International Journal of scientific research and management (IJSRM) ||Volume 2 ||Issue 11||Pages|| 1736-1739 ||Nov 2014|| Website: www.ijsrm.in ISSN (e): 2321-3418

Divergence Reduction In GPUs With SIMT Architecture

***Anju Soosan Baby*[1], *Prof. Balachandran.K*[2]**

[1] Dept. of Computer Science and Engineering,
Christ University Faculty of Engineering
Bangalore, India
anju.baby@mtech.christuniversity.in

[2] Dept. of Computer Science and Engineering,
Christ University Faculty of Engineering
Bangalore, India
balachandran.k@christuniversity.in

Abstract: ***Branch Divergence has a significant impact on the performance of GPU programs. I used three Novel Software based divergence reduction techniques namely Fixed Scheduling, Frequency scheduling, and Balanced scheduling that aims to reduce branch divergence and comparing the execution time of each schedules. The calculation of End Semester Examination marks of Engineering and MBA students is the application to which the scheduling techniques are applied. The divergence condition checked whether the student belongs to Engineering Section or MBA section and based on that calculations further processing have been carried out. Evaluation shows frequency scheduling and balanced scheduling exhibiting large improvement in execution time.***

Keywords: Branch divergence, Optimization, Scheduling, SIMD, SIMT

1. Introduction

Parallel Computing has massively become a Commodity Technology. The main motivation behind massive Parallel Programming is for applications to enjoy a continued increase in speed. Parallel Programming targets the computationally intensive code, it can be divided into small parts and then solve concurrently. The fundamental design philosophies of CPU and GPU are entirely different and this results in a performance gap – CPUs are low latency low throughput Processors handling task parallel problems optimized for sequential code execution whereas GPUs are high latency high throughput Processors solving data parallel problems.

For Parallel Computations SIMD (Single Instruction Multiple Data) is the suitable execution model containing multiple processing elements receive same instruction but operate on different data streams and supervised by the same control unit [1]. GPU adopt SIMT (Single Instruction Multiple Thread) architecture which extends SIMD. SIMD use one thread with wide execution path, whereas SIMT split identical independent work over multiple lockstep threads which means run same set of operations at the same time in parallel [2]. Even though GPU offers high speed, limited programmability is one of the main drawbacks. Programma-
bility restricted in number of ways which include difficulty in handling divergence such as loops and conditional clauses. When a divergence occurs entire batch of threads will execute both sides of branch sequentially, even though each thread executes only one of the paths, So that divergence kills the performance. In this study analysed and compared the execution time of loop optimization techniques for divergence reduction on GPUs which is likely to increase SIMD efficiency To carry out this task an application of data processing of students marks of two different streams has been chosen and the performance evaluation based on the execution time for each scheduling has been studied.

2. Related Works

Since divergence degrades the performance of GPU, there are several attempts to solve this problem which include architectural proposals and software proposals. Majority of the architectural proposals aiming to increase the number of control units, effectively shifting SIMD to MIMD (Multiple Instruction Multiple Data). Some of the efforts are:

- Exploit control flow locality among threads by extending the sharing of resources in a blocks of warps. A common block-wide stack is shared by warps within a block for divergence handling. At a divergent branch, threads are compacted into new warps in hardware [3].
- To improve processor utilization for global rendering algorithms, introduce an SIMT architecture that allows for threads to be created dynamically at runtime. Large application kernels are broken down into smaller code blocks called µ-kernels so that dynamically created threads can execute. These runtime µ- kernels allow for the removal of branching statements that would cause divergence within a thread group, and result in new threads being created and grouped with threads beginning the execution of the same µ-kernel [4].
- Compaction-Adequacy Predictor (CAPRI). CAPRI dynamically identifies the compaction-effectiveness of a branch and only stalls threads that are predicted to benefit from compaction [5].
- Iteration delaying: Executing loop iterations that take the same branch direction and delaying those that take the other direction until later iterations [6].

- Branch distribution: reduces the length of divergent code by factoring out structurily similar code from the branch paths [6].

3. Background

3.1 GPU

In the initial stages GPU has evolved to rendering graphics only. As technology advanced number of cores associated with GPU exploited thereby computational capability also increased. GPU is a massively Parallel Architecture, multithreaded (fine and lightweight threads) with thousands of threads; hundreds of cores and have tremendous computational horsepower and very high memory bandwidth.

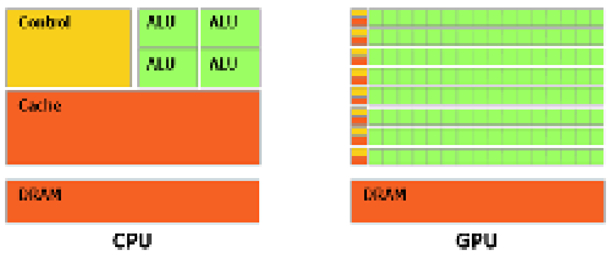

Figure 1: Design philosophies of CPU and GPU

3.2 CUDA

CUDA stands for Compute Unified Device Architecture developed by NVIDIA Corporation. A CUDA program consists of host code and device code: the phases that show diminutive or no parallelism are carved in host part and that have rich amount of parallelism are included in the device part.

3.2.1 Programming model

When a kernel is invoked, it is executed as a grid of thread blocks and it is to be noted that Parallel execution of kernel is not possible, it is always sequential i.e. one kernel after the other.

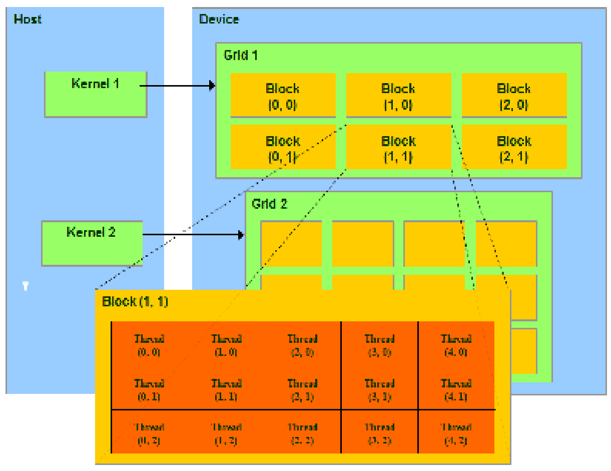

Figure 2: CUDA thread organization

At the top level, each grid comprises of thread blocks which is organized as 2-dimensional (blockIdx.x and blockIdx.y) and all the blocks must have the same number of threads structured in the same manner. Threads are 3-dimensional identified by threadIdx.x, threadIdx.y, threadIdx.z. Threads within a block co-operate each other to accomplish the tasks. Blocks are moved to the Streaming Multiprocessor(SM) for processing as soon as there are enough resources in SM to take the block, for example upto 8 blocks can be assigned to each SM in the GT200 design. Each block is again divided into 32-thread units called a warp which is the unit of thread scheduling. In a SM, number of warps will be higher than number of SPs and it is useful for latency hiding [7].

3.3 Branch Divergence

In GPU, data processing is performing in SIMT fashion i.e. all the threads in a warp execute the same instruction before moving into the next instruction. When all the threads within a warp follow the same path it works well. Suppose for an *if-then-else* construct, the execution works well when either all threads execute *if* part or all execute *else* part. When threads within a warp take different paths i.e. some of the threads take *if* part and others take *else* part, the SIMT execution style no longer works well. In such situations the hardware makes all these paths execute sequentially, even though each thread executes only one of the paths and execution of the warp will require multiple passes: one pass for those that choose *if* part and another pass for those choose *else* part.

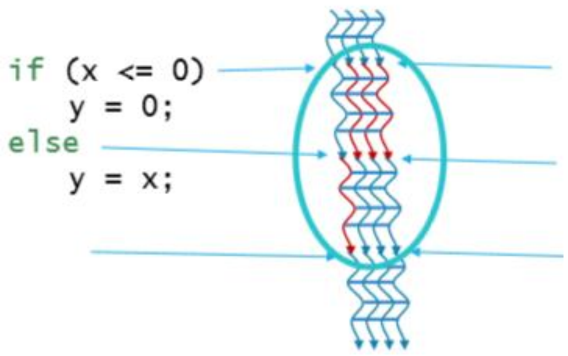

Figure 3: Branch Divergence

4. The Optimizations

For this used two scheduling techniques i.e. fixed scheduling and dynamic scheduling have been used. Two variants of dynamic scheduling are Frequency scheduling and balanced scheduling [8]. Compared the execution time of these schedules with native scheduling. The calculation of End Semester Examination marks of Engineering and MBA students is the application to which the scheduling techniques are applied. The divergence condition checked whether the student belongs to Engineering Section or MBA Section and based on that calculations are happening.
First, created a database which contains 257000 records of the students' details such as name, register number, course, number of subjects, mark of different subjects etc. Then copied these details to a student record and passed to kernel function where the actual computations are carried out.

4.1 Native scheduling

It is the conventional serialization; serialization of a branch takes two consecutive time slots for SIMD units in a warp to process iteration because SIMD units cannot execute different tasks at the same time and one of the slots being always idle.

Algorithm:

```
Bool path
while condition1 do
    path = (bool)(course == 'Engineering')
    if path then
        Path A;
    else
        Path B;
end while
```

4.2 Fixed scheduling

Assuming the simplified case of ‘n’ iterations, each with equal processing demands in divergent branches for every thread in a thread block, i.e., time spent processing path A is equal to time spent processing path B subject to the branching condition [8].

Algorithm:

```
Bool path, next_path
next_path = 0
if condition1 then
  path = (bool)(course == 'Engineering')
while condition1 do
    next_path = !next_path;
    if path == next_path then
    if path then
        Path A
     else
        Path B;
     if condition1 then
        path = (bool)(course == Engineering')
     end if
end while
```

4.3 Frequency scheduling

It is based on majority voting by selecting the most frequent flow among the pending iterations [6]. In CUDA, __ballot () to perform warp voting operations across all lanes (usually with the size of 32) within a warp and should be supported by graphic cards with compute capability of at least 2.0. __ballot() is combining with __popc(), it can be used to accumulate the number of threads in each warp having a true predicate and returns the number of bits set with a 32-bit parameter[9].

Algorithm:

```
Bool path, next_path
int noA, noB
if condition1 then
  path = (bool)(course == 'Engineering')
while condition1 do
    noA = __popc(__ballot(path == true))
    noB = __popc(__ballot(path == false))
    next_path = noA > noB
     if path == next_path then
      if path then
         Path A;
      else
         Path B;
      if condition1 then
      path = (bool)(course == 'Engineering')
    end if
end while
```

4.4 Balanced scheduling

In frequency scheduling if a branch path occurs rarely, it takes a substantial number of iterations to collect a majority vote, resulting a prolonged vertical waste. Instead of a group-wide vote over the next path, voting is performed only by threads that lag most behind the rest of a group [8].

Algorithm:

```
Bool path, next_path
int noA, noB
int idle = 1, max = 1
if condition1 then
  path = (bool)(course == 'Engineering')
while condition1 do
  noA = _popc(__ballot(idle==max && path == true))
  noB = _popc(__ballot(idle==max && path == false))
   next_path = noA > noB
   if (noA != 0 && noB != 0) then
       max++;
    if path == next_path then
        if path then
            Path A;
         else
            Path B;
         if condition1 then
           path = (bool)(course == 'Engineering')
      end if
      else
        idle++
 end while
```

If statement mostly contains the same computation in both branches, which is most probably, remove from the divergence by the compiler optimization and count as overhead. Further, iteration limit should be much higher than no.of blocks * no. of threads that are using.

5. The Results and Discussions

Software implementation of the three iteration scheduling disciplines on the NVIDIA GPU. Except for the simplest native case, the reductions of the predicate expressions across the threads are needed. They are achieved by the hardware supported function, i.e. ballot() in CUDA C. Its 32-bit return value consists of each thread’s predicate value in a bit corresponding to the SIMD lane id [9]. By counting the number of bits set to 1, each thread arrives at the same result. The population count instruction popc() is used for that purpose, which is also realized in hardware on this GPU. The GPUs used were GeForce GT 540 M, based on the Fermi architecture which consists of two streaming multiprocessors each with 48 CUDA cores.

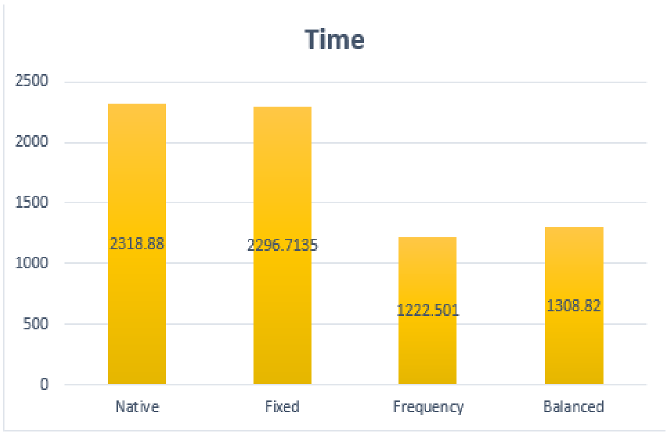

Figure 4: Scheduling Vs Execution time

The above graph shows time taken by different schedules. The performance will vary according to the branching probability and branching frequency. If statement mostly contains the same computation in both branches, which is most probably, remove from the divergence by the compiler optimization and count as overhead.
Fixed scheduling achieved negligible reduction in execution time than native scheduling. For the given size and nature of dataset, frequency scheduling and balanced scheduling accomplished the work with 48% and 44% reduction of execution time respectively.

6. Conclusion

This paper has presented a performance analysis of repeated divergence reduction technique by preserving the aids of SIMD. The execution time depends on the adopted scheduling technique and branching sequence. The more important decision than choosing between the two dynamic scheduling algorithms is the question of payoff threshold at which iteration scheduling becomes rewarding. The time spent in each divergent path, as well as the time spent in non-divergent parts of a loop can be approximated reasonably well by the instruction count at least in compute-bound kernels. On the other hand, the expected number of iterations a SIMD is concurrently processing a divergent path depends on the selected algorithm and branching probabilities.

References

[1] H. Kai and B. Faye A, Computer Architecture and Parallel Processing, McGraw-Hill International Edition, 1985.

[2] B. W. Coon, J. R. Nickolls, J. E. Lindholm and S. D. Tzvetkov, Structured Programming Control Flow in a SIMD Architecture, Washington D.C: NVIDIA Corp, 2011.

[3] W. L. Fung and T. M. Aamodt, "Thread block compaction for efficient SIMT control flow," in *IEEE Computer Society*, Washington, DC, 2011.

[4] M. Steffen and J. Zambreno, "Improving SIMT efficiency of global rendering algorithms with architectural support for dynamic micro-kernels," in *IEEE/ACM Int'l Symp. Microarchitecture (MICRO '10)*, 2010.

[5] M. Rhu and M. Ere, "CAPRI: Prediction of compaction-adequacy for handling control-divergence in GPGPU architectures," *SIGARCH Comput. Archit. News*, pp. 61-71, June 2012.

[6] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in GPU programs," in 4th Workshop General Purpose Processing on Graphics Processing Units (GPGPU '11), 2011.

[7] B. K. David and W. H. Wen-mei, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann Publishers Inc, 2010.

[8] N. Roman, "Loop Optimization for Divergence Reduction on GPUs with SIMT Architecture," in IEEE Transaction Parallel and Distributed computing, 2014.

[9] NVIDIA Corp., "PTX: Parallel Thread Execution ISA Version 2.3," Santa Clara, 2011.

Anju Soosan Baby received her BTech degree in computer science from Mahatma Gandhi Unive-rsity, Kerala in 2012 and MTech from Christ Uni-versity, Bangalore in 2015. Her rese-arch inter-est is Parallel Computing.

K Balachandran, Associate Professor of CSE, pursued his BSc and MCA from Bharathidasan University, MSc Physics from Annamalai University, MTech (IT) from Allahabad Agri-cultural University and MPhil from Alagappa University. He has wo-rked as a scientific officer for two decades with the Department

Atomic Energy, India. Many of his research papers have won the best paper award at national and international conferences. His area of research includes Computer Science and software engineering.