THESIS

# REDUCING OFF-CHIP MEMORY ACCESSES OF WAVEFRONT PARALLEL PROGRAMS IN GRAPHICS PROCESSING UNITS

Submitted by

Waruna Ranasinghe

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2014

Master's Committee:

Advisor: Sanjay Rajopadhye

Wim Bohm
Iuliana Oprea

ABSTRACT

REDUCING OFF-CHIP MEMORY ACCESSES OF WAVEFRONT PARALLEL PROGRAMS IN GRAPHICS PROCESSING UNITS

The power wall is one of the major barriers that stands on the way to exascale computing. To break the power wall, overall system power/energy must be reduced, without affecting the performance. We can decrease energy consumption by designing power efficient hardware and/or software. In this thesis, we present a software approach to lower energy consumption of programs targeted for Graphics Processing Units (GPUs). The main idea is to reduce energy consumption by minimizing the amount of off-chip (global) memory accesses. Off-chip memory accesses can be minimized by improving the last level (L2) cache hits. A wavefront is a set of data/tiles that can be processed concurrently. A kernel is a function that get executed in GPU. We propose a novel approach to implement wavefront parallel programs on GPUs. Instead of using one kernel call per wavefront like in the traditional implementation, we use one kernel call for the whole program and organize the order of computations in such away that L2 cache reuse is achieved. An strip of wavefronts (or a pass) is a collection of partial wavefronts. We exploit the non-preemptive behavior of the thread block scheduler to process a strip of wavefronts (i.e., a pass) instead of processing a complete wavefront at a time. The data transfered by a partial wavefront in a pass is small enough to fit in L2 cache, so that, successive partial wavefronts in the pass reuse the data in L2 cache. Hence the number of off-chip memory accesses is significantly pruned. We also introduce a technique to communicate and synchronize between two thread blocks without limiting the number of thread blocks per kernel or SM. This technique is used to maintain the order of wavefronts.

We have analytically shown and experimentally validated the amount of reduction in off-chip memory accesses in our approach. The off-chip memory reads and writes are decreased by a factor of 45 and 3 respectively. We have shown that if GPUs incorporate L2 cache with write-back cache write policy, then off-chip memory writes also get reduced by a factor of 45. Our approach provides 98% and 74% L2 cache read hits and total cache hits respectively and the traditional approach reports only 2% and 1% respectively.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

In this chapter, we describe the problem, why it is important, the approach we propose to solve the problem, contributions of the thesis, a summary of results, and related work.

## 1.1. Problem

Reducing the energy consumption of wavefront parallel programs targeted for GPUs.

## 1.2. Why problem is important?

There have been several studies carried out on the feasibility of exascale computing [1, 2]. DARPA IPTO Division supported the first study [1] in 2008 and then MITRE Cooperation presented a mid evaluation [2] of the challenges for exascale computing in 2013. The first study identified four major technical challenges for exascale computing and *energy and power wall* is one of them. If the power consumption of current technologies, is simply extrapolated to exascale computing, it is impossible to practically achieve such deployment. It would require about 400 Megawatts to operate. They have identified *the cost of moving data from processor to memory or processor to processor* as the main issue. There are two main research areas in order to reduce the energy consumption of data transfers between off-chip memory and processor. They are, a) hardware architecture; b) software implementation.

Accelerators are used in supercomputers and contribute 35% of the total TOP500 performance. In addition, 7 out of 9 most energy efficient architectures in TOP500 supercomputers are powered by GPUs [3]. While GPUs contribute towards the performance, they also play a key role in producing energy efficient supercomputers. Therefore, reducing the energy

consumption of programs targeted for GPUs will further increase the energy efficiency and contribute towards breaking the exascale power wall.

Apart from exascale challenges, reducing energy consumption also reduces the component failure rate of GPUs [4] as well as provides economic gains by reducing the operational costs (like power consumption for the hardware and power consumption for the cooling system) [5].

In stencil computations, array elements are updated according to a fixed pattern of values of the neighboring elements. Almost all the stencil like programs are parallelized using wavefront parallelism introduced by Lamport in 1974 [6]. The applications of stencils and wavefront parallelism are spread over many areas including particle physics simulations [7], parallel iterative solvers [8], and triangular systems of linear equations. Therefore, reducing the energy consumption of wavefront parallel programs affects the energy consumption of decent amount of applications in various disciplines.

## 1.3. Approach

Off-chip memory accesses can be reduced by improving the last level (L2) cache hits. The typical way of implementing wavefront parallel programs is by using multiple GPU kernel calls (one per wavefront) sequentially. For large enough wavefronts, data required by the wavefront does not fit in L2 cache, therefore, each successive kernel calls read data from off-chip memory, instead of L2 cache.

We introduce a novel mechanism to communicate and synchronize between pairs of thread blocks. By using this communication mechanism, programs can be implemented using one kernel call instead of using multiple kernel calls. In GPUs, only a fixed number of thread blocks get executed concurrently. By exploiting this behavior and by forcing one thread block to process a row of data/tiles, we can ensure that the data/tile space is processed in passes

of certain width $P_h$. Now, size of the partial wavefronts is $P_h$ which is small enough to fit in L2 cache. Therefore, next partial wavefront reads the data from L2 cache. After finishing the first pass, it will move on to the next pass. The proposed implementation reduces the amount of off-chip memory accesses. The idea of multi-pass approach was introduced by Rajopadhye et al. [9] where they propose a GPU-like accelerator called Stencil Processing Unit (SPU) for implementing stencil computations in an energy efficient manner.

## 1.4. Contributions

The main contributions of this thesis are as follows,

(1) A new approach to implement wavefront parallel programs in GPUs, so that, the amount of off-chip accesses are reduced.

(2) A novel way to communicate and synchronize between two CUDA thread blocks, that enforces the order of execution of computations between two thread blocks, independent of the scheduling decisions of the runtime system.

(3) A careful analysis for savings of off-chip memory accesses, both analytically and experimentally using profiling tools.

(4) An energy model for our proposed energy efficient implementation. This is an extension of our joint work done on developing an energy model for tiled GPU programs that implement wavefront parallelization of stencil computations, by Rajopadhye et al. [10].

(5) A micro-benchmark suite to determine the energy consumption of various GPU operations.

(6) A tool to measure the energy consumption of kernels on NVIDIA GPUs

## 1.5. Summary of results

We have analytically shown and experimentally validated the amount of reduction in off-chip memory accesses in our approach. The off-chip memory reads and writes are reduced by a factor of 45 and 3 respectively. The disparity is because the GPU caches use *write-through* cache write policy. If GPUs incorporate L2 cache with *write-back* cache write policy, then off-chip memory writes also get reduced by a factor of 45. Our approach provides 98% and 74% of L2 cache read hits and total cache hits respectively and the traditional approach reports only 2% and 1% respectively. We have also experimentally shown that despite the significant reduction in off-chip memory accesses, the total energy consumption savings are modest. The contribution of energy of off-chip memory accesses in the traditional implementation must be in the same order as the contribution of energy from other components like computations. When this is the case we will see considerable amount of dynamic and total energy savings using our approach.

## 1.6. Related Work

1.6.1. Techniques to synchronize thread blocks. Even though GPUs started off as graphics processors, today's GPUs are general purpose parallel processors with support for programing interfaces and languages like C. The massive computing power of GPUs attracts more and more complex applications (GPGPU)[1] to be ported and developed for GPUs. While the programming constructs in GPUs are sufficient for embarrassingly parallel programs, it is lack of programming constructs to implement more complex programs. For example, absence of a proper mechanism to communicate among thread blocks. There are a number of studies on techniques to synchronize among thread blocks. Xiao et al. [11]

[1]See `http://gpgpu.org` for an exhaustive list of GPGPU applications

propose a synchronizing mechanism among thread blocks by using atomic instructions in CUDA. Since they seek to synchronize among all the thread blocks, they restrict the number of the thread blocks per kernel to the number of SMs and thread blocks per SM to one, to avoid deadlocks. This could affect the performance, since GPU may need more than one thread block per SM to overlap the compute and memory operations. Later, Xiao and Feng [12] proposed a similar technique which has the same flaw in performance.

In many instances like in wavefront parallelized tiled computations, we do not need to barrier synchronize among all the thread blocks at once, but just producer-consumer synchronization between pairs of thread blocks is sufficient. We just need to know whether a thread block has finished its computations up to a certain point. Then the next thread block can start processing. Most of the stencil like wavefront parallel programs have similar execution patterns and do not need a method to synchronize among all the thread blocks at once. Therefore, method we propose does not impose a restriction on the number of thread blocks per kernel. Our technique also may deadlock, but we have taken care of it without restricting the number of thread blocks as well as thread blocks per SM. One Drawback of our method compared to the related work is that, the memory complexity of our method is $O(threadblocks)$, more precisely, equal to the number of thread blocks instead of having just one memory location. Since, the number of thread blocks is much smaller than the problem (input) sizes and GPUs has relatively large off-chip memory capacity, this is not an issue.

1.6.2. Techniques for Improving Energy Efficiency. The area of energy consumption in CPUs has been explored over many years. Although the area of energy consumption in GPUs is a hot topic nowadays, it is yet to be deeply explored. Mittal and Vetter has recently surveyed on state of the art for improving energy efficiency of GPUs [13]. The

techniques can be classified into five main categories, a) dynamic voltage/frequency scaling (DVFS); b) CPU-GPU work load balancing; c) architectural improvements; d) dynamic resource allocation; and e) programming level techniques.

DVFS is a technique where the power/energy is controlled by dynamically changing the voltage of a processor. This change in voltage affects the frequency of the core as well. Anzt et al. [14] propose an technique which based on using DVFS on CPUs while waiting for GPU to finish its kernels. Jiao et al. [15] explore the space of applying DVFS on both GPU cores as well as off-chip memory. Lin et al. [16] propose an approach based on using software pre-fetching together with DVFS.

The next technique is to balance CPU-GPU work load based on the energy consumption. It has been observed that CPU has better energy efficiency for some operations and GPU has better energy efficiency for others. Based on this, by dynamically selecting the best option (out of CPU and GPU) for a given kernel, energy consumption can be reduced. Ma et al. [17] propose a technique to load balance work load among CPU and GPU based on the execution time, so that, both CPU and GPU finish at the same time. Therefore, we can minimize the energy consumed while the CPU is idling. Rofouei et al. [18] describe a technique to chose between CPU and GPU based on the energy efficiency of the kernel.

Improve GPU architecture to provide better energy efficiency while running kernels. This is achieved by changing different architectural components to optimize the energy consumption. Wang et al. [19] propose to put L1 and L2 cache into state-preserving low leakage mode when ever there are no threads scheduled to access L1 and L2 cache. This is a vital improvement when programs are executed with caches disabled or without any shared memory. Gebhart et al. [20] describe an technique to reduce the amount of memory required to

store thread context. Gebhart et al. [21] propose to unify the L1 cache and shared memory. Therefore, before starting the kernel, the proportion of L1 and shared memory can be configured which increase the resource utilization. Rhu et al. [22] propose a technique to find the optimal granularity of off-chip memory transactions, thereby, reducing the unused data fetches. Gilani et al. [23] propose a fused multiply-add unit for integers and a scalar unit to compute common operation with same operands for all the threads within a warp. Rajopadhye et al. [9] propose a GPU-like architecture called Stencil Processing Unit (SPU), for implementing dense stencil computations in an energy-efficient manner. The multi-pass technique, we are using in this thesis was proposed by them.

Dynamic resource allocation is determining the optimal amount of resources for a given kernel based on the fact that some of the programs may not utilize all the resources available in GPUs. Hong and Kim [24] describe a power model and use of the model to determine the optimal number of cores to achieve highest power efficiency. Wang et al. [25, 26] also propose techniques to determine the optimal number of cores. Energy consumption can be reduced by shutting down the unused cores. Song et al. [27] has gone one step beyond and has proposed a technique to throttle the number of thread blocks per Streaming Multiprocessor. Jararweh and Hariri [28], and Wang and Chen [29] propose techniques to manage energy consumption of GPU based clusters.

Programming level techniques are programming transformations and application specific optimizations that reduce the energy consumption of GPU kernels. My technique also belong to this category. Wang et al. [30] describe a technique to save energy by using kernel fusion. In order to find a successful fusion, there should be kernels which are underutilizing the resources of GPU as well as complement the resources utilized by each other. Alonso et al. [31] propose to put the CPU thread in a blocking state rather than in a busy wait polling

loop while waiting for the GPU to finish the computation. Yang et al. [32] study several opensource GPU projects to identify the inefficient code patterns and suggest alternatives so that it improves the energy efficiency of the kernel.

While some of the techniques reduce the performance of the GPU kernel, our technique does not reduce the performance but may improve it. Our technique is also orthogonal to almost all available techniques and complements them.

# CHAPTER 2

# BACKGROUND

In this chapter we present background information that may help to follow the rest of the thesis. We specifically discuss the GPU architecture, the different components of energy/power, tiling and wavefront parallelism, and Smith-Waterman algorithm.

## 2.1. GPU Architecture and Programming Model

A GPU consists of an array of Streaming Multiprocessors (SMs) each one consisting of a set of Scalar Processors (CUDA cores). The number of CUDA cores can vary from 8 to 192 depending on the GPU architecture. All SMs share an off-chip global memory and have its own shared memory and a register file. Modern GPUs are also equipped with a last level cache L2 and L1 cache. The size of the cache is much smaller compared to the CPUs because, GPUs are originally designed for streaming or throughput computing which has limited data reuse [33]. The programming model consists of a grid of *thread blocks*. The threads of a thread block execute concurrently on one SM and multiple thread blocks may execute concurrently on one SM (depending on the availability of shared memory and registers). As thread blocks terminate, new thread blocks are launched [34]. It is expected that the computations in a thread block is independent of the computations in other thread blocks. In other words, one cannot assume the execution order for the grid of thread blocks, and in fact, the GPU runtime system schedules a thread block through to completion without any pre-emption. This enable GPUs to have an extremely lightweight runtime system. Within a thread block, all the threads can access shared memory collaboratively and explicitly synchronize to have a uniform view of shared memory amongst each other. Within a thread block, code is executed in groups of 32 threads called *warp*. Each thread has its private registers. GPUs consist

of different types of memory spaces designed to cater to various types of access patterns of data. A summary of different memory spaces of GPUs is given below. Please refer to the CUDA C programming guide [34] for more details.

(1) Global memory

Global memory is high capacity and shared among all the streaming multiprocessors (SMs) and persistent across multiple kernel calls. It is used to store the input and output data of kernels. Global memory accesses are cached in L2 cache which is shared across all the SMs. Global memory accesses are also cached in L1 cache which is local to each SM. L1 cache is not coherent. Global memory accesses must be coalesced (threads in a warp must access consecutive elements in an array) to have the best performance.

(2) Shared memory

Shared memory is shared among all the threads within a thread block. There is one shared memory per SM. The capacity of the shared memory is low. Therefore, it should be used carefully so that we have enough resources left to accommodate multiple thread blocks in the same SM. So that, the memory latency can be hidden. Shared memory is highly banked to increase the concurrent accesses. Therefore, we need to make sure that there are no bank conflicts from accesses of a warp (or half a warp) of threads to shared memory. All the conflicted access will be serialized and degrade the performance. Shared memory and L1 cache reside in the same physical memory. Prior to launch the GPU kernel, one can configure the amount of L1 and shared memory sizes out of a predefined set of configurations. Data is persistent within a lifetime of a thread block.

(3) Registers

Located closest to the functional units and have the minimum access latency. A register is bound to a single thread and not shared among the threads. Number of registers per SM is limited and this is critical resource which limits the number of thread blocks per SM. Size of a register is 32bits. Each thread has a hard limit on the number of registers that can be used. Whenever this limit is exceeded, all the remaining register data are spilled in to local memory which resides in device memory, therefore has a high latency. Local memory get cached in L1 cache. Register spilling can also happen when we declare large register arrays, this can be resolved by declaring the array as volatile, but this may prevent the compiler optimizations.

(4) Constant memory

Constant memory resides in the device memory and has a constant cache. Constant memory is read only. Threads in a warp should access the same memory location in constant memory to have the best performance.

(5) Texture memory

Texture memory resides in device memory and has a cache. This is a read only memory. The texture cache is optimized for 2D spatial locality. Unlike other memories, texture has the capability of returning processed data.

(6) Read-only memory

Read-only memory is available in Kepler GPUs. This is the same memory space as texture memory, but has direct access to memory and no need to use texture in-built functions.

## 2.2. Components of Energy Consumption

The energy consumption of GPUs consists of two main parts, namely static energy and dynamic energy. Static power refers to the power consumption even when the GPU is idling. Dynamic power is the power consumed by different components of the GPU while in operation. Dynamic power can be decomposed into power consumed by memory operations and power consumed by compute operations. Power consumed by memory operations can be further decomposed into power consumed by off-chip data transfers and on-chip data transfers. We can keep on expanding this further. But I'm going to stop right here and derive the formulas corresponding the above description. The symbols are defined in the Table 2.1

$$E_{\text{alg}} = E_{\text{stat}} + E_{\text{dyn}}$$

$$E_{\text{dyn}} = E_{\text{mem}} + E_{\text{ops}}$$

$$E_{\text{mem}} = E_{\text{offchp}} + E_{\text{onchp}}$$

$$E_{\text{offchp}} = M_{\text{io}} e_{\text{gr}} \tag{1}$$

The total energy of the program depends the amount of off-chip data transfers. Therefore, by reducing the off-chip data transfers, we can reduce the total energy consumption.

## 2.3. Stencil Computations

In stencil computations, values of elements are updated according to a fixed pattern of values of the neighboring elements. In other words, computation can be represented using a recurrence. The values of the elements are updated iteratively. Jacobi is an example for a stencil and Smith-Waterman algorithm is an example for a stencil *like* program.

TABLE 2.1. Description of symbols

| Symbol | Description |
|---|---|
| $E_{\text{alg}}$ | Total energy consumption of the program |
| $E_{\text{stat}}$ | Static energy consumption of the program |
| $E_{\text{dyn}}$ | Dynamic energy consumption of the program |
| $E_{\text{mem}}$ | Energy consumed by memory operations of the program |
| $E_{\text{ops}}$ | Energy consumed by compute operations of the program |
| $E_{\text{offchp}}$ | Energy consumed by off-chip data transfers of the program |
| $E_{\text{onchp}}$ | Energy consumed by on-chip data transfers of the program |
| $E_{\text{tile}}$ | Dynamic energy consumed by a tile |
| $p_{\text{stat}}$ | static power of device |
| $e_{\text{gr}}$ | Energy for a off-chip↔register transfer |
| $e_{\text{sr}}$ | Energy for a shared↔register transfer |
| $e_{\text{gs}}$ | Energy for a off-chip↔shared transfer |
| $e_{\text{sync}}$ | energy for a single synchronization |
| $e_{\text{j}}$ | Energy per operation of type `j` |
| $M_{\text{offchp}}$ | Number of off-chip data transfers of the program |
| $N_{\text{tiles}}$ | Number of tiles in the program |
| $E_{\text{gs}}$ | Energy consumed for off-chip, shared memory transfers per pass |
| $E_{\text{ls}}$ | Energy consumed for L2, shared memory transfers per pass |
| $E_{\text{sr}}$ | Energy consumed for register, shared memory transfers per pass |
| $E_{\text{ar}}$ | Energy consumed for computations per pass |
| $E_{\text{sync}}$ | Energy consumed for thread synchronizations per pass |
| $M_{\text{io}}$ | Number of off-chip data transfers per pass |
| $P_{\text{h}}$ | Height of a pass in tiles |
| $P_{\text{w}}$ | Width of a pass in tiles |
| $N_{\text{P}}$ | Number of passes in the program |
| $T_{\text{alg}}$ | Execution time of the program |
| $V_{\text{oc}}$ | Volume of data related to off-chip accesses of the program |
| `S` | Space dimention |
| `T` | Time dimention |
| $\texttt{op}_{\texttt{j}}$ | Number of operations of type `j` in loop body. |
| $t_{\text{S}}$ | Tile size along space dimention |
| $t_{\text{T}}$ | Tile size along time dimention |
| $k$ | Number of thread blocks that get scheduled concurrently at time |
| $M_{\text{smem}}^{\text{max}}$ | Shared memory available per a SM |
| $M_{\text{reg}}^{\text{max}}$ | Registers available per a SM |
| $M_{\text{smem}}^{\text{alg}}$ | Shared memory consumed by program per thread block |
| $M_{\text{reg}}^{\text{alg}}$ | Registers used by program per thread |

## 2.4. TILING OF PROGRAMS AND WAVEFRONT PARALLELISM

Tiling [35**?** , 36] is a program transformation to improve the locality of memory accesses. Tile is a group/block of computations. There are many ways to tile programs. *Orthogonal*

*tiling* [35] is performed by making the tile boundaries parallel to the boundaries of the original domain. While, *orthogonal tiling* works for Smith-Waterman like programs, it is not legal for programs with Jacobi like dependences. Jacobi like stencils can be tiled easily by tiling the space dimensions, but it results in memory bound programs rather than compute bound. There are other tiling techniques like *diamond tiling* [36], *hybrid hexagonal tiling* [37], and *time skewing* [38–40] followed by rectangular tiling to obtain compute bound tiles. Once tiling is applied, we can find wavefronts where tiles within an wavefront can be processed simultaneously. The idea of wavefronts was introduced by Lamport [6] and he named it as *hyperplane* method.

## 2.5. Smith Waterman

Smith Waterman [41] is a dynamic programing solution to find the optimal local alignments between two nucleotide sequences. Two nucleotide sequences are represented by $A = a_1 a_2 \dots a_N$ and $B = b_1 b_2 \dots b_M$. Substitution matrix is given by $s(a_i, b_j)$ which provides the similarity scores between two nucleotides. The weight for the deletions of length $k$ (gap penalty) is given by $W_k$. The optimal alignment score is given by the maximum score in table $H$ (see equation 2).

$$H(i,0) = 0, \qquad 0 \le i \le M$$

$$H(0,j) = 0, \qquad 0 \le j \le N$$

$$H(i,j) = max \begin{Bmatrix} 0 \\ H(i-1,j-1) + s(a_i, b_j) \\ max_{(k \ge 1)}\{H(i-k,j) - W_k\} \\ max_{(l \ge 1)}\{H(i,j-l) - W_l\} \end{Bmatrix}, \qquad 1 \le i \le M \quad \text{and} \quad 1 \le j \le N \quad (2)$$

There are two types of gap penalties, affine and profile based. In this study, we only consider affine gap penalties. Therefore, equation 2 can be simplified to equation 3.

$$
\begin{aligned}
&H(i,0) = 0, && 0 \leq i \leq M \\
&H(0,j) = 0, && 0 \leq j \leq N \\
&H(i,j) = max \left\{ \begin{array}{l} 0 \\ H(i-1,j-1) + s(a_i, b_j) \\ H(i-1,j) - W \\ H(i,j-1) - W \end{array} \right\}, && 1 \leq i \leq M \quad \text{and} \quad 1 \leq j \leq N \qquad (3) \\
&W = \rho + k\sigma
\end{aligned}
$$

where $\rho$ is the gap open penalty and $\sigma$ is the gap extension penalty. Affine gap penalty introduces another set of equations 4, where, the number of gaps are not tracked explicitly. Therefore, reduces the amount of branch statements in the implementation. This is achieved by retaining the previous row and column sums called $F$ and $E$, $m \times n$ tables [42]. This is the preferred algorithm for GPUs since branch divergence[1] serializes the execution, the performance of the CUDA kernel is reduced.

$$
\begin{aligned}
&H(i,0) = 0, && 0 \leq i \leq M \\
&E(i,0) = 0, && 0 \leq i \leq M \\
&H(0,j) = 0, && 0 \leq j \leq N \\
&F(0,j) = 0, && 0 \leq j \leq N
\end{aligned}
$$

[1]Branch divergence is threads within a warp taking different execution paths.

$$E(i,j) = max \left\{ \begin{array}{l} E(i,j-1) - \sigma \\ H(i,j-1) - \rho \end{array} \right\}, \qquad 0 \leq i \leq M \quad \text{and} \quad 1 \leq j \leq N \tag{4}$$

$$F(i,j) = max \left\{ \begin{array}{l} F(i-1,j) - \sigma \\ H(i-1,j) - \rho \end{array} \right\}, \qquad 1 \leq i \leq M \quad \text{and} \quad 0 \leq j \leq N \tag{5}$$

$$H(i,j) = max \left\{ \begin{array}{l} 0 \\ E(i,j) \\ F(i,j) \\ H(i-1,j-1) + s(a_i, b_j) \end{array} \right\}, \qquad 1 \leq i \leq M \quad \text{and} \quad 1 \leq j \leq N \tag{6}$$

From here onwards, the set of equations 4 with tables $F$, $E$ and $H$ are considered.

## 2.6. Energy models for GPU

Energy model is a cost model to compute the energy consumed by a program. Studies on CPU energy models have been carried out for more than a decade. There are energy models based on empirical data, performance counters and instruction mix [43–45], architectural simulations and event counters [46–49] and energy models for cache and RAM [50–52]. There are energy models which take the temperature into account as well [47, 48]. Shao [53] has described an instruction level energy model for another state of the art many core accelerator/processor, Xeon Phi.

Hong and Kim [24] presented a GPU power model to predict the number of optimal GPU cores to achieve the peak memory bandwidth for a kernel. An analytical model is used to predict the execution time [54] which has enabled prediction of the power consumption statically. They have also discussed the effects of temperature changes. However, they have predicted the minimum number of cores required for a program to achieve the peak memory

bandwidth of GPU. While this approach may work for memory bandwidth bound programs, it is unlikely to produce better results for compute-bound programs like tiled stencil computations. The model proposed by Rajopadhye et al. [10] is much simpler, because their model does not depend on warp,thread level parameters and number of ptx level instructions. Nagasaka et al [55] has modeled GPU power of kernels using performance counters. Lim et al [56], GPUWattch [57] and GPUSimPow [58] are simulation based power models. McPAT [49] is the basis for Lim et al [56] and GPUWattch [57] uses GPGPUSim [59] to simulate execution time. Simulation and performance counters based models are not feasible solutions when it requires to take decisions at compile time. Specially cycle accurate simulation methods take huge amount of time (i.e. GPGPUsim takes 11 hours to simulate 50ms kernel) to simulate a very short program [60]. Therefore, most of the time it is not feasible to use simulations to measure or model energy consumption of GPU kernels.

There are studies on energy models [61, 62] focused on reducing the energy for both CPU and GPU. Energy models are used to determine how to balance the load among CPU and GPU, so that it reduces the energy consumption.

# CHAPTER 3

# IMPLEMENTING WAVEFRONT PARALLELISM ON GPUS

In this chapter, we first explain the tiling of the iteration space (or dynamic programing table), then, we show how to come up with sub-tiles within a tile. Tiling and sub-tiling is common for both traditional and energy efficient (our) implementations. Then, we describe the traditional approach of implementing wavefront parallel programs and derive formulas for the amount of off-chip memory accesses. Finally, we describe our approach of implementing wavefront parallel programs and discuss the issues and solutions.

We use Smith Waterman as the reference wavefront parallel program and explain the tiling and implementation. The derivation of formulas for off-chip memory accesses are specific to rectangular tiling. These formulas can be derived for other tiling mechanisms in a similar way.

## 3.1. Tiling Smith Waterman

A cell in dynamic programming table $H$ of Smith Waterman depends on the west, south and southwest cells (see Figure 3.1). *Orthogonal tiling* [35] is legal (no cyclic dependences

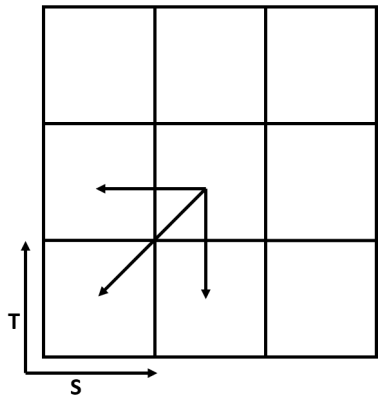


Figure 3.1. Dependencies of a cell in dynamic programming table $H$ of Smith Waterman

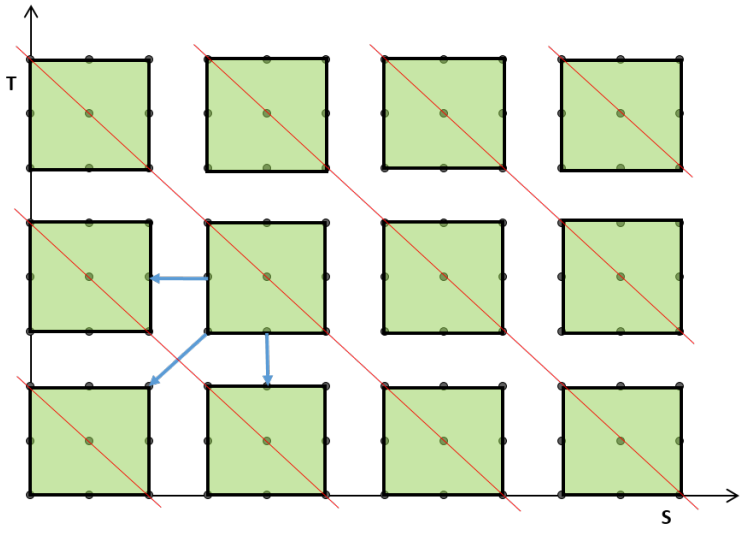


FIGURE 3.2. Orthogonal tiling for Smith Waterman. A tile depends on (blue arrows) the West, South and southwest tiles. Therefore, all the tiles in a wavefront (red lines) can be computed in parallel. Wavefronts are executed sequentially from southwest corner to northeast corner.

among tiles) for this dependence pattern. Therefore, the table space is tiled using rectangular tiles. All the tiles along a wavefront (northwest- southeast) can be computed in parallel (see Figure 3.2). A tile depends on the last column of tile to the west, last row of tile to the South and top right corner element of the tile to the southwest. The wavefronts are executed sequentially from southwest to northeast. Aforementioned execution order of wavefronts suffer from pipeline fill-flush stages. This is inherent to Smith Waterman due to the nature of the dependences (the tile is the southwest corner should be computed before computing any other tile). Therefore, diamond tiling [36] and hexagonal tiling [63] cannot be used for Smith Waterman algorithm.

## 3.2. Parallelization within a tile

In the previous section 3.1 we talked about the parallelization of tiles. In this section we are going to discuss the parallelization within a tile. Within a tile the dependences are shown in Figure 3.1. Therefore, all the elements in a diagonal can be processed in parallel. If we

implement in a naive manner, after computing each cell, a synchronization is needed. It may decrease the performance due to the overwhelming number of synchronizations. Therefore, we use sub-tiles within the tile. Sub-tiling for optimizing the Smith-Waterman on GPUs was introduced by Hains et al [64]. Instead of computing just one row of cells per thread, a few rows of cells (sub-tile height) are computed by a thread. A thread computes cells in column-major order. After computing a specified number of columns (sub-tile width), a synchronization is done. Then the next thread can start computing the sub-tile just above the current sub-tile. Therefore, a synchronization is done only per a sub-tile, upon the completion of the processing all the cells within a sub-tile. We can use Figure 3.2 to visualize sub-tiling as well. In the figure, green rectangles refer to sub-tiles and a row of sub-tiles processed by a thread. Steady-state wavefront is a large enough wavefront where all the threads are busy processing a sub-tile. Sub-tiling should make sure that number of columns of sub-tiles is large enough (or there exists steady-state wavefronts), so that, most of the time all the threads are busy processing cells. Having more threads than the number of columns of sub-tiles increases the number of idle threads and leads to poor performance. Therefore, in our experiments we do not consider data points corresponding to aforementioned inefficient sub-tiles. The data from the sub-tile to the left is produced by the same thread, therefore, registers can be used to transfer data. But data from the sub-tile below is generated by the previous thread, therefore, we need to use shared memory to transfer data among two adjacent threads. A thread can read data produced by another thread only if that thread has done a synchronization. The *syncthreads*() [34] function of CUDA is used to synchronize among threads in a thread block. By executing *syncthreads*() function, the data produced by a thread can be make visible to all the other threads in the thread block. One should take extra care when working with *syncthreads*() since there can be undefined behavior when

synchronizations are not used/placed properly. Most importantly, we should make sure that a *syncthread*() in our kernel is executed by all the threads in a thread block. In other words, we cannot use *syncthreads*() within code blocks with branch divergence. These limitations in CUDA programming model are mentioned in the appendix.

## 3.3. Traditional Implementation of Wavefront Parallelization for GPUs

The table space is first tiled as mentioned in section 3.1. All the cells in a tile are computed by a *thread block*. The tiles also have the same dependence pattern as shown in Figure 3.1. Therefore the tiles in a diagonal can be processed in parallel (one tile per thread block). Let the height and the width of a tile denoted by $t_{\mathrm{S}}$ and $t_{\mathrm{T}}$ respectively. The arrows show the data flow among tiles. Each tile (except for the tiles in the first row and first column) reads a column of size $t_{\mathrm{S}}$ from the tile to the left and reads a row of size $t_{\mathrm{T}}$ from the tile right below it. In addition, each tile reads a sequence of size $t_{\mathrm{S}}$ and a sequence of size $t_{\mathrm{T}}$, the total length of sequences is `S` and `T` respectively. The red diagonal lines refer to the set of tiles that can be processed in parallel (see Table 2.1 for the definition of symbols).

$$N_{\text{tiles}} = \frac{\mathtt{ST}}{t_{\mathrm{S}} t_{\mathrm{T}}}$$

In traditional implementation of Smith Waterman algorithm in GPU, a kernel call is made for each wavefront and the wavefronts are executed one after the other (sequentially) in the host (cpu) code. This sequential execution of the kernel calls ensure that the scheduling of tiles respects the data dependences among the tiles. We consider programs with large enough input sizes, so that, the amount of data accessed by a single wavefront does not fit in the last level (L2) data cache in GPUs. Therefore, for each wavefront, input and output data are fetched from global memory (off-chip). The amount of global memory access for

Smith-Waterman algorithm is given by equation 7

$$\text{Number of kernel calls} = N_w = \frac{\mathtt{S}}{t_\mathrm{S}} + \frac{\mathtt{T}}{t_\mathrm{T}} - 1$$

$$\text{DNA sequence data fetched per tile} = t_\mathrm{S} + t_\mathrm{T} \text{ characters} = t_\mathrm{S} + t_\mathrm{T} \quad \text{bytes}$$

$$\text{Table data fetched and written per tile} = 4(t_\mathrm{S} + t_\mathrm{T}) \text{ integers} = 16(t_\mathrm{S} + t_\mathrm{T}) \quad \text{bytes}$$

$$V_\mathrm{oc} = 17(t_\mathrm{S} + t_\mathrm{T})\frac{\mathtt{ST}}{t_\mathrm{S} t_\mathrm{T}} \tag{7}$$

### 3.4. Energy Efficient Implementation of Wavefront Parallelization

The traditional implementation of wavefront parallel programs uses a GPU kernel call per wavefront. As described in section 3.3 all the data that is required for the whole wavefront must be fetched from off-chip memory. If we can compute a piece of a wavefront (partial wavefront) at a time, we can improve the reuse in L2 cache. If we can make sure that the data required by partial wavefront is small enough to fit in L2 cache, then data required by the partial wavefront of the next wavefront is fetched from the L2 cache. After sweeping through the first set of partial wavefronts across all wavefronts, we can move on to the next set of partial wavefronts. We can visualize these sets of partial wavefronts as passes (see Figure 3.4).

Let's look at the implementation of aforementioned passes. We have two choices; 1) Use one kernel call per pass, 2) Use one kernel call for the whole program. We will first discuss about the 1$^{\text{st}}$ method and then move on to the 2$^{\text{nd}}$ method.

#### 3.4.1. One kernel call per pass.

Since a pass comprises of multiple rows of tiles, we can use one thread block to process a complete row of tiles (see Figure 3.3). Assuming that two issues relating to the correctness are addressed (and we will show how to do this later),

all the communication within the row of tiles (denoted by horizontal blue arrows) can be done through registers except for the data read by the first tile and the data written by the last tile. For the data transfers represented by blue arrows, we don't even need to lookup L2 cache. All the red and green arrows within a pass still refer to off-chip memory, but since data required by the wavefronts fits in L2 cache, all the off-chip memory references will hit L2 cache except for the first and last row of a pass. Therefore, we can avoid most of the off-chip communications as compared to the traditional implementation.

But, still we need a mechanism to retain the order of partial wavefronts. We present our mechanism in the section 3.4.3. If we make sure that the number of thread blocks or rows of tiles per pass is less than or equal to the number of thread blocks allowed to execute concurrently at a given time in the GPU, then we will not get into deadlocks as described in section 3.4.4 because, all the thread blocks in the kernel get scheduled to run concurrently.

The amount of kernel calls is equal to the number of passes, $N_{\mathrm{P}}$. Passes are executed sequentially one after the other. The $p^{\mathrm{th}}$ pass starts only after $p-1^{\mathrm{th}}$ pass completely finish its computations. There are couple of issues with this approach,

- Performance issues due to pipeline filling and flushing for each and every pass depends on the architecture of the GPU. Hence leaving thread blocks idling at the beginning and end of each pass; and
- The code will be architecture dependent, since the number of kernel calls depend on $P_{\mathrm{h}}$. $N_{\mathrm{P}} = \frac{\mathtt{S}}{t_{\mathrm{S}} P_{\mathrm{h}}}$ and $P_{\mathrm{h}} = N_{\mathrm{sm}} k$, where $k$ is the number of thread blocks that get scheduled concurrently per SM. $k = \min\left\{ \left\lfloor \frac{M_{\mathrm{reg}}^{\mathrm{max}}}{M_{\mathrm{reg}}^{\mathrm{alg}}} \right\rfloor, \left\lfloor \frac{M_{\mathrm{smem}}^{\mathrm{max}}}{M_{\mathrm{smem}}^{\mathrm{alg}}} \right\rfloor, \beta \right\}$, where $\beta$ is the maximum number of thread blocks allowed per SM for a given GPU architecture. Both $k$ and $N_{\mathrm{sm}}$ depends on the architecture of the GPU.

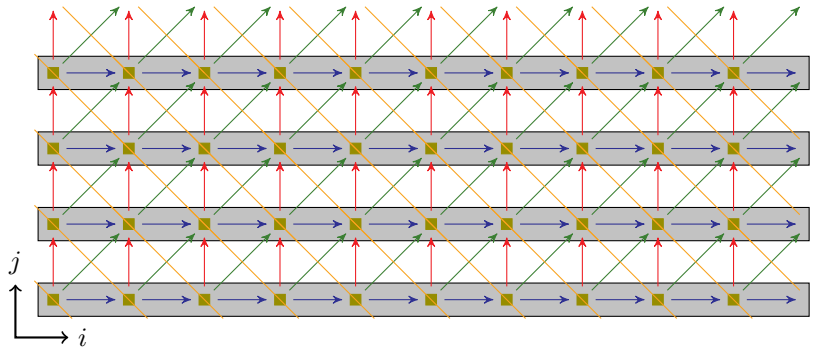


FIGURE 3.3. Parallelization within a pass. A thread block computes a whole row of tiles. A small green square represents a tile and arrow direction represents the data flow among tiles. Each gray rectangle represents a thread block. All 4 rows in the diagram belongs to the first pass of the program and this diagram shows only the first pass. Data flow shown by blue horizontal arrows can be done through registers (within same thread block). Therefore, it won't even need to lookup L2 cache. All the red and green arrows within the pass still refer to off-chip memory, but since data required by the wavefronts fits in L2 cache, all the off-chip memory references will hit L2 cache. Orange lines represent partial wavefronts.

3.4.2. ONE KERNEL CALL FOR THE WHOLE PROGRAM. We can solve both issues that we have discussed in the end of previous approach (section 3.4.1) by using one kernel call for the whole program. In this section, we discuss the problems we face in this approach and the solutions to the problems that we faced in the previous approach as well as the techniques we introduce to maintain the wavefront order.

A GPU has $N_{\text{sm}}$ streaming multiprocessors. If there are enough resources in a streaming multiprocessor to accommodate $k$ number of thread blocks concurrently, then, $P_{\text{h}} = N_{\text{sm}} * k$ thread blocks will be executed simultaneously in the GPU.

Due to the thread block scheduling mechanism of CUDA runtime in streaming multiprocessors, only $P_{\text{h}}$ thread blocks get scheduled at a time. If we assume that, thread blocks are scheduled in the order of tile row number (in general this is not true, but we resolve this problem later.), then, it will first schedule the first $P_{\text{h}}$ rows of tiles starting from the bottom of Figure 3.4. This scenario automatically divides the whole tile space into passes, each with

$P_{\mathrm{h}}$ consecutive rows of tiles. Therefore, we no longer depend on the architectural parameters like in the previous approach. Now, each pass is executed in wavefronts as shown in Figure 3.4. Once a pass is completed, then the next pass is processed. If we do this carefully, the next pass will not wait till the previous pass is completely finished. Instead, whenever a thread block finishes a row of tiles, the next available thread block (which start processing a row of tiles from the next pass) will be scheduled. Therefore, the overhead of pipeline filling and flushing is no longer an issue. The number of off-chip memory accesses are same as when we use *one kernel call per pass* (see section 3.4.1).

Let's derive the formula for the amount of off-chip memory accesses for the Smith-Waterman algorithm. The DNA sequence, aligned to the vertical axis of the dynamic programming table, fetched once per all the tiles. The first tile in each row fetches an array of size $t_{\mathrm{S}}$ of DNA sequence to registers, all the subsequent tiles in the same row reuse the same register array. The DNA sequence aligned to the horizontal axis must be fetched from global memory per pass, within the pass, it hits L2 cache without going for global memory. Following formulas represent the amount of off-chip memory accesses for both DNA sequence data as well as table data. Usually, stencil computations do not involve read only data like DNA sequences.

$$\begin{aligned}
\text{Total volume of DNA sequence data fetched} &= \mathtt{S} + \frac{\mathtt{ST}}{t_{\mathrm{S}} N_{\mathrm{P}}} \text{bytes} \\
\text{Total table data fetched and written} &= 16\mathtt{S} + \frac{16\mathtt{ST}}{t_{\mathrm{S}} N_{\mathrm{P}}} \text{bytes} \\
V_{\mathrm{oc}} &= 17\mathtt{S} + \frac{17\mathtt{ST}}{t_{\mathrm{S}} N_{\mathrm{P}}} \text{bytes} \qquad (8)
\end{aligned}$$

If we assume that $t_{\mathrm{T}} = t_{\mathrm{S}}$ then our energy efficient implementation provides a factor of $N_{\mathrm{P}}$ reduction of off-chip memory accesses compared to the traditional implementation. The

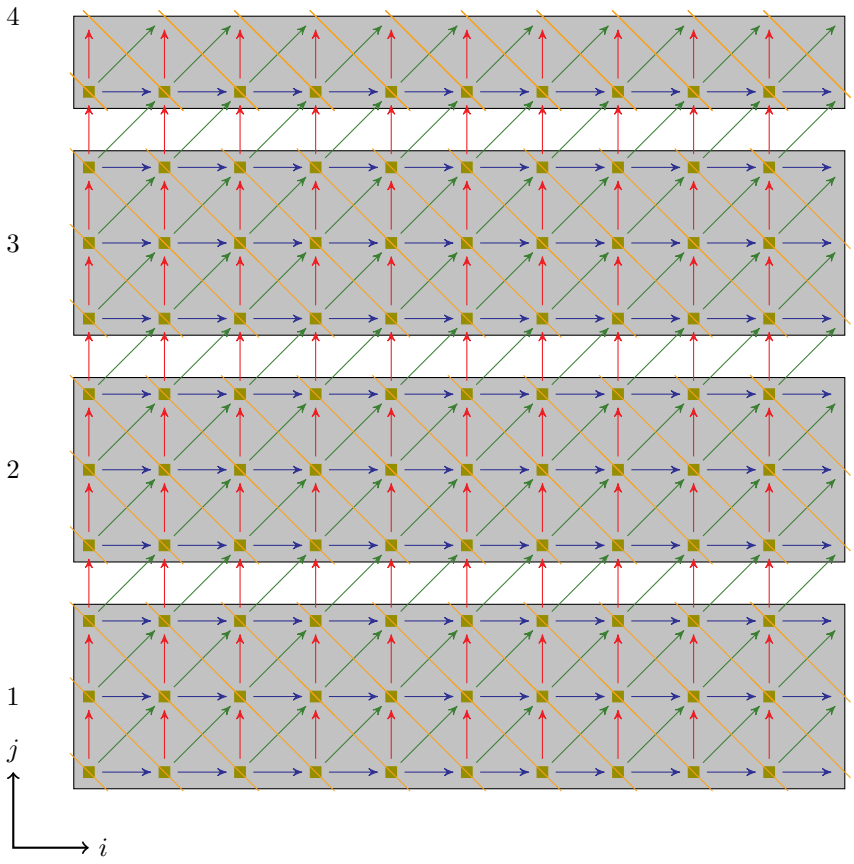


FIGURE 3.4. Passes of tiles. The diagram shows 4 passes. Starts from the first pass and moves on to the next pass as the current pass finishes computing. Within the pass, same wavefront parallelism is there, but with shorter wavefronts (partial wavefronts), so that, the data required by the wavefront fits in L2 cache. All the arrows between two passes represent a L2 cache miss and red and green arrows within a pass represent L2 cache hit. The partial wavefronts are shown in orange color.

formulas 8 are true if L2 cache has a *write-back* write policy. But most of the NVIDIA GPUs has a *write-through* cache, which make each write operation in L2 cache to go to off-chip memory. This will increase the number of global memory transfers compared to *write-back* cache policy. Let's derive the formulas again for this scenario.

$$\text{Total volume of DNA sequence data fetched} = \mathtt{S} + \frac{\mathtt{ST}}{t_{\mathrm{S}} N_{\mathrm{P}}} \text{bytes}$$

$$\text{Total table data fetched} = 8\mathtt{S} + \frac{8\mathtt{ST}}{t_{\mathrm{S}} N_{\mathrm{P}}} \text{bytes}$$

$$\text{Total table data written} = 8\mathtt{S} + \frac{8\mathtt{ST}}{t_{\mathrm{S}}}\text{bytes}$$

$$V_{\mathrm{oc}} = 17\mathtt{S} + \frac{\mathtt{ST}}{t_{\mathrm{S}}}\left(\frac{9}{N_{\mathrm{P}}} + 8\right)\text{bytes} \tag{9}$$

But, achieving this reduction in off-chip memory accesses is not trivial. Let's recall that in the traditional implementation, wavefront order is maintained by using one kernel call per each wavefront and by executing kernel calls one after the other. Since we are using just one kernel call for a pass or for the whole program, now we have to maintain the wavefront order explicitly.

3.4.3. Maintaining the execution order of wavefronts. One of the main challenges of the implementation described in Section 3.4 is maintaining the execution order of the wavefronts explicitly. This requires a mechanism to synchronize or communicate among CUDA *thread blocks*. Although CUDA does not provide any straight forward constructs to synchronize among thread blocks, we can implement a simple lock which can be used to maintain the order of the wavefronts. In general, a thread block should starts its computations only if the blocks that it depends on have already finished their computations.

To achieve this, one master thread in each thread block updates a variable in global memory upon end of the execution of a tile and waiting on a condition based on the value of a variable before starting the computations of a tile. The update of the variable indicates the successors (thread blocks who are waiting for the predecessors to finish the execution) to start the computation. The code structure of the locking/synchronization mechanism is shown in the code listing 3.1. We disabled the level 1 (L1) cache of the GPU, so that, all the local changes within the SM get flushed to higher level caches. Therefore, changes made

LISTING 3.1. Code structure for the locking mechanism

```
if (threadId == 0) {
        while(lock[my_row-1] <= column_of_tile)
        {} //spin wait
}
//Compute a tile here
//release/update the lock
if (threadId == 0) {
        lock[my_row]++;
}
```

within the SM are visible to the thread blocks in other SMs. We marked the arrays defined in global memory with *volatile* keyword to skip L1 cache[1].

3.4.4. AVOIDING DEADLOCKS. The other main challenge is to avoid deadlocks caused by the locking mechanism combined with the scheduling mechanism of CUDA thread blocks. CUDA assumes that thread blocks are independent therefore, order of execution of thread blocks is undefined. Hence, if we statically map $1^{st}$ thread block to process $1^{st}$ row of tiles, $2^{nd}$ thread block to process $2^{nd}$ row of tiles and so on, then, this mapping works only if the thread blocks are scheduled in in ascending order. This means that we make an assumption about the scheduling order of the thread blocks, and this contradicts with the CUDA programing semantics. Therefore, there is a chance for the last set of thread blocks get scheduled first, these thread blocks will wait till previous thread blocks update the corresponding global variable. But, these global variables will never be updated, since the thread blocks responsible to update the variables will not be scheduled until currently scheduled thread blocks finish their computation. This leads to a deadlock situation. Therefore, we need a dynamic mechanism to assign the row number of the tiles to the thread blocks as thread

[1]We also tried to disable L1 cache by using a compiler option. But it did not work. The kernel was either hanging or generating incorrect results depending on the GPU architecture (GTX 480, K20c). We currently do not have an explanation for this, but we hypothesize that the compiler command we used, does not work. We also saw similar behavior for some of the data points even when the arrays are defined as *volatile*.

LISTING 3.2. Assign the row number of the tiles to be processed by the thread block dynamically

```
if (threadId == 0) {
        row_of_tile = atomicInc(&tileRowNo, noOfRowsOfTiles);
}
```

blocks get scheduled in the GPU. We use a simple mechanism based on atomic increment function in CUDA which is proposed by Yan et al. [65]. As the first computation in the thread block, we increment a variable in global memory by 1 using one of the threads in the thread block (see listing 3.2). The thread block process the row of tiles which correspond to the result of atomic increment function. Therefore, which ever the thread block get scheduled next, will pick up the next row of tiles to process.

Since the main objective of reducing off-chip memory accesses is to reduce the energy consumption of the program, in the next chapter we derive models for the energy consumption of both traditional and our approach, using the hardware and software parameters.

# CHAPTER 4

# ENERGY MODELING

In this chapter, we describe energy models for both traditional and energy efficient implementations of wavefront parallel programs. We model only the energy consumption of the GPU. The results in this chapter were developed in collaboration with Prajapati et al. [10], and are part of a larger project. Here, we only deal with the case where we use rectangular tiles for programs with one dimensional data space, and extend the results to also model the optimized code that we described in Chapter 3. We also developed models for other types of tiling like hexagonal and for higher dimensions, but those results are not relevant to the work in this thesis.

The total energy consumed by a GPU is divided into two parts: *static* energy which is the energy consumed even when the GPU is powered on but idle, and *dynamic* energy which is the energy consumed for operations excluding the static part. Table 2.1 describes the parameters that have been used in our formulas. Static energy is proportional to the execution time. $E_{\text{stat}} = p_{\text{stat}} T_{\text{alg}}$. The dynamic energy is the weighted sum of different types of operations carried out during the program execution.

## 4.1. Traditional implementation

Dynamic energy can be represented as the dynamic energy consumed per a tile multiplied by the number of tiles $E_{\text{dyn}} = N_{\text{tiles}} E_{\text{tile}}$. The energy per tile can be further divided into energy for off-chip memory transfers, energy for on-chip transfers, i.e., between shared and register memory, energy for computations and energy for synchronizations among threads. During validation of the model, Prajapati et al. noticed that the contribution of energy towards $E_{\text{tile}}$ from $E_{\text{sync}}$ is negligible compared to other parameters. Therefore, the term

$E_{\text{sync}}$ is safely dropped.

$$\begin{aligned} E_{\text{tile}} &\approx E_{\text{gs}} + E_{\text{sr}} + E_{\text{ar}} \\ &= e_{\text{gs}} M_{\text{io}} + E_{\text{sr}} + E_{\text{iter}} V_{\text{tile}} \end{aligned} \tag{10}$$

Having computed the static and dynamic energy the total energy is just

$$E_{\text{alg}} = E_{\text{stat}} + E_{\text{dyn}} = p_{\text{stat}} T_{\text{alg}} + N_{\text{tiles}} E_{\text{tile}}, \tag{11}$$

where, $N_{\text{tiles}}() = \frac{\mathtt{ST}}{V_{\text{tile}}}$, and $V_{\text{tile}}$ is the volume of (number of computations in) a tile, $V_{\text{tile}} = t_{\text{S}} t_{\text{T}}$.

4.1.1. Rectangular Tiling. The following equations expand on the components of energy per tile equation 12. $M_{\text{io}} = \alpha\,(t_{\text{S}} + t_{\text{T}})$. $\alpha = 4.25$ for Smith-Waterman. The integer part and fractional part of the constant 4.25 refers to the data transfers to/from integer arrays and char array (represented in 4byte units) respectively.

$$\begin{aligned} E_{\text{gs}} &= \alpha\,(t_{\text{S}} + t_{\text{T}})\, e_{\text{gs}} \\ E_{\text{sr}} &= \left( V_{\text{tile}} + 5 t_{\text{T}} \frac{t_{\text{S}}}{s_{\text{S}}} \right) e_{\text{sr}} \\ E_{\text{iter}} &= 5 e_{\text{add}} + 5 e_{\text{max}} \\ E_{\text{tile}} &= E_{\text{gs}} + E_{\text{sr}} + E_{\text{ar}} \\ &= \alpha\,(t_{\text{S}} + t_{\text{T}})\, e_{\text{gs}} + \left( V_{\text{tile}} + 5 t_{\text{T}} \frac{t_{\text{S}}}{s_{\text{S}}} \right) e_{\text{sr}} \\ &\quad + 5 V_{\text{tile}}\,(e_{\text{add}} + e_{\text{max}}) \end{aligned}$$

where, $V_{\text{tile}} = t_{\text{S}} t_{\text{T}}$

## 4.2. Energy efficient implementation

In this section, we extend the above energy model to represent the energy efficient implementation that is described in the section 3.4. There are two main differences in the energy model compared to the energy model for the tradition implementation: the granularity of dynamic energy is presented per pass instead of per tile, and in addition to the dynamic energy components modeled above, we introduce a new energy parameter: energy for memory transfers between L2 cache and registers.[1]

As before, we ignore the energy for synchronization, and the corresponding formula is,

$$E_{\text{pass}} = E_{\text{gs}} + E_{\text{ls}} + E_{\text{sr}} + E_{\text{ar}}. \tag{12}$$

The formulas for $E_{\text{sr}}$ and $E_{\text{ar}}$ are same as before but need to multiply by the number of tiles per pass, $N_{\text{tilepass}}$.

$$P_{\text{h}} = \frac{\mathtt{S}}{t_{\text{S}} N_{\text{P}}}$$

$$P_{\text{w}} = \frac{\mathtt{T}}{t_{\text{T}}}$$

$$N_{\text{tilepass}} = P_{\text{h}} P_{\text{w}}$$

The computation of height of pass, width of pass, energy for off-chip transfers and energy for L2 transfers depend on the tiling technique. In this report, we consider simple rectangular tiling and derive the corresponding formulas.

$$E_{\text{gs}} = \left(\alpha P_{\text{h}} t_{\text{S}} + \beta P_{\text{w}} t_{\text{T}}\right) e_{\text{gs}}$$

[1]Here onward, all the symbols represent quantity per pass, instead of per tile.

where $\alpha$ and $\beta$ correspond to the number of arrays (with element of size 4bytes) accessed along the perimeter of a pass in parallel to `S` and `T` dimensions respectively. Most of the real world samples like stencil like programs, $\alpha = \beta$, therefore, we only use $\alpha$ by substituting for $P_{\mathrm{h}}$ and $P_{\mathrm{w}}$ we get,

$$E_{\mathrm{gs}} = \alpha \left( \frac{\mathtt{S}}{N_{\mathrm{P}}} + \mathtt{T} \right) e_{\mathrm{gs}}$$

Since a thread block computes a row of tiles, all the transfers along a row of tiles can be done through registers. Therefore, to count the number L2 cache hits we just need consider the transfers between row of tiles.

$$\begin{aligned} E_{\mathrm{ls}} &= \alpha \mathtt{T} \left( P_{\mathrm{h}} - 1 \right) \\ &= \alpha \mathtt{T} \left( \frac{\mathtt{S}}{t_{\mathrm{S}} N_{\mathrm{P}}} \right) \end{aligned}$$

The formula for $E_{\mathrm{ls}}$ is true if the write policy of L2 cache is *write-back*. Both read and write requests hit the cache and write hits are not immediately written to off-chip memory. If the cache write policy of L2 cache is *write-through* then despite the write hit in cache, it will be written back to off-chip memory immediately by reducing the energy savings of our technique. Then,

$$\begin{aligned} E_{\mathrm{gs}} &= \left( \alpha P_{\mathrm{h}} t_{\mathrm{S}} + \alpha P_{\mathrm{w}} t_{\mathrm{T}} + \frac{\alpha}{2} P_{\mathrm{w}} t_{\mathrm{T}} \left( P_{\mathrm{h}} - 1 \right) \right) e_{\mathrm{gs}} \\ &= \alpha P_{\mathrm{h}} t_{\mathrm{S}} + \frac{\alpha}{2} P_{\mathrm{w}} t_{\mathrm{T}} \left( P_{\mathrm{h}} + 1 \right) \\ &= \alpha P_{\mathrm{h}} t_{\mathrm{S}} + \frac{\alpha}{2} \mathtt{T} \left( P_{\mathrm{h}} + 1 \right) \\ E_{\mathrm{ls}} &= \frac{\alpha}{2} \mathtt{T} \left( P_{\mathrm{h}} - 1 \right). \end{aligned}$$

NVIDIA Fermi and Kepler GPUs have a L2 cache with *write through* write policy. We have not validated the proposed model experimentally, but, we strongly believe the accuracy of the model since this is a expansion to a thoroughly validated model.

# CHAPTER 5

# MICRO-BENCHMARKING

Data sheets provided by NVIDIA [66, 67] do not reveal information on the hardware parameters that we are interested in. In this chapter, we describe the approach we followed to determine hardware level energy parameters listed in Table 2.1. We use micro-benchmarks, which are small pieces of code each having an operation of interest stressed, and the NVIDIA NVML library to measure instantaneous power at any time of the execution of a micro-benchmark. Total consumed energy is equal to the product of the average measured power and the execution time of the micro-benchmark.

## 5.1. Methodology

We wrote micro-benchmarks for each hardware energy parameter in Table 2.1 to determine the values of these parameters experimentally. Micro-benchmarks are implemented in such away that the operation in focus is stressed, so that, the execution time and total energy consumption is dominated by the focused operation.

Before the execution of micro-benchmark, GPU is heated to a higher temperature. The execution time of the micro-benchmarks are large enough, for temperature of the GPU to reach a steady state. We obtain instantaneous power readings at different time instances of execution time while the micro-benchmark is being executed in the GPU. The average of power readings taken during the steady state temperature is considered as the average power of the micro-benchmark. Then, the total consumed energy is computed by multiplying the average power by the time period in which the average power is computed. Finally, by using the formula 13

$$e_{\mathrm{j}} = \frac{E_{\mathrm{dyn}}}{\mathtt{op}_{\mathrm{j}}} = \frac{E_{\mathrm{alg}} - E_{\mathrm{stat}}}{\mathtt{op}_{\mathrm{j}}} = \frac{E_{\mathrm{alg}} - p_{\mathrm{stat}} T_{\mathrm{alg}}}{\mathtt{op}_{\mathrm{j}}} \tag{13}$$

where the parameters in the formula are as defined in Table 2.1, the energy $e_{\mathtt{j}}$ for operation `j` can be calculated.

## 5.2. Implementation

We need static power consumption ($p_{\text{stat}}$) to compute all other energy parameters. Therefore, the static power consumption of the GPU is measured first. The static power of the GPU is obtained while the device is idle but operates in its highest performance state.

Micro-benchmarks are carefully implemented in such away that there are no shared memory bank conflicts and all the global memory accesses are coalesced. The body of the benchmark is repeated to make the focused operation dominant in both execution time and total energy consumption. These computations are simple enough so that *nvcc* compiler can easily optimize away most of the computations. Therefore, we have introduced minimum complexities to the micro-benchmarks to avoid aforementioned optimizations (see listing 5.1). We checked the *ptx* [68] code of the micro-benchmarks to validate that the computations are not optimized away.

NVIDIA NVML [69] library is used to obtain power readings of the GPU. The library reads power of the GPU once every 16 milliseconds in milliwatts for NVIDIA Kepler GPUs. The length of the interval depends on the GPU architecture and NVIDIA driver. We use the approach proposed by Lang et al. [70] to determine the interval. Average power is computed by taking the average of power readings over the time duration where the temperature is in steady state. Finally the equation (13) is used to compute energy parameters. The resulting parameter values are provided in the Table 5.1

LISTING 5.1. Structure of a micro-benchmark. This particular micro-benchmark is for the single precision "add" operation. "k" is a very large number. Statements that correspond to line numbers 8 and 11 change the values of the registers so that main computations (line 2-5) do not get optimized away

```
1  for (t = 0; t < k; t++) {
2    c_0_0 += a_0+b_0;
3    c_0_1 += a_0+b_1;
4           ⋮
5    c_2_2 += a_2+b_2;
6
7    a_0 = (a_0+1.1f)+1.7f;
8           ⋮
9
10   b_0 = (b_0+1.1f)+1.7f;
11          ⋮
12 }
```

TABLE 5.1. Table of the energy parameter values from the micro-benchmark approach

| Parameter Name [unit] | Value |
|---|---|
| $p_{\text{stat}}$ [W] | 48 |
| $e_{\text{gs}}$ [J] | $2.2\times10^{-9}$ |
| $e_{\text{sr}}$ [J] | $2.23\times10^{-10}$ |
| $e_{\text{fadd}}$ [J] | $5.3\times10^{-11}$ |
| $e_{\text{fmultiply}}$ [J] | $3.7\times10^{-11}$ |
| $e_{\text{iadd}}$ [J] | $7.2\times10^{-11}$ |
| $e_{\text{imax}}$ [J] | $4.8\times10^{-11}$ |

## 5.3. CHALLENGES

This section describes the challenges faced while implementing micro-benchmarks and measuring energy/power consumption.

5.3.1. CHALLENGES IN IMPLEMENTING MICRO-BENCHMARKS. The main challenge is to avoid optimization of computations. Since the computations in micro-benchmarks are simple and the dependencies among the computations are trivial, *nvcc* compiler optimizes the loops that we introduce to repeat computations, resulting in less amount of computations which is not we desire. Therefore, we need to introduce dependencies between iterations of the

loop so that loop does not get optimized away. There are few things that we need to be aware of before introducing dependencies. The loop body should be large enough, so that, we have enough instruction level parallelism (ILP). The computations that we introduce to change values of registers at the end of each iteration must use the same operation that is being benchmarked. One must go through the generated *ptx* code to verify that the loops are not optimized away and the loop body is dominated by the operation in focus. Finally, the performance of the micro-benchmark (in GFLOPS, GOPS, GB/s) must be comparable with the specification of GPU.

5.3.2. Challenges in measuring energy-consumption. NVIDIA announced their new power measurement functionality in NVML library at the end of 2013. It was a good news despite the limited support for GPU architectures, since we do not have to use external instruments or third party power modeling libraries to measure power of GPUs. We started off with implementing a C function to report power consumption of a GPU kernel (matrix multiplication using cuBLAS library [71]). Then we plotted instantaneous power readings against the time (see Figure 5.1). The shape of the power curve was not what we expected. Our expectation was sudden increase in power at the start and then constant power consumption, and at the end of the execution sudden drop of the power readings. But actual result is not the same as we expected. In the beginning power is gradually increasing with a inverse exponential slope (i.e. $y = \ln x$). It reaches the steady state power. At the end of the GPU kernel, power gradually decreases again following a exponential decay slope. Burtscher et al. [72] reported the same behavior and they claimed that the power readings at the beginning and the end of the kernel shows this exponential behavior due to the discrepancy in the power sensor of the NVIDIA Kepler GPUs. They have proposed a formula to correct the power readings where true power at a given time is proportional to

the measure power and slope of the power curve at that point in time, $P_{\text{true}} = CP_{\text{meas}}\frac{dP}{dt}$ where $C$ is a constant. However, we conjecture that this could also be a result of thermal effect on the power consumption of the GPU.

The aforementioned behavior of the NVML library can be further illustrated by executing few kernels back to back and recording the instantaneous power (see Figure 5.2). At the start of the first kernel call we see the gradual increase in power as before. Once it reaches steady state, it retain in the same power until all three kernel calls are finished. At the end, after finishing third kernel call, power decays similar to Figure 5.1. Therefore, if we need to measure the instant power consumption accurately, first, we need to run the kernel multiple times so that it reaches and retains in steady state of power/temperature. We take power reading at least for 40 seconds and we use all the readings after 15 second time stamp to compute the average power. Therefore the final energy consumption of the kernel is measured by maintaining nearly a constant temperature. Each hardware energy parameter is measured under a constant temperature. But actual benchmarks may operate in a different temperature. Using formulas suggested by Hong and Kim [24] we can compute the hardware energy parameters at different operating temperatures of the GPU.

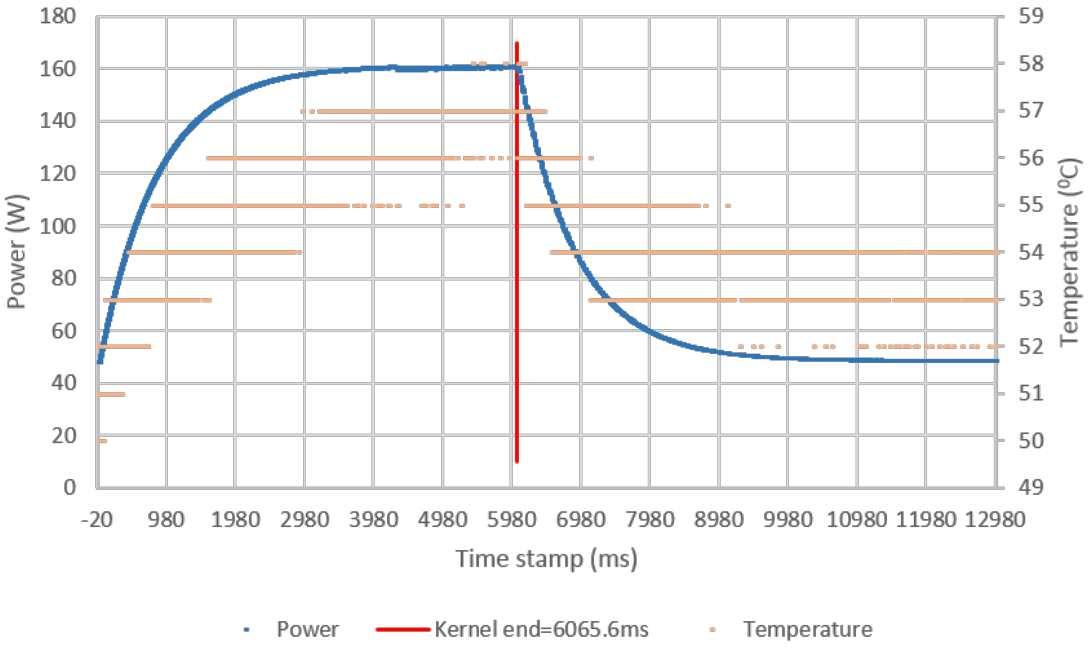


FIGURE 5.1. Instantaneous power curve for cuBLAS matrix multiplication of size 19456. Execution of the kernel starts at time stamp 0ms and ends (red line) at 6065.6ms. During the first ∼3 seconds power consumption increases gradually with an inverse exponential slope. Then, it reaches steady state. Just after the end of kernel, power gradually drops with a decaying exponential slope. Temperature also follows a similar pattern. NVIDIA NVML's temperature readings are in integers, therefore, there are discrete steps in the curve.

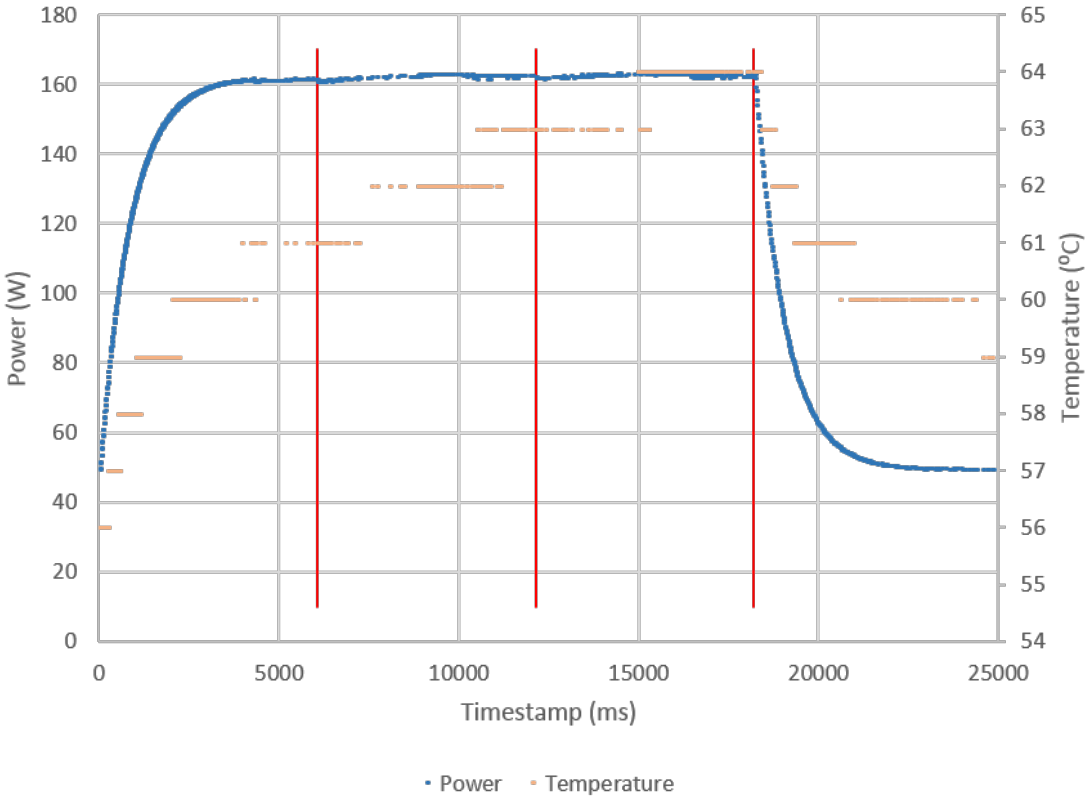


FIGURE 5.2. Instantaneous power curve for 3 consecutive kernels of cuBLAS matrix multiplication of size 19456. Execution of the first kernel starts at time stamp 0ms and ends (red line) at vertical red line. Vertical red lines indicate the end of a kernel call and beginning of the next kernel call. Similar to the Figure 5.1, during the first ∼3 seconds, power consumption increases gradually with an inverse exponential slope. Then, it reaches steady state and retains the steady state until 3 kernel calls are done. Just after the end of 3$^{rd}$ kernel, power gradually drops with a decaying exponential slope. Temperature also follows a similar pattern. NVIDIA NVML's temperature readings are in integers, therefore, there are discrete steps in the curve.

# CHAPTER 6

# RESULTS

In the previous chapter, we discussed about the calibration of hardware energy parameters and measuring energy consumption of GPU kernels experimentally. In this chapter, we present the experimental validation results for the number of off-chip memory accesses for both traditional and our implementation. Finally, we present the experimental results for the energy consumption.

## 6.1. Experimental Setup

Our experiments are run on NVIDIA GTX 480 (Fermi) and K20c (kepler) GPUs. The configuration of GPUs and the environment are provided in Table 6.1. All the experimental results are based on implementations of Smith-Waterman algorithm.

Table 6.1. Configurations of NVIDIA GTX 480 and K20c GPUs

| Parameter Name [unit] | GTX 480 | K20c |
|---|---|---|
| GPU architecture | Fermi | Kepler |
| Manufacturing technology [nm] | 40 | 28 |
| CUDA compute capability | 2.0 | 3.5 |
| Off-chip (global) memory [GB] | 1.5 | 4.68 |
| SMs | 15 | 13 |
| Cores per SM | 32 | 192 |
| Clock rate [MHz] | 1401 | 706 |
| L2 cache [KB] | 768 | 1280 |
| Shared memory [KB] per SM | 48 | 48 |
| Registers per SM | 32768 | 65536 |
| Max. concurrent thread blocks per SM | 8 | 16 |
| Max. registers per thread | 63 | 255 |
| CUDA version | 6.0 | 6.0 |
| CUDA driver | 340.46 | 331.89 |
| Power profiling support | No | Yes |
| gcc version | 4.6.3 | 4.4.7 |
| OS | Fedora 20 (3.16.3-200) | Red Hat 4.4.7-4 |
| CPU | Intel Q9550 | Xeon E5-2620 v2 |
| CPU Clock [GHz] | 2.83 | 2.10 |

## 6.2. Validation of Number of Off-chip Memory Transfers

In sections 3.3 and 3.4, we have analytically showed the amount of off-chip memory accesses for both traditional and our energy efficient implementations. In this section, we experimentally validate the number of off-chip memory accesses predicted by our analytical models. NVIDIA *nvprof* [73] profiler is used to read the hardware performance counters (or events according to *nvprof* terminology) and get the amount of off-chip memory accesses. The target GPU is NVIDIA GTX 480. The Table 6.2 provides the program parameter values used for this experiment.

### 6.2.1. Computing the volume of off-chip memory transfers analytically.

Using the formulas and substituting data from Table 6.2, we get,

$$\text{Volume of off-chip data for traditional approach} = 408GB$$

$$\text{Volume of off-chip data for our approach (WT cache)} = 66GB$$

$$\text{Volume of off-chip data for our approach (WB cache)} = 4GB$$

where WT refers to *write-through* cache policy and WB refers to *write-back* cache policy. As we can see there is a significant reduction (factor of 6) in the global memory data volume, in our implementation. If we can use WB policy, then we will have two orders of magnitude reduction in global memory accesses. This could be an interesting design choice that GPU hardware designers should take into account.

### 6.2.2. Computing the amount of off-chip memory transfers experimentally.

Table 6.4 lists the events and corresponding symbols that we used to find the off-chip and the L2 cache accesses. Table 6.5 reports the experimental values of each counter listed

in Table 6.4 for both implementations. The data volume correspond to off-chip memory transfers are listed below.

$$\text{Number of off-chip memory accesses (32byte)} = R_{\text{OC0}} + R_{\text{OC1}} + W_{\text{OC0}} + W_{\text{OC1}}$$

$$\text{Volume of off-chip data for traditional approach} = 409GB$$

$$\text{Volume of off-chip data for multi-pass (WT cache)} = 70GB$$

$$\text{Volume of off-chip data for our approach (WB cache)} = 7GB$$

If we compare the analytical and the experimental results, they almost equal to each other which validates the claims of reduction in off-chip memory accesses. The volume of off-chip transfers with WB cache is computed by extrapolating the performance counter results for WT cache.

Table 6.5 reports additional information that shows the improvement in savings of off-chip memory accesses. It shows 45 factor of reduction in off-chip memory *reads*. We have also gained 3 factor of reduction in off-chip memory *writes* despite the WT cache policy. This is due to the fact that we are using one thread block to process a complete row of tiles, so that all the data dependencies along the row, are transfered through registers instead of using off-chip memory. The number of L2 cache *write* requests are also reduced by a factor of 3 due to the same reason. There is only a small reduction in L2 cache *read* requests. This is caused by the mechanism we use to maintain the order of wavefronts (see section 3.4.4). During the pipeline fill stage, thread blocks in the first pass, spin on a busy-wait checking a value in off-chip memory which is cached in L2 cache. This busy-wait on data in L2 cache increases the number of L2 cache read requests resulting in small reduction in L2 cache *read* requests. One can reduce the effect of this problem by designing thread interruption

TABLE 6.2. Program parameter values for the memory access validation experiment

| Parameter Name [unit] | Value |
|---|---|
| S | $2^{21}$ |
| T | $2^{21}$ |
| $t_{\mathrm{S}}$ | 512 |
| $t_{\mathrm{T}}$ | 256 |
| $s_{\mathrm{S}}$ | 8 |
| $s_{\mathrm{T}}$ | 1 |

TABLE 6.3. Program parameters specific to our implementation.

| Parameter Name [unit] | Our Implementation |
|---|---|
| Number of rows of tiles | 4096 |
| Threads per thread block | 64 |
| $M_{\mathrm{reg}}^{\mathrm{alg}}$ | 63 |
| $M_{\mathrm{smem}}^{\mathrm{alg}}$ | 5552 |
| $k$ | 8 |
| $P_{\mathrm{h}}$ | 120 |
| $N_{\mathrm{P}}$ | 34 |

techniques, but it will take GPUs more closer toward CPUs and increase the complexity of GPU hardware.

## 6.3. Energy Consumption

In the previous section, we validated the significant savings of off-chip memory transfers. In this section, we present the experimental energy consumption results for the traditional and our implementations. The energy consumption of Smith-Waterman program is measured using the approach discussed in section 5.

Although, we expected the execution time of the kernel to be similar for both implementations, most of the time our implementation shows 22% better performance[1]. Now, let's find out the reasons behind this improvement. When we implement our approach, we were able to avoid one of the shared memory arrays. Therefore, traditional approach consume more

[1]The percentage improvement is averaged over all the data points

TABLE 6.4. Description as it appear in the help command of *nvprof* tool, of profiling events used to validate the amount of off-chip memory and L2 cache accesses. Symbol is used in the formulas of section 6.2

| Event Name | Symbol | Description |
|---|---|---|
| fb_subp0_read_sectors | $R_{\text{OC0}}$ | Number of DRAM read requests to sub partition 0, increments by 1 for 32 byte access. |
| fb_subp1_read_sectors | $R_{\text{OC1}}$ | Number of DRAM read requests to sub partition 1, increments by 1 for 32 byte access. |
| fb_subp0_write_sectors | $W_{\text{OC0}}$ | Number of DRAM write requests to sub partition 0, increments by 1 for 32 byte access. |
| fb_subp1_write_sectors | $W_{\text{OC1}}$ | Number of DRAM write requests to sub partition 1, increments by 1 for 32 byte access. |
| l2_subp0_total_read_sector_queries | $R_{\text{L20}}$ | Total read requests to slice 0 of L2 cache. This includes requests from L1, Texture cache, system memory. This increments by 1 for each 32-byte access. |
| l2_subp1_total_read_sector_queries | $R_{\text{L21}}$ | Total read requests to slice 1 of L2 cache. This includes requests from L1, Texture cache, system memory. This increments by 1 for each 32-byte access. |
| l2_subp0_total_write_sector_queries | $W_{\text{L20}}$ | Total write requests to slice 0 of L2 cache. This includes requests from L1, Texture cache, system memory. This increments by 1 for each 32-byte access. |
| l2_subp1_total_write_sector_queries | $W_{\text{L21}}$ | Total write requests to slice 1 of L2 cache. This includes requests from L1, Texture cache, system memory. This increments by 1 for each 32-byte access. |

shared memory per tile. But, the number of concurrent thread blocks per SM, $k$ depends on the amount of shared memory utilization per thread block. Therefore, almost all the time, $k$ of traditional is less than $k$ of our implementation. This is one of the main reasons behind this improvement.

TABLE 6.5. Profiler values for the events listed in table 6.4 for both traditional and our implementation. The corresponding program parameters are given in the table 6.2

| Counter Symbol | Traditional Implementation | Our Implementation |
|---|---|---|
| $R_{OC0}$ | $3.63\times10^9$ | $8.29\times10^7$ |
| $R_{OC1}$ | $3.64\times10^9$ | $8.03\times10^7$ |
| Total off-chip reads | $7.27\times10^9$ | $1.63\times10^8$ |
| $W_{OC0}$ | $3.24\times10^9$ | $1.09\times10^9$ |
| $W_{OC1}$ | $3.24\times10^9$ | $1.09\times10^9$ |
| Total off-chip writes | $6.47\times10^9$ | $2.18\times10^9$ |
| $R_{L20}$ | $3.71\times10^9$ | $3.36\times10^9$ |
| $R_{L21}$ | $3.71\times10^9$ | $3.45\times10^9$ |
| Total L2 reads | $7.41\times10^9$ | $6.82\times10^9$ |
| $W_{L20}$ | $3.24\times10^9$ | $1.09\times10^9$ |
| $W_{L21}$ | $3.24\times10^9$ | $1.09\times10^9$ |
| Total L2 writes | $6.48\times10^9$ | $2.18\times10^9$ |
| L2 cache read hits % | 2 | 98 |
| L2 cache write hits % | 0 | 0 |
| Total L2 cache hits % | 1 | 74 |

We see average of 13% energy savings in our method. It is obvious since, our implementation performs better compared to the traditional implementation. Even though we reduced off-chip memory accesses significantly, we do not see similar reduction in energy consumption for Smith-Waterman. Actually, for Smith-Waterman, the reduction in off-chip memory accesses does not help in reducing the energy consumption significantly. Let's see why. The first row of the Table 6.6 shows the contribution of different energy components (in Jules and percentage within parentheses ) towards a tile of Smith-Waterman. The energy consumption is dominated by both shred-to-register transfers and computations (max-add). The contribution of off-chip memory transfers is less than 2%. Therefore, even if we avoid all the off-chip memory accesses, we merely get 2% reduction in dynamic energy consumption which is negligible.

TABLE 6.6. Average energy contribution for a tile by different energy components.

| Benchmark | $E_{gs}$ (%) | $E_{sr}$ (%) | $E_{ar}$ (%) |
|---|---|---|---|
| Smith-Waterman [J] | $1.9\times10^{-5}$ (2) | $4.6\times10^{-4}$ (41) | $6.5\times10^{-4}$ (57) |
| Jacobi 2D [J] | $2.5\times10^{-5}$ (21) | $8.1\times10^{-5}$ (67) | $1.5\times10^{-5}$ (12) |

The static power ($p_{stat}$) is another factor that affects the percentage energy savings compared to the total energy consumption. If the contribution of static power is high such that the contribution of dynamic energy becomes small, then again our approach of reducing off-chip memory accesses will not help to reduce overall energy consumption significantly.

In summary, our approach reduces energy consumption significantly, only if the contribution of the energy for off-chip memory transfers are at-least in the same order as other energy components. For example, second row if Table 6.6 shows the contribution of energy by different components for Jacobi 2D 5 point stencil. In this case, the contribution of energy of off-chip memory transfers is 21% when implemented in traditional approach. If we assume 6 factor of reduction in global memory transfers (which was the case for Smith-Waterman in a GPU with L2 cache with WT cache policy, see section 6.2), we will see 17.3% reduction in dynamic energy, and 5% reduction in total energy consumption.

## 6.4. CONCLUSION

We introduced a novel way of implementing wavefront parallel programs to reduce the number of off-chip memory accesses significantly for GPUs. Our validation analysis confirms the significant savings of off-chip memory accesses compared to the traditional implementation. We have experimentally showed that, significant reduction in off-chip memory accesses does not necessarily reduce the off-chip memory accesses significantly. If we are to expect considerable savings in energy after reducing the off-chip memory transfers, then the contribution of energy for off-chip memory transfers in the traditional implementation must be

in the same order as other energy components.The energy savings will be significant for the programs implemented with traditional approach where the contribution of energy from off-chip memory transfers in the same order as other energy components like energy for register to shared memory transfers. Jacobi2D is an example for afore mentioned situation and will be able to save more energy by using our approach of implementation. Off-chip memory accesses and the energy consumption can be further reduced by designing last level cache with *write-back* cache write policy which results in 45 times reduction in off-chip memory accesses compared to a factor of 6 with *write-though* policy.

Even though we can apply this technique to wide class of programs, we have only used Smith-Waterman for the experiments. We can consider more programs to validate the results in our future work. If we are going to use this technique to reduce the total energy consumption, then, we will also need to look at techniques to identify programs where the energy consumption of off-chip memory transfers are in same order as other energy components. A code generator to generate code for our implementation is another task that is in our future work.

# BIBLIOGRAPHY


[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[2] D. McMorrow and M. Corporation, *Technical Challenges of Exascale Computing*. MITRE Corporation, 2013.

[3] TOP500, "TOP500 list presented at ISC'14," June 2014.

[4] D. Anderson, J. Dykes, and E. Riedel, "More than an interface—SCSI vs. ATA," in *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, (Berkeley, CA, USA), pp. 245–257, USENIX Association, 2003.

[5] R. Bianchini and R. Rajamony, "Power and energy management for server systems," *Computer*, vol. 37, pp. 68–76, Nov 2004.

[6] L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, vol. 17, pp. 83–93, Feb. 1974.

[7] K. R. Koch, R. S. Baker, and R. E. Alcouffe, "Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor," *Transactions of the American Nuclear Society*, vol. 65, no. 108, pp. 198–199, 1992.

[8] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, vol. 49. National Bureau of Standards Washington, DC, 1952.

[9] S. Rajopadhye, G. Iooss, T. Yuki, and D. Connors, "The Stencil Processing Unit: GPGPU done right," 2013.

[10] N. Prajapati, W. Ranasinghe, V. Tandrapati, R. Andonov, H. Djidjev, and S. Rajopadhye, "Energy modeling and optimization for GPU stencil computations," in *Manuscript submitted for publication*, 2014.

[11] S. Xiao, A. Aji, and W. chun Feng, "On the robust mapping of dynamic programming onto a graphics processing unit," in *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pp. 26–33, Dec 2009.

[12] S. Xiao and W. chun Feng, "Inter-block GPU communication via fast barrier synchronization," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, April 2010.

[13] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving GPU energy efficiency," *CoRR*, vol. abs/1404.4629, 2014.

[14] H. Anzt, V. Heuveline, J. Aliaga, M. Castillo, J. Fernandez, R. Mayo, and E. Quintana-Orti, "Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms," in *Green Computing Conference and Workshops (IGCC), 2011 International*, pp. 1–6, July 2011.

[15] Y. Jiao, H. Lin, P. Balaji, and W. Feng, "Power and performance characterization of computational kernels on the GPU," in *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pp. 221–228, Dec 2010.

[16] Y. Lin, T. Tang, and G. Wang, "Power optimization for GPU programs based on software prefetching," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pp. 1339–1346, Nov 2011.

[17] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures," in *Parallel Processing (ICPP), 2012 41st International Conference on*, pp. 48–57, Sept 2012.

[18] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M. Sarrafzadeh, "Energy-aware high performance computing with graphic processing units," in *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2008.

[19] Y. Wang, S. Roy, and N. Ranganathan, "Run-time power-gating in caches of GPUs for leakage energy savings," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pp. 300–303, March 2012.

[20] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 235–246, ACM, 2011.

[21] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 96–106, IEEE Computer Society, 2012.

[22] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient GPU architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 86–98, ACM, 2013.

[23] S. Gilani, N. S. Kim, and M. Schulte, "Power-efficient computing for compute-intensive GPGPU applications," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 330–341, Feb 2013.

[24] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 280–289, ACM, 2010.

[25] P.-H. Wang, C.-L. Yang, Y.-M. Chen, and Y.-J. Cheng, "Power gating strategies on GPUs," *ACM Trans. Archit. Code Optim.*, vol. 8, pp. 13:1–13:25, Oct. 2011.

[26] Y. Wang and N. Ranganathan, "An instruction-level energy estimation and optimization methodology for GPU," in *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pp. 621–628, Aug 2011.

[27] S. Song, M. Lee, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Energy-efficient scheduling for memory-intensive GPGPU workloads," in *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, (3001 Leuven, Belgium, Belgium), pp. 19:1–19:6, European Design and Automation Association, 2014.

[28] Y. Jararweh and S. Hariri, "Power and performance management of GPUs based cluster," *Int. J. Cloud Appl. Comput.*, vol. 2, pp. 16–31, Oct. 2012.

[29] H. Wang and Q. Chen, "Optimization power consumption model of reliability-aware GPU clusters," *The Journal of Supercomputing*, vol. 67, no. 1, pp. 153–174, 2014.

[30] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded GPU," in *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pp. 344–350, Dec 2010.

[31] P. Alonso, M. Dolz, F. Igual, R. Mayo, and E. Quintana-Orti, "Reducing energy consumption of dense linear algebra operations on hybrid CPU-GPU platforms," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pp. 56–62, July 2012.

[32] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "Fixing performance bugs: An empirical study of open-source GPGPU programs," in *Proceedings of the 2012 41st International Conference on Parallel Processing*, ICPP '12, (Washington, DC, USA), pp. 329–339, IEEE Computer Society, 2012.

[33] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235–246, March 2010.

[34] NVIDIA Corporation, *CUDA C Programming Guide v6.0*, Apr. 2014. v6.0.

[35] R. Andonov and S. Rajopadhye, "Optimal orthogonal tiling of 2-D iterations," *Journal of Parallel and Distributed Computing*, vol. 45, pp. 159–165, September 1997.

[36] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 40:1–40:11, IEEE Computer Society Press, 2012.

[37] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for GPUs," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 66:66–66:75, ACM, 2014.

[38] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, (New York, NY, USA), pp. 30–44, ACM, 1991.

[39] J. McCalpin and D. Wonnacott, "Time skewing: A value-based approach to optimizing for memory locality," tech. rep., Technical Report DCS-TR-379, Department of Computer Science, Rugers University, 1999.

[40] D. Wonnacott, "Achieving scalable locality with time skewing," *Int. J. Parallel Program.*, vol. 30, pp. 181–221, June 2002.

[41] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981.

[42] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705 – 708, 1982.

[43] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 93–, IEEE Computer Society, 2003.

[44] G. Contreras and M. Martonosi, "Power prediction for Intel XScale reg; processors using performance monitoring unit events," in *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, pp. 221–226, Aug 2005.

[45] G. QU, N. KAWABE, K. USAMI, and M. POTKONJAK, "Code coverage-based power estimation techniques for microprocessors," *Journal of Circuits, Systems and Computers*, vol. 11, no. 05, pp. 557–574, 2002.

[46] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 83–94, June 2000.

[47] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects," tech. rep., 2003.

[48] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Transactions on Architecture and Code Optimization*, vol. 1, pp. 94–125, 2004.

[49] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469–480, Dec 2009.

[50] M. Kamble and K. Ghose, "Analytical energy dissipation models for low power caches," in *Low Power Electronics and Design, 1997. Proceedings., 1997 International Symposium on*, pp. 143–148, Aug 1997.

[51] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," tech. rep., Technical Report 2001/2, Compaq Computer Corporation, 2001.

[52] S. J. E. Wilton and N. Jouppi, "Cacti: an enhanced cache access and cycle time model," *Solid-State Circuits, IEEE Journal of*, vol. 31, pp. 677–688, May 1996.

[53] Y. S. Shao and D. Brooks, "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, (Piscataway, NJ, USA), pp. 389–394, IEEE Press, 2013.

[54] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 152–163, ACM, 2009.

[55] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of GPU kernels using performance counters," in *Green Computing Conference, 2010 International*, pp. 115–122, Aug 2010.

[56] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for GPU architectures using McPAT," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 19, pp. 26:1–26:24, June 2014.

[57] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 487–498, ACM, 2013.

[58] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 97–106, Apr 2013.

[59] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163–174, April 2009.

[60] J.-C. Huang, L. Nai, H. Kim, and H.-H. Lee, "TBPoint: Reducing simulation time for large-scale GPGPU kernels," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 437–446, May 2014.

[61] D. Q. Ren, "Algorithm level power efficiency optimization for CPU-GPU processing element in data intensive SIMD/SPMD computing," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 245 – 253, 2011. Data Intensive Computing.

[62] D. Q. Ren and R. Suda, "Global optimization model on power efficiency of GPU and multicore processing element for SIMD computing with CUDA," *Computer Science - Research and Development*, vol. 27, no. 4, pp. 319–327, 2012.

[63] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for GPUs," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 66:66–66:75, ACM, 2014.

[64] D. Hains, Z. Cashero, M. Ottenberg, W. Bohm, and S. Rajopadhye, "Improving CUDASW++, a parallelization of Smith-Waterman for CUDA enabled devices," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, (Washington, DC, USA), pp. 490–501, IEEE Computer Society, 2011.

[65] S. Yan, G. Long, and Y. Zhang, "Streamscan: Fast scan algorithms for GPUs without global barrier synchronization," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 229–238, ACM, 2013.

[66] NVIDIA Corporation, *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012. v1.0.

[67] NVIDIA Corporation, *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009. v1.0.

[68] NVIDIA Corporation, *Parallel Thread Execution ISA*, Aug. 2014. v6.5.

[69] NVIDIA Corporation, *NVML Reference Manual*, Mar. 2014. vR331.

[70] J. Lang and G. Rünger, "High-resolution power profiling of GPU functions using low-resolution measurement," in *Euro-Par 2013 Parallel Processing*, pp. 801–812, Springer, 2013.

[71] NVIDIA Corporation, *cuBLAS*, Aug. 2014. v6.5.

[72] M. Burtscher, I. Zecena, and Z. Zong, "Measuring GPU power with the K20 built-in sensor," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, (New York, NY, USA), pp. 28:28–28:36, ACM, 2014.

[73] NVIDIA Corporation, *Profiler User's Guide*, Aug. 2014. v6.5.

# APPENDIX A

# CUDA IMPLEMENTATION

## A.1. Synchronization patterns for wavefront parallelization

The popular barrier synchronization code patterns do not work with CUDA syncthreads function. For example, code Listing A.1 either deadlock or produce incorrect results.

Most of us confuse syncthreads() in CUDA as a regular barrier for threads in a thread block. But this is not a regular barrier of threads, because according to NVIDIA programming guide [34], all the threads should reach the same syntactic syncthreads() function in-order to produce correct results (to function as a barrier). Otherwise the output is undefined. This is also can be explained as there cannot be any syncthreads() inside branched code unless all the threads evaluate to the same branch.

Listing A.1. Synchronization pattern for wavefronts with pipeline filling and flush

```
for (t = 0; t < tId; t++) {
  syncthreads();
}

for (t = tId; t < tId + SUBTILES; t++) {
  int x = t - tId;
  for (s = 0; s < SUBTILE_WIDTH; s++) {
      //process sth column of sub-tile;
  }
  syncthreads();
}

for (t = tId; t < THREADS-1; t++) {
  syncthreads();
}
```

## A.2. Explore Memory Space

In this section, we discuss the use of memory hierarchy for different input and temporary variables used in the Smith Waterman kernel. All the input data (two DNA sequences) initially reside in global memory.

(1) Global Memory

Global memory is used to initially store the inputs, two DNA sequences. Even though DNA sequences supposed to be a sequence of alphabet A, C, G, T, in the host the character sequence is encoded with 0, 1, 2, 3 casted to 8bit length integers. This enables to access the substitution matrix without using any conditional statements. Therefore, it reduces the branching and increase the performance.

(2) Shared Memory

Shared memory is used to store an array of length $x$ of DNA sequence aligned to horizontal axis. Since this array of data is accessed by all the threads in a thread block, shared memory is chosen.

There is another level of tiling (level 2 tiles) within the first level of tiles. Each thread in a thread block responsible for a row of second level of tiles. The last row of each second level tile is read by a different thread other than the thread who produce this data. Therefore the same set of data is accessed by more than one thread. Hence the last row correspond to a row of second level tiles is stored in shared memory.

Substitution matrix is read only. By intuition, we tend to use constant memory to store substitution matrix. But, as described in section 2.1, to use constant memory with optimal performance, all the threads in a warp should access the same address in the constant memory. But in the case of substitution matrix, the

threads in a warp may access different addresses (depending on the sequence data). Therefore, shared memory is a better place for substitution matrix.

(3) Register (local) Memory

The portion of the array (length of $y$) of DNA sequence aligned to vertical axis is stored in registers. For a row of tiles, each thread access distinct portion of the array. Therefore, each thread reuse the same set of data throughout a row of tiles. Each thread maintain a column of data correspond to the table of Smith Waterman in registers. A thread progress on processing the sub-table column by column. Therefore it's enough to maintain a column of height equal to the height of sub-tile amount of data in registers. An extra register is used to store the element read from the sub-tile tile above.

(4) Constant Memory

Performance of accessing constant memory is optimal when all the threads in a warp access the same address in the constant memory. Therefore, gap penalty and gap extension penalty is stored in constant memory. Even though the substitution matrix is read only, we cannot use constant memory as described in section 2.1.

(5) L1 Cache

L1 cache in GPUs are not coherent within the same kernel call. Since we use only one kernel call and data written by a streaming multiprocessor is read by another streaming multiprocessor, the later processor may read outdated data from global memory or from L2 cache. There are few resolutions for this issue.

(a) Disable L1 cache using compiler option *-dlcm=cg*. Therefore, all the memory requests are served by L2 cache which is coherent.

(b) Use special ptx instructions within the CUDA kernel to by pass L1 cache, using *asm* instruction. This option is almost equal to (a), except that L1 cache can be used if the data is only accessed by a single streaming multiprocessor.

(c) Use memory fence functions available in CUDA to flush L1 cache. Therefore, the coherency of L1 cache can be enforced.

In this implementation, the option (a) is used as the resolution. Investigation on the other two options are left for the future work.

(6) L2 Cache

We select the tile sizes such that the volume of data accessed by $P$ rows of tiles fits in the L2 cache.

(7) Texture Memory

As explained in section 2.1 , texture memory is optimized for 2D spatial locality. Therefore, substitution matrix is a good candidate for texture memory. Due to the variety of ways we can use texture memory, we have not explored the texture memory within the scope of this paper and have left it for future work.