branch* version by 6% to 13.5%.

pancy but the *if-branch* kernel uses just 6 registers which allows this kernel to fully utilize all the devices computational power by running at 100% occupancy. These facts already explain the increase in performance of the previous benchmark layout, but for the random decision mask this factor becomes way more important in combination with the SIMT architecture.

In the previous non randomized benchmark explained in the sub-section 5.2, all *warps* except one executed only the *if-branch* or the *else-branch* kernel. Now we have a completely different setup. For this benchmark every *warp* that contains two *threads* that take different *branches* will have to execute the instructions of both *branches*. As said before, in the worst case section between 8% and 16% of probability of the *else-branch* executions is the highest, because every *warp* contains at least one or more *threads* executing in the *else-branch*. Here we can say that for all 2^{22} *threads* where each works on one of the 2^{22} elements, the *branched* as well as the *split-version*, all instructions of both *branches* or both kernels have to be executed. The big difference is that the *branched* version is limited to utilize 67% of the devices resources for both *branches*, whereas the split version executes the *if-branch* kernel at 100% and the *else-branch* kernel at 67%. If we use the equation 1 explained in the sub-section 4.3, we can say that the *branches* in the original *branched* version run roughly at the same speed, only if we neglect the overhead and assuming that $T = \sum_{i=1}^{n} t_i$.

$$speedup = \frac{T}{\sum_{i=1}^{n} \frac{t_i \times \rho_{\min}}{\rho_i} + \sigma} = \frac{2}{\sum_{i=1}^{n} \frac{1 \times 0.67}{1} + \frac{1 \times 1}{1} + 0} \approx 19.7\% \quad (2)$$

This represents an average speed-up compared to the worst case scenario which is in the area where 8% to 16% of the threads have to use the branch with the lower occupancy. Till now it is only possible to give an estimate of the speed-up for a problem where the branches are executed in a random fashion distributed equally over the threads. Furthermore the final result may be influenced by the other hardware limitations such as bank conflict in shared memory. Those limitations and how they influence the overall performance still has to be considered for every single problem.

6. CONCLUSION

To run kernels with the highest possible occupancy is one of the major tasks for any GPGPU programmer. Any transformation or optimization that can reduce the usage of hardware resources that reduce the occupancy are a big contribution to the overall performance of a GPGPU program execution. Our

loop and *branch* transformation helps to increase the occupancy and parallelism for some special cases of *loops* and a more general case of *branches*.

The *loop splitting* is an example for a transformation that might seem counterproductive on most other architectures than a GPU, but here where occupancy is a major player in the performance game, it can have a positive impact on the overall performance for the NVIDIA G80 Architecture.

In any case *branches* are not a good thing to use in any SIMD or SIMT architecture, but for some algorithms there are not that many other efficient ways to implement them without using *branching*. Therefore optimizing *branches* in a way that either the number of instructions per *branch* gets minimized or the input data set reduces the probability of *warps* where both *branches* have to be executed is also a major task for GPGPU programmers; especially when we know that in many cases there is no way to prevent both *branches* from being executed within a *warp*, and most of this *branches* differ in complexity and therefore in the usage of hardware resources. In those cases, *branch splitting* is a promising transformation that can drastically improve the performance of a GPGPU application.

References

[1] M. Woo, J. Neider, T. Davis, and D. Shreiner. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Boston, MA: Addison-Wesly, 2005.

[2] T. Halfhill. "Parallel processing with CUDA: Nvidia's high-performance computing platform uses massive multithreading". *Microprocessor Report.* Vol. 22, pp.1-8, January 28, 2008.

[3] NVIDIA. Compute Unified Device Architecture Programming Guide. NVIDIA: Santa Clara (CA): NVIDIA, 2007.

[4] E. Kilgariff, R. Fernando. *GPU Gem 2.* Santa Clara (CA): Addison-Wesly, 2006.

[5] G. Goumas, A. Sotiropoulos, and N. Koziris. "Minimizing completion time for loop tiling with computation and communication overlapping". 15th International Parallel and Distributed Processing Symposium, Apr, 2001.

[6] J. Xue, Q. Huang, and M. Guo. "Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences". In *Proc. International Conference on Parallel Processing*, ICPP, 2005. pp. 107- 115.

[7] F. Rastello and Y. Robert. "Automatic partitioning of parallel loops with parallelepiped-shaped tiles", *IEEE Transactions on Parallel and Distributed Systems,* vol. 13, n.° 5, pp. 460-470, 2002.

[8] K. Hogstedt, L. Carter, and J. Ferrante. "On the parallel execution time of tiled loops". *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, n.° 3, pp. 307- 321, March 2003.

[9] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzar´an, D. Padua, K. Pingali, P. Stodghill, and P. Wu. "A comparison of empirical and model-driven optimization". In *Proc. of Programing Language Design and Implementation*, June 2003, pp. 63–76.

[10] P. Kisubi, P. Knijnenburg, and M. O'Boyle. "The effect of cache models on iterative compilation for combined tiling and unrolling". In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2000, pp. 237–246.

[11] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Wen-mei. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA". *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73–82.

[12] H. Nguyen. *GPU Gem 3,* Santa Clara, CA: Addison-Wesly, 2007.

[13] J. Stratton, S. Stone, and W. Wen-mei. MCUDA: An efficient implementation of CUDA Kernels on multi-cores. Center for Reliable and High-Performance Computing, 2008.

[14] Open64. http://www.open64.net.

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, vol. 28, pp. 39–55, 2008.

[16] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Wen-mei. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73–82.

[17] J. Stratton, S. Stone, and W. Wen-mei. MCUDA: An efficient implementation of CUDA Kernels on multi-cores. Center for Reliable and High-Performance Computing, 2008.

[18] S. Che, J. Meng, J. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors. The First Workshop on General Purpose Processing on Graphics Processing Units, October, 2007.

[19] S. Carrillo, J. Siegel, and X. Li. 2009. A control-structure splitting optimization for GPGPU. In *Proceedings of the 6th ACM Conference on Computing Frontiers*. Ischia (Italy), 2009, May 18-20.

[20] Hennessy, D. Patterson. *Computer Architecture: A Quantitative Approach,* 4 ed. San Francisco, CA: Morgan Kaufmann, MA, 2006.

[21] L. Ravi. "A technique for variable dependence driven loop peeling". *In Fifth International Conference on Algorithms and Architectures for Parallel Processing*: October 23-25, 2002, Beijing (China).