Heuristics for Conversion Process of GPU's Kernels for Multiples Kernels with Concurrent Optimization Divergence

José Ritomar Carneiro Torquato[1], **Esteban Walter Gonzalez Clua**[1]
[1] Institute of Computing, Federal Fluminense University, Niteroi, RJ, Brazil

Abstract - *Graphics Processing Units have been created with the objective of accelerating the construction and processing of graphic images. In its historical evolution line, concerned with the large computational capacity inherent, these devices started to be used for general purposes. However, the design of the GPUs don´t work well with divergent algorithms, mainly conditionals and repetitions. In this work we present a strategy for finding the divergence root of the kernels and try to deduce alternative solutions, decomposing them into concurrent kernels. We developed mechanisms for the user in order to easily readapt his code and take advantages of architectures that support concurrent kernels.*

Keywords: Divergence; Concurrents Kernels; Warps; GPGPU.

1 Introduction

GPUs (Graphics Processing Units) were designed to make to process polygons, and they have a peculiar feature: the same sequence of operations to different data. Following its historical evolution, current GPUs keep following this paradigm in its architectural models. In this style of execution, all the hardware involved executes the same instruction, before moving on to the next one. In fact the model brings benefits by reducing the cost of production and offering an optimized memory access. The new architecture Kepler GK110, is called by the NVidia "The next generation of GPUs" and still uses the same concepts of multiprocessor streams [1]. We believe that this architecture remains in awhile because, in practice, this restriction is what makes technologically feasible to massively parallel architecture.

Diverging code is defined as the fact that a stream of code executed in a parallel environment can take different directions in each of its instances. In Single Instruction Multiple Thread (SIMT) architecture, occurring divergence, all statements that do not follow the same path are forced to wait at the point of divergence. It is noteworthy that this is not a limitation of the solution, but the hardware architecture.

In this work we identify strategies that can minimize the effects of divergence in execution time of parallel applications. The optimization algorithm is currently the main and most efficient way to reduce the impact of divergence, forcing the implementation to follow a single path. A commonly adopted technique consists in separating the code into two parts, running a first leg and then the other. This was the only way to deal with this problem on GPUs until a little time ago. Although it is shown effective, in many cases the time dependence of data makes this solution inappropriate. With the Fermi GPUs series, Nvidia started implementing concurrent kernels. We present a new technique to divide a code divergence by using this technology. Preliminary tests showed that we can reduce the divergence by creating concurrent kernels.

In this paper we identify the mechanisms used to reduce the impact of the difference in execution time of parallel algorithms. Furthermore, we propose the use of concurrent kernels based on new generations of GPUs, such as Kepler, as an alternative in treating the problem.

The remainder of the paper is organized as follows. Section 2 provides background on GPU's evolution and Unified Architecture. Section 3 describes the divergence problem. Section 4 presents the optimizations of divergence, evaluation methodology and results. Section 5 discusses related work, and gives directions for future work.

2 Unified architecture

The first video cards created were simple and the severe hardware limitations made unimaginable graphics processing by them. Following the chronological evolution emerged raster, fixed function and programmable devices. These last one brought pixels and vertex processors, able to treat, only and respectively, pixels and vertices. At that time, there were not multi-core CPUs so the GPU was seen as an alternative to increase the processing power in specific tasks. Thus, researchers from different areas began to "consider" data input of mathematical calculations as vertices and pixels, making the use of these processors in solving mathematical equations possible. For the first time a GPU was used with general-purpose, giving rise to GPGPU (General Purpose GPU)

Processors of vertices and pixels did nothing beyond their specific tasks, increasing the interest in the computational power of devices, as well as the inconvenience of having to map all that was wanted in vertices and pixels. In addition, the processors were built only to the treat their structures, and an application that performed more vertices or pixel would leave the other processors idle.

Nvidia proposed a unified architecture in their cores [2], creating a new architecture called CUDA (Compute Unified Device Architecture). Some advantages over previous architectures CUDA GPUs are:

- Memory random access: access to any region of memory to read and write;
- Manageable user- Cache: threads can cooperate reading and writing data in shared memory and any thread can access the shared memory of its block ;
- Low learning curve: simple extensions of C language, without requiring knowledge of graphics or graphics APIs.

Programming models for GPU (as CUDA and OpenCL) are designed to allow legacy programs to take advantage of new features in a transparent way. In other words, programs originally written for a particular architecture are scalable to the following architecture. Also, allow the use of heterogeneous systems, thus CPUs and GPUs are distinct and separate memory devices. Each of them performs the function for which they are best prepared.

CUDA facilitates programming since it allows developers to focus on developing their algorithms without the need to learn language specific mechanisms. Instead, it provides a minimum length of the C / C + + to construct parallel applications.

3 Divergence problem

During the execution of the code by the GPU, each decoded instruction is sent to the scheduler. They remain queued until despatch in execution units, often called warps. This approach reduces the time for loading and decoding of instructions by N execution units, however, it does not require instructions to follow the same path. If there would be a piece of code in which some instruction keep on processing, they execute while the others wait for a different point of divergence [3][4]. Thus, a conditional statement can result in divergence when it is based on values that are particular to the specific thread [5].

For example, one if instruction may cause the thread to follow different paths, or, similarly, a loop may cause divergence whether the conditions are based on the thread's own values.

To demonstrate the impact of the divergence, we must consider the following code, similar to what occurs in problems of reducing vectors:

```
01 if (threadIdx.x < 32)
02 {
03     if (threadIdx.x < 16)
04     {
05         if (threadIdx.x < 8)
06             func_a1();
07         else
08             funca2;
09     }
10     else
11     {
12         func_b();
13     }
14 }
```

Listing 1: Divergence problem demonstration.

We will use the code in Listing 1 to illustrate how the divergence can affect the efficiency. Its execution results in data that are displayed in Figure 1.

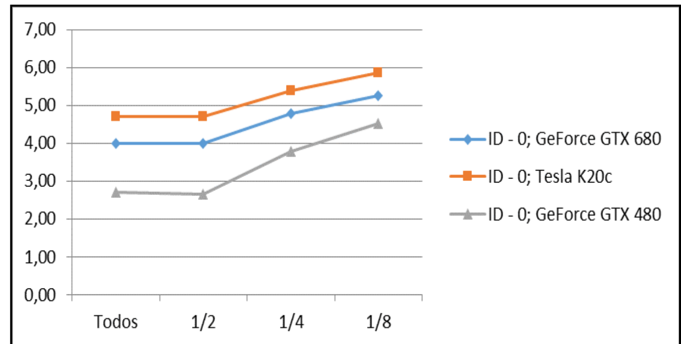

Figure 1: Sample of how the divergence may have strong impact on performance

The first line of code in Listing 1 eliminates all threads of the block except the first 32 threads (first warp), the one we will use for our analysis. This does not result in any difference within a specific warp. The other warps of the block simply do not scale to this session and wait.

Analyzing only the first warp, we observed that in line 3 the test `threadIdx.x < 16` is done, what breaks the warp is carried out exactly in half. In the graph first transition is noticed, this operation does not result in actual divergence since the CUDA kernels are organized in banks of 16 cores, not 32. Thus, the scheduler cyclely sends instructions to two or more sets of 16 cores and the paths of true and false conditional statement run on cores from different banks.

In the subsequent step, the threads 16 to 31 call `func_b` function (line 12), however, threads 0 to 15 have another condition associated (line 05). Therefore, this time is not based on half of the warp, but in a quarter of it. So, we need a minimum of 16 threads for scheduling. Thus, the first eight

threads will proceed to the function `func_a1` while the remaining eight (8.. 15) await.

The functions `func_b` and `func_a1` will continue their instructions independently and shoot the second half of the warps. This is less efficient than the search for a single statement, but nevertheless, better than a sequential execution. Eventually `func_a1` will finish and `func_a2` will start the threads 0-7. Meanwhile, `func_b` might also have been completed.

Analyzing the best result different levels of divergence are perceived. The first one is great, without divergence. The second one differs based on half of the warp but does not result in real divergence, since they run in parallel. Dividing the first half of warps into two groups, these should run in series, as they will expect a stretch to be finished and only then the next starts. Once again, dividing the first group in a total of four paths they will also result in a serial execution case.

4 Optimization of divergence

4.1 *Naïve* Test

An example of simple demonstration was created in order to highlight the importance of separating different kernels and create separate concurrent Kernels. Considering that the program will receive, as input, a vector of k positions filled with N numbers, which alternate between large and small values, as shown in Figure 2 below:

i =	*0*	*1*	*2*	*3*	*4*	*5*	*...*	*k-4*	*k-3*	*k-2*	*k-1*	*k*
N =	5	5000	5	5000	5	5000	5	5000	5	5000	5	5000

Figure 2: Input of the First Demonstration Kernel.

In the next step, our test program will run on a kernel, shown in Listing 2, a repetition by N times (with N being the value of the position i of the input vector) and the input vector is stored in global memory.

In our first test, we have a *Naïve* approach, which reads data sequentially. We will have half the cores using a small value and the other half using a large value (in the same block) and it is hoped that the cores running the repetition with the highest number of iterations dictate the overall runtime.

Next, we used an index thread strategy, forcing a block to take the odd and another the even numbers. Our objective is to allow two kernels to perform the same function concurrently. Thus, we come to the result shown in Listing 3.

The kernels shown in listing 2 and 3 are equal in function, however, we put some "intelligence" in the while loop within lines 5 and 15 in Listing 3 in order to force these kernels specifically deal with values from the same class (all small or all large). Thus, the `kernel02a` will only treat small values of Figure 2 while the `kernel02b` treats the others.

```
01  __global__ void kernel01(int *a)
02  {
03      int i = a[threadIdx.x];
04      __shared__ int k;
05      while (i > 0){
06          i--;
07          k+=i;
08      }
09  }
```

Listing 2: Initial kernel.

```
01  __global__ void kernel02a(int *a)
02  {
03      int i = a[threadIdx.x];
04      __shared__ int k;
05      while (((i % 2 == 0) && i > 0)){
06          i--;
07          k+=i;
08      }
09  }
10
11  __global__ void kernel02b(int *a)
12  {
13      int i = a[threadIdx.x];
14      __shared__ int k;
15      while ((i % 2 != 0) && (i > 0)){
16          i--;
17          k+=i;
18      }
19  }
```

Listing 3: Concurrents Kernels.

The Table 1 summarizes the execution times, and Figure 3 shows these results graphically comparing them:

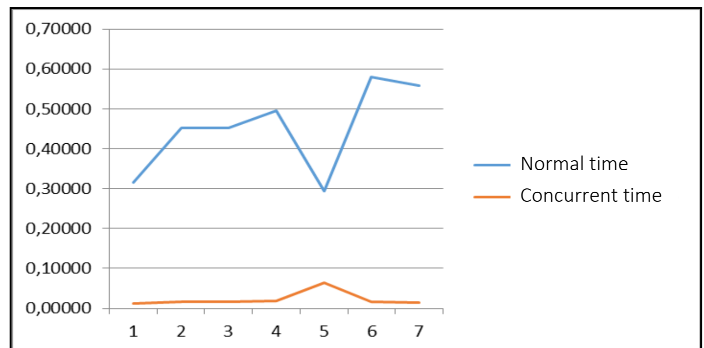

Figura 3: Initial x Concurrents Kernels

Host	Device	Normal Time	Concurrent Time
K10 Motorhead	GeForce GTX 680	0,31466	0,01200
K10 Motorhead	Tesla K10.G1.8GB	0,45114	0,01722
K10 Motorhead	Tesla K10.G1.8GB	0,45142	0,01702
K20 - Clash	Tesla K20c	0,49501	0,01869
K20 - Clash	GeForce GTX 680	0,29312	0,06470
Orange Lab Pos	GeForce GTX 480	0,57942	0,01533
Orange Lab Pos	GeForce GTX 480	0,55824	0,01523

Table 1: Comparison of execution times in the first kernel demonstration

4.2 *Sum Reduction*

A reduction algorithm extracts a single value from a matrix, calculated by comparing every element of it. The reduction may be to sum, to the maximum or minimum values, of the components. These algorithms share the same structure. A reduction may be performed sequentially stepping through each element of the array. When an element is visited, the action to be taken depends on the desired reduction. To sum reduction, the current value is accumulated [1].

Listing 4 shows a CUDA kernel for reduction of sum. The input matrix data were placed in main memory, the array was divided so that each block CUDA reduce a portion of the original matrix. The reduction will be made in device, using the shared memory, in other words, there will be a shared variant where the partial sums will be saved. Each iteration of the line 6 loop is a round of reduction. The syncthreads () statement in the for loop ensures the necessary timing for the performace of the previous iteration and to prepare the threads for the next iteration. Each round of implementation of even elements will contain the partial sums of each pair after iteration until all sums are performed.

The kernel of Listing 4 has caused divergence of the iteration loop of line 6. In this place only threads with even threadIdx.x values perform the sum due to the condition imposed on line 9. Such divergence can be reduced with a change in the algorithm.

```
01  __global__ void sumReduceD(const Utype *a, Utype *sum)
02  {
03      __shared__ int partialSum[arraySize];
04      unsigned int t = threadIdx.x;
05      partialSum[t] = a[t];
06      for(int stride = 1; stride < blockDim.x; stride *= 2)
07      {
08          __syncthreads();
09          if(t % (2*stride) == 0)
10          {
11              partialSum[t] += partialSum[t+stride];
12              sum[0] = partialSum[t];
17          }
18      }
19  }
```

Listing 4: Divergent Reduction Sum

```
01  __global__ void sumReduceN(const Utype *a, Utype *sum)
02  {
03    __shared__ int partialSum[arraySize];
04    unsigned int t = threadIdx.x;
05    partialSum[t] = a[t];
06    for(int stride = blockDim.x >> 1; stride > 0; stride >>= 1)
07    {
08      __syncthreads();
09      if(t < stride)
10      {
11        partialSum[t] += partialSum[t+stride];
12        sum[0] = partialSum[t];
17      }
18    }
19  }
```

Listing 5: Optimized Sum Reduction.

The modified kernel in Listing 5 adds elements that are in the middle of a section, rather than adding neighboring elements. At the end of the first iteration, the sum is stored in the first half of the array. At each iteration of the loop the overall operation is divided by 2 by shifting step by one bit to the right, an economical way to perform division by 2. Note that the kernel in Listing 5 also has an `IF` (line 9) which means that it will still have divergence, however, the amount of threads that execute this instruction is minimal compared to the previous case.

To verify the efficiency of concurrence we have unbundled the kernel of Listing 5 in two others, each of them responsible for performing half the reduction of sum.

```
01  __global__ void sumReduceC1(const Utype *a, Utype *sum, long offset)
02  {
03      if (threadIdx.x < offset)
04      {
05          __shared__ int partialSum[arraySize];
06          unsigned int t = threadIdx.x;
07          partialSum[t] = a[t];
08          for(int stride = blockDim.x>>1; stride > 0; stride >>= 1)
09          {
10              __syncthreads();
11              if(t < stride)
12              {
13                  partialSum[t] += partialSum[t+stride];
14                  sum[0] = partialSum[t];
15              }
16          }
17      }
18  }
19
20  __global__ void sumReduceC2(const Utype *a, Utype *sum, long offset)
21  {
22      if (threadIdx.x >= offset)
23      {
24          __shared__ int partialSum[arraySize];
25          unsigned int t = threadIdx.x;
26          partialSum[t] = a[t];
27          for(int stride = blockDim.x>>1; stride > 0; stride >>= 1)
28          {
29              __syncthreads();
30              if(t < stride)
31              {
32                  partialSum[t] += partialSum[t+stride];
33                  sum[0] = partialSum[t];
34              }
35          }
36      }
37  }
```

Listing 6: Concurrent Sum Reduction

Host	Device	Normal Time	Divergent Time	Concurrent Time
K10 Motorhead	ID - 0; GeForce GTX 680	0,01210	0,01523	0,01411
K10 Motorhead	ID - 1; Tesla K10.G1.8GB	0,01834	0,02387	0,02256
K10 Motorhead	ID - 2; Tesla K10.G1.8GB	0,01846	0,02390	0,02214
K20 - Clash	ID - 0; Tesla K20c	0,02458	0,04464	0,03117
K20 - Clash	ID - 1; GeForce GTX 680	0,01168	0,01533	0,01440

Table 2: Comparison of execution times on Sum Reduction Algorithm

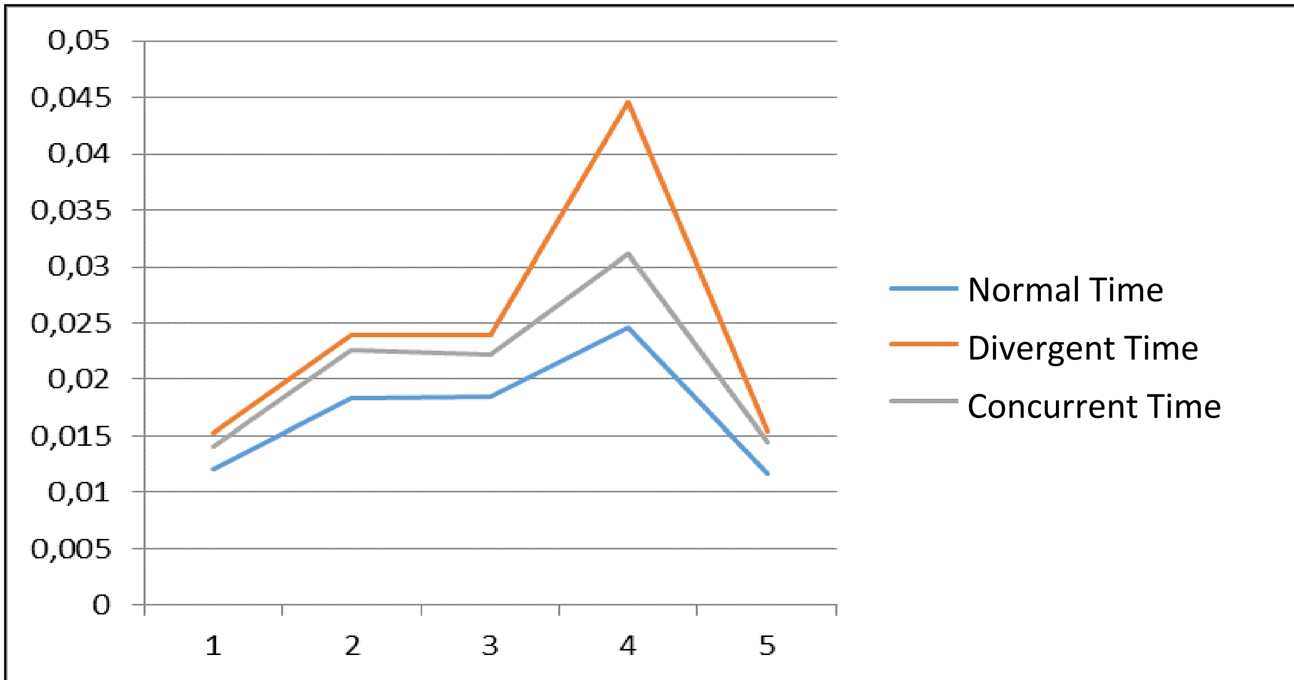

Figure 4: Graph of execution times on Sum Reduction Algorithm

The `sumReduceC1` and `sumReduceC2` kernels in Listing 6 run concurrently, each being responsible for elements of the two halves of the array, delimited by the offset variant. The Table 2 below shows the times taken in the implementation to reduce the sum of an array with 1024 elements.

5 Conclusions and Futurework

Here we present the problem of disparity in kernels, ie, the divergence that is the result of the emergence of distinct branches of implementation due to conditional or repetitions present in algorithms.

In this paper we propose a new approach to minimize the effects of them through the use of concurrent kernels and found satisfactory results that justify further study on the topic.

As future work, we propose algorithms to analyze patterns in two suites used for investigation of parallel applications. The Rodinia [6] suite is often used to measure multi / many core and parallel data applications, covering a wide range of parallel communication patterns, among them applications of medical imaging, bioinformatics, physical simulation, image processing, etc.. The Parboil [7] suite brings a suite of applications useful to study the performance of the architecture and compilers.

It is also intended to analyze the Cetus [8] which is a source code translator for multicolored infrastructure that fosters research in architecture for compiler optimizations with automatic parallelization.

Thus, we seek to list a series of strategies to map different types of problems, enabling transform a single kernel into n concurrent kernels. We hope to contribute to a set of heuristics that migth assist in mapping, preferably in an automatic way and with less divergence as possible.

6 References

[1] D. B. Kirk and W. W. Hwu, Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, 2010.

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A unified graphics and computing architecture,” Micro, IEEE, vol. 28, no. 2, pp. 39–55, 2008.

[3] S. Cook, CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs. Newnes, 2012.

[4] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr., “Divergence Analysis and Optimizations,” in Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 320–329.

[5] T. D. Han and T. S. Abdelrahman, “Reducing branch divergence in GPU programs,” in Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, 2011, pp. 3:1–3:8.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for

heterogeneous computing," in Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, 2009, pp. 44–54.

[7] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Cent. Reliab. High-Performance Comput., 2012.

[8] H. Bae, D. Mustafa, J.-W. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S. Midkiff, "The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation," Int. J. Parallel Program., vol. 41, no. 6, pp. 753–767, 2013.