A Control-structure Splitting Optimization for GPGPU

Snaider Carrillo Jakob Siegel Xiaoming Li
ECE Department
University of Delaware, USA

ABSTRACT

Control statements in a GPU program such as loops and branches pose serious challenges for the efficient usage of GPU resources because those control statements will lead to the serialization of threads and consequently ruin the occupancy of GPU, that is, the number of threads running concurrently. Unlike traditional vector processing units that are inside a general purpose processor, the GPU cannot leave the control statements to the CPU because fine-grain statement scheduling between GPU and CPU is impossible. We need an effective method to handle the control statements "just in place" on the GPUs.

In this paper, we propose novel techniques to transform control statements so that they can be executed efficiently on GPUs. Our techniques smartly increase code redundancy, which might be deemed as "de-optimization" for CPU, to improve the occupancy of a program on GPU and therefore improve performance. We focus our attention on how common programming structures such as loops and branches decrease the occupancy of single kernels and how to counter that. We demonstrate our optimizations on a synthetic benchmark and a complex parallel algorithm, the Lattice Boltzmann Method (LBM). Our results show that these techniques are very efficient and can lead to an increase in occupancy and a drastic improvement in performance compared to non-split version of the programs.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Performance

Keywords

CUDA, GPGPU, optimizations

1. INTRODUCTION

GPUs have become the most powerful computation devices in modern of-the-shelf PCs. Until recently, it was a challenge to implement an algorithm efficiently to run on a GPU because the functionality of such a device was plainly geared toward graphics acceleration, and didn't offer an interface to perform non-graphics operations. The introduction of the Compute Unified Device Architecture (CUDA) programming framework [1] makes the computational power of GPUs easier to utilize. However, even though the problem of writing a program that can *work* on a GPU seems to have been solved, the question of how to tune a program to make it *work well* on a GPU is only rudimentary understood and insufficiently investigated.

A detailed description about CUDA and its supporting hardware can be found in [1]. Here we briefly discuss the most important factors that impact the performance of CUDA programs. The factors are new and unique to CUDA, hence are not considered in the traditional CPU optimization techniques.

The CUDA defines a new architecture called SIMT (single-instruction, multiple-thread)[1] which allows a multiprocessor GPU chip map an individual thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state. This is a cost-effective hardware model to exploit data parallelism. The SIMT architecture can be ineffective for algorithms that require diverging control flow decisions, such as those generated from *if-else* statements, because the concurrency among threads will be reduced if threads within the same thread group (called *warp* in CUDA) follow different branches[2].

One of the main tasks of optimizing a program for CUDA is find the optimal numbers of threads and blocks that will keep the GPU fully occupied. Factors affecting the resource occupancy include 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[1]. One important limit of occupancy of a program is the number of registers each thread of the program requires. For example, to reach the maximum possible number of 12,288 active threads in a 128-processor GeForce 8, the compiler cannot assign more than 10 registers per thread. However, the CUDA compiler usually over-assigns registers per thread, which decrease the occupancy of the kernel, because the CUDA compiler tries to optimize the single-thread performance while ignoring the overall resource pressure of a multi-thread program.

This paper presents new instruction level transformation techniques that improve the utilization of hardware resources of the NVIDIA CUDA platform. Our techniques are novel applications of seemingly common program transformations. In other words, they smartly increase code redundancy, which might be deemed as "deoptimization" for CPU, to improve the occupancy of a program on GPU and therefore improve performance.

2. LOOP AND BRANCH OPTIMIZATION

Optimizing a CUDA kernel for better occupancy allows for better usage of the devices computational resources and better hiding of memory latency, and usually gives a better performance. The basic idea of our techniques is freeing hardware resources by purposefully *increasing* code size by splitting common control structures.

2.1 Loop Splitting

Loop splitting or Loop fission is a simple optimization that breaks a loop into two or more smaller loops. Loop splitting is especially useful for reducing the register pressure of a CUDA kernel, which can be translated to better occupancy and overall performance improvement. If a kernel contains a loop where in the loop body multiple operations are performed and each operation relies on different inputs and those operations are independent, this optimization can be applied. The splitting leads to smaller loop bodies and hence reduces the loop register pressure. Therefore, this optimizations is applicable to kernels that don't reach 100% occupancy because of register usage.

Figure 1 left, shows a pseudo code segment from a CUDA kernel where we can split the loop. After splitting, only ptr1 and ptr2 have to be kept 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 the data into registers before the actual usage. 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 frees at least 2 register which can in many cases give an increase in occupancy of up to 33%.

2.2 Branch Splitting

As with 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. Branch splitting is only useful for kernels that don't run with 100% occupancy. In particular, even only one branch of an if statement has lower occupancy, e.g., using excessive registers or shared memory, the whole if statement will always run with that lower occupancy even if the branch that leads to the lower occupancy is never executed.

The branch splitting technique splits a branch of the initial kernel into two kernels, where one kernel executes only the if-branch and the other kernel only executes the else branch. Even that additional kernel invocations incur overhead, we get an increase in performance since the occupancy got increased for at least part of the initial kernel.

The best scenario for branch splitting, that is, the worst case for using the single kernel approach, is when at least two threads in a warp execute different branches. Because in the SIMT architecture, 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. After splitting, both the kernels still have to be executed. However, instead of running both branches with the lower occupancy of the two branches, now the kernel with the branch that uses fewer hardware resources can be executed with a higher occupancy. Especially for kernels that work on large datasets and where every data element is handled by a single thread which is pretty common for CUDA, the increase in occupancy can easily outperform the overhead of the additional kernel invocation. To calculate the theoretical maximum speedup for branch splitting, the following formula can be used:

$$speedup = \frac{T}{\sum_{i=1}^{n} \frac{t_i}{\frac{\rho_i}{\rho_{\min}}} + overhead} \tag{1}$$

T is the runtime for the worst case of the branch-version when the instructions of all branches n are executed. In ideal conditions, neglecting all optimizations that are applied at hardware level, T can roughly be expected to be $T = \sum_{i=1}^{n} t_i$. ρ_i is the occupancy possible for branch i when it would run on its own, $\rho_{\min}$ is the occupancy the branched version gets executed with and t_i is the runtime of the single branch before the splitting. As an example we assume that the branches in the original branched version run roughly at the same speed and $T = sum_{i=1}^{n} t_i$ and the occupancy of one branch is limited to 67% while the other branch could run at 100%. When neglecting the overhead, the maximum speedup is 19.7%. If the branch that might run with higher occupancy has a runtime that is much lower than the low occupancy branch, the additional occupancy might fail to compensate for the overhead incurred by the branch splitting. As a guideline, we use the following rules to determine if a kernel with branch statements will benefit from the branch splitting transformation: A kernel (1) does not run at 100% occupancy; (2) contains two or more branches; and (3) has branches utilizing different amounts of hardware resources.

3. SYNTHETIC BENCHMARK AND APPLICATION

The probability of how often either branch of an "if-else" statement is taken plays a major role in the overall performance of the transformed code. To show how the performance changes we designed a synthetic benchmark where we have full control over which branch is taken by a decision mask. The benchmarks are synthesized in a way that the if-branch uses fewer registers than the else-branch. This allows for a possible occupancy of 100% for the if- and 67% for the else-branch. The benchmark works on a fixed data set of 4 million elements. The overall runtime for the kernel executions is measured. For the split version the measurements also include the host-side overhead for the additional kernel invocation to give a fair comparison of the overall change in performance. To show how the distribution and density of which branch is taken affects the performance we synthesize 2 different layouts of the decision mask, a linear decision mask and a random decision mask as illustrated in *Figure 3*. Figure 2 show the pseudo code of the synthetic kernel and its split version respectively. The benchmark runs on a Intel dual core 2.8 GHz with 2GB of ram and two GeForce 8800 GTX GPUs where only one was used for this benchmark.

3.1 Layout 1: two section decision masks

First a decision mask that gets filled up from one side, that is, the else-branch taken probability is increased steadily from 0% to 100% is used, *Figure 3 left*. Therefore we have only two sections in the mask: a growing else-branch section and a decreasing if-branch section . For the first iteration none thread will execute the else-branch or else-kernel. For the last iteration every thread executes the else-branch or else-kernel.

The change in performance throughout the iterations is a smooth transition between the two extreme cases of 0% and 100% else-branch executions in the first and the last iteration, *Figure 4*. This can be explained by looking at what the threads in each 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 on the boundary between the two sections of the decision mask. All the other warps only execute the if- or the else-branch/kernel. For the first iteration, 0% else-branch executions, the branch-version executes only the if branch but only with with 67% occupancy because the device has no way to know if all or how many threads in all blocks running in a multiprocessor take the if-branch. For that reason the device has to subscribe resources assuming that all blocks 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-kernel immediately when there are only if-kernel executions and therefore in this case each multiprocessor is running at 100% occupancy. For the last iteration, both versions execute only the else part with 67% occupancy. Only at this point the overhead of additional kernel invocations and the additional loads needed by the two kernels of the split version might lead to a lower performance than the branched version. For this decision mask layout, the unfavorable scenarios only account for roughly 5% of all cases.

```
kernel(ptr1,ptr2,ptr3,ptr4,ptr_result){
  ...
  for i=0 to N
    x += ptr1[i] * ptr2[i];
    y += ptr3[i] / ptr4[i];
  end
  ...
}
```

```
kernel(ptr1,ptr2,ptr3,ptr4,ptr_result){
  ...
  for i=0 to N
    x += ptr1[i] * ptr2[i];
  end
  for i=0 to N
    y += ptr3[i] / ptr4[i];
  end
  ...
}
```

Figure 1: ***left:*** **Pseudo code for a kernel that qualifies for loop splitting.** ***right:*** **The same code after loop splitting.**

```
branchedkenrnel(){
  load decision mask
  load input data used by both branches
  if decision mask[tid] == 0
    load input data for if branch
    perform calculations using 6 registers
  else if decision mask[tid] == 1
    load  input data for else branch
    perform calculations using 13 register
  end if
}
```

```
ifkernel(){
  load decision mask
  if decision mask[tid] == 0
    load all input data
    perform calculations using 6 registers
  end if
}

elsekernel(){
  load decision mask
  if decision mask[tid] == 1
    load all input data
    perform calculations using 13 register
  end if
}
```

Figure 2: Pseudo codes for the single kernel version (*left*) and the split/two-kernel version (*right*).

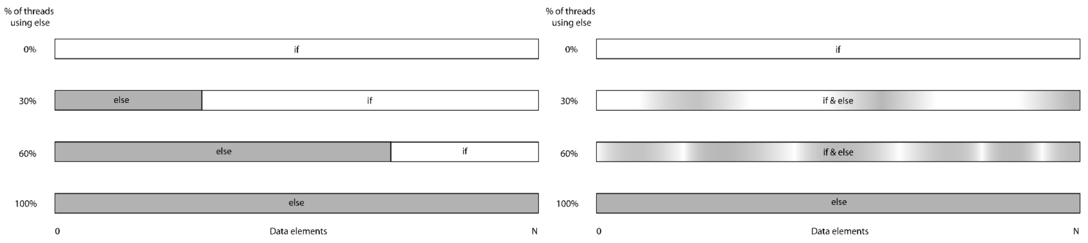

Figure 3: ***left:*** **Linear decision mask. Starting at the first iteration from no else-branch executions for any data element to only else-branch executions for the last iteration.** ***right:*** **Symbolic representation of the random decision mask.**

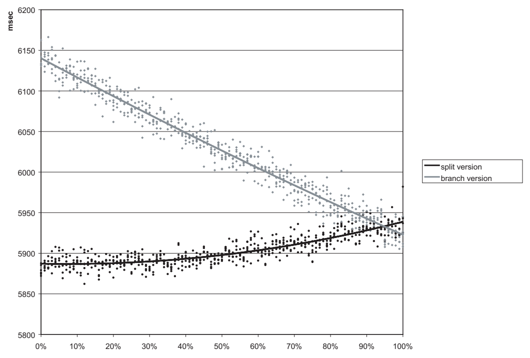

Figure 4: Runtime in msec vs. the probability of else-branches being taken with the sectional decision mask.

Even that a distribution of the if - else condition that we defined with this decision mask is not the most common for algorithms that are written for CUDA we still got a good speedup for more than 95% of the tested distributions. 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, for example, using two kernels with reduced input sets, so that not every single data element has to be checked about which branch has to be executed and hence only one kernel works on each section. The next section will discuss a more realistic decision mask with a random distribution.

3.2 Layout 2: random decision mask

Generally the condition that defines which branch is taken has a more randomized distribution compared to what is discussed in the previous section. We generate a random decision mask that is randomly initialized in every iteration with increasing probabilities of executing the else branch. The probability starts from 0% else-branch executions to 100% else-branch executions. As can be seen *Figure 5*, the results for the extreme cases of 0% else-branch executions and 100% else branch executions are the same as in the previous experiment. But as soon as the we step away from those extreme cases we see a drastic drop in performance. The reason for the drop is the serialization of branch statement in the 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 case of the sectional decision mask, only one warp in the entire system had to do that, the warp that handles the data segment where the decision mask switched from if to else. With this random decision mask we have an entirely 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, 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 few iterations.

For the branched-version of the benchmark the performance decreases more the farther we step away from the starting point till we reach the worst case scenarios between 8% and 16% . In this area we have the highest probability that every warp at least contains one thread that executes the else-

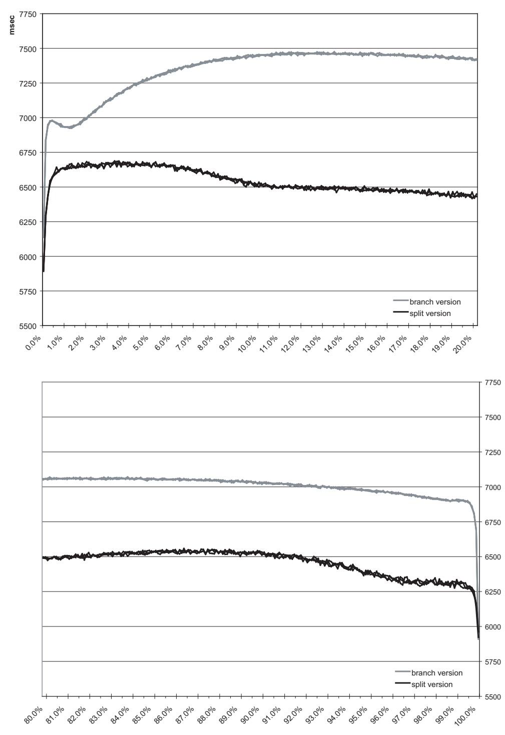

Figure 5: Runtime in msec vs. the probability of the else-branch being taken with a random decision mask. The worst scenario for the branched version is in the area between 8% and 16%, where at least on thread per warp executes the else branch. In 98% of the cases, the split version outperforms the branch version by 6% to 13.5%.

branch. This forces every single thread to step through the instructions of both branches.

The split version on the other hand performs with a very similar overall pattern except that the initial drop in performance is much smaller than that of the branched version and that throughout the runs, the performance of the split version does not change as much as for the branch version. The performance for the split version especially for the worst case scenarios is 14% better than that for the branch-version.

Two factors can explain the better performance of the split version: the reduction of the serialization of branch statements and the lowering of the resource usage per thread. In the branched version the usage of 13 registers and the size of 256 threads per block limit the occupancy of the multiprocessors to 67%. In the split version the else-kernel still uses 13 registers and runs at 67% occupancy but the if-kernel uses just 6 registers which allows this kernel to fully utilize all the devices computational power by running at 100% occupancy.

Clearly, there are so many other layouts that we cannot discuss all of them in this paper. However, the performance gain analysis and the determination of whether or not to apply control structure splitting are the same. For most cases, the control structure splitting optimization will improve performance. Furthermore, branches in a program are frequently inter-dependent. That knowledge can be leveraged to further enhance performance.

3.3 In Lattice Boltzmann Method

We applied our control-structure splitting optimizations on a function of a complex parallel algorithm, the Lattice Boltzmann Method (LBM)[4][3]. To separate the performance improvement of our optimizations from other optimizations, we use a baseline implementation of LBM that has been already optimized for CUDA using common optimization techniques. One of the most computational intensive kernels of our LBM algorithm is the one that makes each thread work on a fluid node but perform different operations for each neighboring node according to its type: fluid or fluid wraparound boundary nodes. The branch that handles the particles streaming inside the lattice uses slightly more hardware resources than the branch that handles the streams to the wraparound buffers. By splitting the loop and the branches of this kernel, the number of used registers is reduced, which allows the split kernels to run at 100% occupancy. This leads to a significant gain in performance and a more than 60% reduction of the execution time, *Figure 6*.

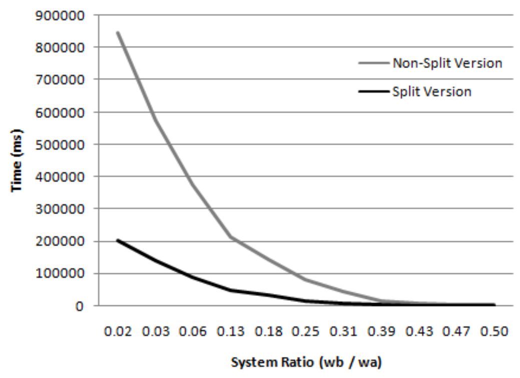

Figure 6: Runtime of LBM on a $256 * 256$ problem layout

4. CONCLUSION

Optimizing GPU programs to achieve the highest possible occupancy is one of the major tasks for any GPGPU programmer. In this paper we propose loop-splitting and branch-splitting transformation that increase the occupancy loops and branches in a problem. The transformations might seem counter productive on most architectures other than a GPU, but on GPU where occupancy is a major factor for performance, they can have a positive impact on the overall performance. Furthermore, branches are generally a performance bottleneck in any SIMD or SIMT architecture because in many cases there is no way to prevent the execution of both branches within a warp which is a major hurdle for the performance on GPU program. In those cases branch splitting is a promising transformation to increase the occupancy. We demonstrate our control-structure splitting transformation in a synthetic benchmark and a complex parallel algorithm, the LBM algorithm. The experiment results show that our transformations add negligible overhead but in most cases can significantly improve the performance of a GPGPU application.

Acknowledgment: This project is supported by NSF grant CNS0719909.

5. REFERENCES

[1] C. NVIDIA. Compute Unified Device Architecture Programming Guide. *NVIDIA: Santa Clara, CA*, 2007.

[2] 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*, pages 73–82, 2008.

[3] S. Succi. The Lattice Boltzmann Equation for Fluid Dynamics and Beyond. 2001.

[4] Y. Zhao. Lattice Boltzmann based PDE solver on the GPU. *The Visual Computer*, 24(5):323–333, 2008.