Performance Debugging of GPGPU Applications with the Divergence Map

Bruno Coutinho  Diogo Sampaio  Fernando M. Q. Pereira  Wagner Meira Jr.
Departamento de Ciência da Computação  Universidade Federal de Minas Gerais, Brazil
{coutinho,fpereira,meira}@dcc.ufmg.br, dnsampaio@gmail.com
June 7, 2010

Abstract

The increasing programability and the high computational power of Graphical Processing Units (GPU) make them attractive to general purpose programming. However, taking full benefit of this execution environment is a challenging task. One of these challenges stem from divergences, a phenomenon that occurs when threads that execute in lock-step are forced to take different program paths due to branches in the code. In face of divergences, some threads will have to wait, idly, while their diverging siblings execute. Optimizing the code to avoid divergences is difficult, because this task demands a deep understanding of programs that might be large and convoluted. In order to facilitate the detection of divergences, this paper introduces the divergence map, a data structure that indicates the location and the volume of divergences in a program. We build this map via dynamic profiling techniques, which we have implemented on top of an open source CUDA compiler. To illustrate the importance of the divergence map, we have used it to pin-point the core regions that must be optimized in well known public applications. By hand optimizing some applications, we have added 9-11% speedups onto kernels that have already gone through the sieve of many programmers.

1 Introduction

Increasing programmability and low hardware cost are boosting the use of graphical processing units (GPU) to run general purpose applications. Illustrative examples of this new trend are the rising popularity of CUDA\textsuperscript{1}, Stream SDK\textsuperscript{2} and OpenCL\textsuperscript{3}. Running general purpose programs in GPUs is attractive because these processors are massively parallel. For instance, the Nvidia’s GeForce GTX 285 series has 240 processing units and 30,720 hardware threads. Such a hardware has allowed GPU-based applications to run over 400x faster than equivalent CPU based programs [1]. This trend is likely to continue, as upcoming hardware more closely integrates GPUs and CPUs [2], and new models of heterogeneous hardware are introduced [3].

GPUs are highly parallel; however, due to its restrictive programming model, not every application can benefit from all their processing power. In these processors, threads are organized in groups that execute in lock-step. Such groups are called warps in the Nvidia jargon\textsuperscript{1}, or wavefronts in ATI’s\textsuperscript{2}. To better understand the rules that govern threads in the same warp, we can imagine that each warp might use a number of processing units, but has only one instruction fetcher. As an example, the GeForce GTX 285 GPU is able to run 30 warps at the same time; each warp consists of 32 threads, and uses 8 processing units. Thus, each warp might perform 32 instances of the same instruction in four cycles of the hardware pipeline. Regular applications, such as scalar vector multiplication, fare very well in GPUs, as we have the same operation being independently performed on different chunks of data. However, not every application is so regular, and divergences may happen.

Divergences happen when threads inside the same warp follow different paths after processing the same branch. The branching condition might be true to some threads, and false to others. Given that each

\textsuperscript{1}See The CUDA Programming Guide, 1.1.1
\textsuperscript{2}See ATI CTM Guide
\textsuperscript{3}See The OpenCL Specification, 1.0
warp has access to only one instruction at each time, in face of a divergence, some threads will have to wait, idly, while others execute. Hence, divergences may be a major source of performance degradation. As an example, Baghsorkhi et al. [4] have analytically found that approximately one third of the execution time of the prefix scan benchmark [5], included in the CUDA software development kit (SDK), is lost due to divergences. Optimizing an application to avoid divergences is problematic for a number of reasons. First, some parallel algorithms are inherently divergent; thus, threads will naturally disagree on the outcome of branches. Second, finding highly divergent branches burdens the application developer with a tedious task, which requires a deep understanding of code that might be large and convoluted.

In this paper we introduce the Divergence Map, a data structure that indicates the location and the volume of divergences. We build the divergence map via dynamic profiling of the GPU program, or kernel, as it is normally called. The divergence map indicates how many warps have visited each kernel branch, and how many divergences have happened per branch. This data structure has two main purposes. First, it shows to the application developer the program points that cause performance degradation due to diverging execution paths. Second, the divergence map provides useful feedback to self-adjusting compiler optimizations, which rely on profiling information to decide the best ways to improve a program.

We have implemented a dynamic profiler as a patch to the Ocelot open source [6] CUDA compiler. We have produced divergence maps to many benchmarks freely available in the Internet, and we have used this information to manually optimize two of these applications. As an example, by changing only 2.3% of the code of a well-known implementation of parallel quicksort [7] we got speedups of up to 9.2%. This number may seem small, but this gain applies onto a highly optimized application, that has gone through three years of public scrutiny. All the code and an extensive description of these experiments are available in our webpage 4. Although we have implemented our tool on a CUDA compiler, the techniques introduced in this paper fit any architecture that follows the single-instruction multiple-data (SIMD) execution model that characterizes GPU’s warps. Our profiler also works with predicated instructions, which, like ordinary branches, also suffer from divergences.

2 Related Work

Profilers are old allies of compiler designers. Since Graham et al.’s highly influential gprof [8], many profiling tools and techniques have been described. Profiling has been initially used to measure the dynamic behavior of sequential applications [8, 9, 10], but subsequent works have extended profiling into the realm of parallel computers with great success [11, 12, 13, 14, 15]. Parallel profilers, such as the recent works of Tallent et al. [14] or Eyerman and Eeckhout [11] generally focus on systems where threads are able to execute independently of each other. Thus, profiling applications that fit the SIMD model, used in GPU’s warps, is still a challenge.

The development of general purpose applications on GPUs has received a substantial amount of attention recently; however, there are few works dealing directly with divergences in SIMD architectures. Fung et al. [16] determined the best program regions to re-converge divergent threads, and the hardware necessary to perform this task. Kerr et al. [6] have evaluated, via an emulator, the impact of divergences on the performance of CUDA applications. Our objectives are similar to Kerr et al.’s, but we do profiling instead of simulation. Simulation has the advantage of using a controlled environment to obtain data; however, it has problems of its own, i.e simulation accuracy. Baghsorkhi et al. [4] have described an analytical model that estimates the cost of divergences; however, they must assume that a given branch is divergent, a fact that we check via profiling.

We know two GPU profilers. On the academic side, Boyer et al. [17] have described a profiler that supports debugging; however, it does not handle divergences in any special way. On the industrial side, Nvidia has released the CUDA Visual Profiler, or CVP 5. CVP lets the CUDA developer to probe several aspects of the kernel behavior, including the existence of divergent threads in the parallel program. Regarding the measurement of divergencies, our tool is more precise than CVP. CVP tells the developer how many divergences have occurred during the kernel execution, probably, as we speculate, by reading some counter implemented in hardware. However, it does not point the place or the volume in which the divergences happened – exactly the information that we provide, and that we believe is the most useful to guide programmers in optimizing their applications.

4http://www2.dcc.ufmg.br/laboratorios/llp/wiki/doku.php?id=coutinho
3 Divergences

We will illustrate the concept of divergence showing how this phenomenon occurs in an implementation of the bitonic sorting algorithm [18]. The code, which we took from Cederman et al. [7], is given in Figure 1; we will focus on the boxed part. We have labeled three interesting conditional branches with the entries in the divergence map that we have obtained via the precise profiling strategy described in Section 5. The divergence map is a function that maps branches to integer pairs. The first integer denotes the number of divergences, and the second denotes the number of times the block was visited by a warp.

Figure 2 shows the simplified PTX representation of this kernel. PTX is the high-level instruction set used to represent CUDA kernels. We have augmented this control flow graph with numbers denoting the addresses of each instruction in the kernel code. Additionally, we have added to the figure the divergence map entries after the three conditionals that we have outlined in Figure 1.

The \( O(n \ln^2 n) \) complexity penalizes the bitonic sort method when we are restricted to a sequential hardware; nevertheless, the ability to perform \( n/2 \) comparisons, where \( n \) is the array size, in parallel, makes this algorithm attractive to the GPU execution environment. However, we must remember that we are tied to our SIMD execution model. Bringing back the metaphor of Section 1: we have four functional units in this example, but only one instruction fetcher. Invariably the condition \((\text{tid} & k) == 0\) will have a divergence. In face of divergences, some threads will not process any instruction. We indicate this fact by giving these threads the symbol \( \bullet \) in the idle cycles. We assume that \( \text{values} \), the input array, is \( \{4, 3, 2, 1\} \), and that the id of thread \( t_n \) is \( n, 0 \leq n \leq 3 \). When \( k = 2 \) and \( j = 1 \), the input array causes two divergences. The first split happens at cycle \( i = 3 \), due to the branch \( \text{bra } t1 \) \( \text{bra } t2 \), and it separates \( t_0 \) and \( t_2 \) from \( t_1 \) and \( t_3 \). This divergence happens because the condition \( \text{ixj} > \text{tid} \) is true only for \( t_0 \) and \( t_2 \). The second split happens at cycle \( i = 6 \), due to the branch \( \text{bra } t3 \), and it separates threads \( t_0 \) and \( t_2 \). After the first iteration of the outer loop in Fig-

---

6http://www.cs.chalmers.se/~dcs/gpuqsortdcs.html

7See PTX: Parallel Thread Execution, ISA v.1.3
<table>
<thead>
<tr>
<th>cycle</th>
<th>opcode</th>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>xor</td>
<td>$t_1$</td>
<td>$t_1$</td>
<td>$t_1$</td>
<td>$t_1$</td>
</tr>
<tr>
<td>2</td>
<td>gt</td>
<td>$t_2$</td>
<td>$t_2$</td>
<td>$t_2$</td>
<td>$t_2$</td>
</tr>
<tr>
<td>3</td>
<td>bra</td>
<td>$t_3$</td>
<td>$t_3$</td>
<td>$t_3$</td>
<td>$t_3$</td>
</tr>
<tr>
<td>4</td>
<td>and</td>
<td>$t_4$</td>
<td>$t_4$</td>
<td>$t_4$</td>
<td>$t_4$</td>
</tr>
<tr>
<td>5</td>
<td>eq</td>
<td>$t_5$</td>
<td>$t_5$</td>
<td>$t_5$</td>
<td>$t_5$</td>
</tr>
<tr>
<td>6</td>
<td>bra</td>
<td>$t_6$</td>
<td>$t_6$</td>
<td>$t_6$</td>
<td>$t_6$</td>
</tr>
<tr>
<td>7</td>
<td>load</td>
<td>$t_{14}$</td>
<td>$t_{14}$</td>
<td>$t_{14}$</td>
<td>$t_{14}$</td>
</tr>
<tr>
<td>8</td>
<td>load</td>
<td>$t_{15}$</td>
<td>$t_{15}$</td>
<td>$t_{15}$</td>
<td>$t_{15}$</td>
</tr>
<tr>
<td>9</td>
<td>gt</td>
<td>$t_{16}$</td>
<td>$t_{16}$</td>
<td>$t_{16}$</td>
<td>$t_{16}$</td>
</tr>
<tr>
<td>10</td>
<td>bra</td>
<td>$t_{17}$</td>
<td>$t_{17}$</td>
<td>$t_{17}$</td>
<td>$t_{17}$</td>
</tr>
<tr>
<td>11</td>
<td>load</td>
<td>$t_{18}$</td>
<td>$t_{18}$</td>
<td>$t_{18}$</td>
<td>$t_{18}$</td>
</tr>
<tr>
<td>12</td>
<td>load</td>
<td>$t_{19}$</td>
<td>$t_{19}$</td>
<td>$t_{19}$</td>
<td>$t_{19}$</td>
</tr>
<tr>
<td>13</td>
<td>if</td>
<td>$t_{20}$</td>
<td>$t_{20}$</td>
<td>$t_{20}$</td>
<td>$t_{20}$</td>
</tr>
<tr>
<td>14</td>
<td>bra</td>
<td>$t_{21}$</td>
<td>$t_{21}$</td>
<td>$t_{21}$</td>
<td>$t_{21}$</td>
</tr>
<tr>
<td>15</td>
<td>store</td>
<td>$t_{22}$</td>
<td>$t_{22}$</td>
<td>$t_{22}$</td>
<td>$t_{22}$</td>
</tr>
<tr>
<td>16</td>
<td>store</td>
<td>$t_{23}$</td>
<td>$t_{23}$</td>
<td>$t_{23}$</td>
<td>$t_{23}$</td>
</tr>
<tr>
<td>17</td>
<td>store</td>
<td>$t_{24}$</td>
<td>$t_{24}$</td>
<td>$t_{24}$</td>
<td>$t_{24}$</td>
</tr>
<tr>
<td>18</td>
<td>store</td>
<td>$t_{25}$</td>
<td>$t_{25}$</td>
<td>$t_{25}$</td>
<td>$t_{25}$</td>
</tr>
<tr>
<td>19</td>
<td>sync</td>
<td>$t_{44}$</td>
<td>$t_{44}$</td>
<td>$t_{44}$</td>
<td>$t_{44}$</td>
</tr>
</tbody>
</table>

Figure 3: A snapshot of the execution trace over of the program in Figure 2. Each thread trace is given as an ordered sequence $[t_1, \ldots, t_n]$, where each $t_i$ is an instruction in the kernel, or a • indicating that the thread did not execute in that cycle.

In order to improve the kernel, we start by noticing that the code trace formed by block $L_3$ followed by block $L_5$ is very similar to the trace $L_4 + L_6$. The only divergence comes from the conditionals in program points 9 and 13. Armed with this observation, we bind the blocks $L_5$ and $L_6$ together, using the index manipulation trick that gives us the program in Figure 4 (a). Divergences might still happen; however, whereas the maximum divergent path in Figure 2 contains eight instructions, the worst divergence case in Figure 4 (b) contains six instructions. This optimization gives a speed-up of 6.75%.

The code in Figure 4 (b) still provides us with optimization opportunities. We can use the ternary selectors available in CUDA to merge basic blocks $L_3$ and $L_4$, thus removing the divergence at the end of block $L_2$. The modified source is shown in Figure 4 (c), and the equivalent PTX code is given in Figure 4 (d). An instruction such as $%a = \text{sel} %xj %p$ assigns $%xj$ to $%a$ if $%p$ is not zero. It assigns $%xj$ to $%a$, otherwise. In the new program, the worst case divergent path has only two instructions. This final optimization gives us a speed-up of 9.2%.

**GPUs ask for new optimization techniques** We have observed that the optimization techniques that improve the efficiency of CUDA kernels might not apply to sequential programs. For instance, the kernel in Figure 4 (d) has a path with 11 instructions, which is executed every time the branching condition evaluates to true. On the other hand, the equivalent code sequence, in Figure 2 contains nine instructions. In order to measure the relative performance between all these kernels in a sequential machine, we have compiled them to x86, using gcc -O0 to preserve the original sequences of instructions, and we have run each program on a Intel x86 board, with 2G of RAM and 1.2MHz of clock. Running one iteration of each binary on a four million elements array, we observe that the original, non-optimized PTX code from Figure 2 is 19% faster than the program in Figure 4 (b), and 11% faster than the program in Figure 4 (d), the opposite of what we would find in CUDA!
5 Profiling via Instrumentation

We have implemented the divergence map as two arrays of integers, that we call \( \tau \) and \( \delta \), such that \( \tau[b] \) stores the number of warps that have visited basic block \( b \), and \( \delta[b] \) stores the number of divergences that took place at \( b \). We insert the necessary instrumentation automatically in a three-phase process:

1. **Initialization**: when we reserve memory on the GPU address space to store our arrays, and initialize these arrays with zero’s.

2. **Measurement**: when we compute the number of visits and divergences, storing the results in our arrays.

3. **Reading**: when we copy to the CPU the data accumulated during the profiling of the kernel program, in the GPU.

Because the initialization and reading phases are trivial, we will only describe the measurement phase.

We detect branches via program instrumentation. Currently we insert the code in Figure 5 at each branch. This figure shows the instrumentation of block \( L_2 \) of the example in Figure 2. We split each instrumented basic block \( b \) into three new blocks: \( b_{up} \), \( b_{bottom} \), and \( b_{\text{find writer}} \). Notice that not every branch is divergent, and we are improving our profiler to avoid instrumenting non-divergent branches. The code that performs the instrumentation executes two tasks: (i) in blocks \( b_{up} \) and \( b_{\text{find writer}} \) we find a thread – the *writer* – to update the arrays \( \delta \) and \( \tau \). (ii) in block \( b_{bottom} \) we detect and report the divergence.

We let the writer to be the thread with the lowest identifier among the active threads in the warp. In Figure 5 the variable \( %\text{laneid} \) denotes the thread identifier inside the warp. We cannot simply choose as the writer the thread with \( %\text{laneid} = 0 \), because this thread may be idle due to a previous divergence. Thus, in blocks \( b_{up} \) and \( b_{\text{find writer}} \) we loop through the live threads, looking for the one with lowest \( %\text{laneid} \).

Once we have a suitable writer, we perform the detection of divergences via voting. The PTX instruction \( %p = \text{vote.uni.pred, %q} \) sets \( %p \) to true if all the threads in the warp find the same value for the predicate \( q \). Thus, we vote on the predicate that controls the outcome of a potentially divergent branch. If every warp thread agree on the predicate, no divergence happens at that particular moment of the program execution; otherwise, a divergence takes place, and it is necessary to increment the \( \delta \) array. The instruction \( @\text{AmWriter, %a}[L_2] = \text{atom.add, %a}[L_2] 1 \), in the ninth program point in Figure 5, adds one to \( \delta[L_2] \) if, and only if, the predicate \( @\text{AmWriter} \) is true.

The instrumentation requires the serialization of warp threads at branching points, because the updating of the divergence map is performed via atomic operations. Thus, the worst case complexity cost of the instrumentation, per branch, is proportional to the number of warps in the running kernel times the number of threads per warp. However, given that nested
divergences are rare, the thread with lowest identifier in the warp will be often present among the active threads. Thus, the loop in block \( L_2(\text{find writer}) \), in Figure 5, tends to find a valid writer after the first iteration. We give an empirical evaluation of the instrumentation cost in Section 6.

6 Experimental Results

We have evaluated experimentally 11 publicly available CUDA benchmarks, instrumenting a total of 25 kernels, and finding divergences in 19. In order to guarantee reproducibility, we have also made our compiler, and all the scripts used in these experiments available in our webpage. We run these experiments on a Nvidia GeForce GTX 260 powered video card.

The Benchmarks: These experiments use the following benchmarks, which, due to space constraints, we will refer by two letters only: Quicksort (qs), is the program that gave us the bitonic sort kernel used throughout the paper. Scan of Large Arrays (sc), is NVIDIA’s implementation of parallel prefix sum \(^8\). Cudaseg (cs) does segmentation of biomedical images \(^9\). The rest of our benchmarks comes from the Rodinia Benchmark Suite \([19]\): Back Propagation (bp), BFS (bf), CFD (cf), HeartWall (hw), Hotspot (hp), Needleman-Wunsch (nw), SRAD (sr) and Stream Cluster (st). In this section we will only show results for kernels that had at least one divergent branch; six kernels in our benchmark suite did not show divergences, and we omit them. All these benchmarks are actual applications, publicly available in their author’s webpages, that use kernels with meaningful sizes – Table 2 shows the size of each kernel. Because the benchmarks might contain more than one kernel, we discriminate them with the first and last letter of their names.

Divergences in general purpose applications: Table 1 shows the number of divergences in our benchmarks. We see that a large proportion of the conditional branches present in typical CUDA applications – between 50% and 60% – give origin to divergences. The figure also isolates the program points with the largest number of divergences per kernel. From these figures we see that some program points suffer from a non-negligible number of divergences. Interesting is the fact that these benchmarks have gone through the eye of many developers, and are extremely optimized; yet, divergences still are very common. Table 1 shows that the most visited branches tend to be the most divergent, although there are exceptions, like Stream Cluster. Therefore, traditional profiling techniques, such as those employed on gprof \([8]\), can easily pin-point the most executed program regions, but these might not necessarily be the biggest sources of divergences.

Probing the instrumentation overhead: Table 2 gives some figures about the overhead imposed by our instrumentation. We see that our instrumentation approach adds a substantial overhead to the target kernel. In terms of code size, the instrumented program tends to grow between 2 and 3 times. The overhead is even bigger in terms of time, multiplying the execution time of applications by a factor of up to 1,500 in the parallel prefix scan (sc, pn). Nevertheless, these numbers are similar to other instrumentation based profiling strategies \([20]\). More important: our instrumentation does not change the semantics of block \( L_2(\text{find writer}) \).

\(^8\)http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html
\(^9\)http://code.google.com/p/cudaseg/
Most divergences occur in few branches: We found an interesting property in the analyzed benchmarks: divergences are very concentrated. Table 3 illustrates this pattern inherent to all benchmarks. We notice that nine kernels have more than 90% of the divergences in one branch, and only one kernel – cudaseg – contains more than six highly divergent branches. This pattern of concentration seems to point that the developer should focus on a few program regions in order to optimize the effect of divergences on SIMD applications.

Optimizing SRAD: In addition to bitonic sort (Section 4), we have used the divergence map to optimize Rodinia’s SRAD. We got 11.5% of time speedup by merging the contents of different branches. We have reported these gains to the authors of Rodinia, and have been told that our optimized kernel will be part of the next edition of this benchmark suite. We speculate that it is possible to optimize the other applications too, but, as we are not familiar with the source code, we have not tried it.

7 Conclusion

This paper has introduced the divergence map, a concept that helps application developers to implement more efficient programs on single-program-multiple-data execution environments, such as the CUDA architecture. The divergence map highlights the program regions that cause thread divergences, and thus degrade performance. We build divergence maps via divergence maps via the divergence map over SIMD applications.
dynamic profiling. Our experiments have shown that the divergence map is an important tool, as traditional profilers would not be able to correctly point the program locations that account for the majority of thread divergences. Additionally, we have showed how the divergence map has helped us to manually optimize well-known CUDA applications. All the software described in this paper, in particular our version of the Ocelot compiler that contains our profiler, is publicly available \(^\text{10}\).

References


\(^{10}\)http://www2.dcc.ufmg.br/laboratorios/llp/wiki/doku.php?id=coutinho