THESIS

EXTENDING AND VALIDATING THE STENCIL PROCESSING UNIT

Submitted by

Revathy Rajasree

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2016

Master's Committee:

Advisor: Sanjay Rajopadhye

Sudeep Pasricha
Charles W. Anderson

ABSTRACT

EXTENDING AND VALIDATING THE STENCIL PROCESSING UNIT

Stencils are an important class of programs that appear in the core of many scientific and general-purpose applications. These compute-intensive kernels can benefit heavily from the massive compute power of accelerators like the GPGPU. However, due to the absence of any form of on-chip communication between the coarse-grain processors on a GPU, any data transfer/synchronization between the dependent tiles in stencil computations has to happen through the off-chip (global) memory, which is quite energy-expensive. In the road to exascale computing, energy is becoming an important cost metric. The need for hardware and software that can collaboratively work towards reducing energy consumption of a system is becoming more and more important.

To make the execution of dense stencils more energy efficient, Rajopadhye et al. [10] proposed the GPGPU-based accelerator called Stencil Processing Unit that introduces a simple neighbor-to-neighbor communication between the Streaming Multiprocessors (SM) on the GPU, thereby allowing some restricted data sharing between consecutive threadblocks. The SPU includes special storage units, called *Communication Buffers*, to orchestrate this data transfer and also provides an explicit mechanism for inter-threadblock synchronization by way of a special instruction. It claims to achieve energy-efficiency, compared to GPUs, by reducing the number of off-chip accesses in stencils which in turn reduces the dynamic energy overhead. Uguen [13] developed a cycle-accurate performance simulator for the SPU, called SPU-Sim, and evaluated it using a matrix multiplication kernel which was not suitable for this accelerator. This work focuses on extending the SPU-Sim and evaluating the SPU architecture using a more insightful benchmark.

We introduce a producer-consumer based inter-block synchronization approach on the SPU, which is more efficient than the previous global synchronization, and an overlapped multi-pass execution model in the SPU runtime system. These optimizations have been implemented into SPU-Sim. Furthermore, the existing GPUWattch power model in the simulator has been refined to provide better power estimates for the SPU architecture. The improved architecture has been evaluated using a simple 2-D stencil benchmark and we observe an average of 16% savings in dynamic energy on SPU compared to a fairly close GPU platform. Nonetheless, the total energy consumption on SPU is still comparatively high due to the static energy component. This high static energy on SPU is a direct impact of the increased leakage power of the platform resulting from the inclusion of special load/store units. Our conservative estimates indicate that replacing the current design of these L/S units with DMA engines can bring about a 15% decrease in the current leakage power of the SPU and this can help SPU outperform GPU in terms of energy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Energy efficiency has become an important metric both in High Performance Computing as well as embedded systems. It has been identified as one of the top 10 challenges towards achieving exascale computing by DoE's Subcommitte on Exascale Computing [12]. Improving energy efficiency is critical for accomplishing the DoE mission of 1 exaflop within a 20 MW power budget. This work focuses on exploring an energy-efficient mechanism for executing an important class of programs called *stencils*. Such computations occur in the core of numerous scientific as well as general-purpose applications which include PDE solvers, image processing, particle simulations, etc. They have even found a place among the Berkeley dwarfs [2]. The key characteristic of a stencil is that it involves computing a multi-dimensional grid of data points, where each point is computed as a function of a subset of its neighbors. This implies that there are dependences between data points which may impose constraints on when a point can be computed. Due to the broad applicability of stencil computations, accelerating these can be beneficial in many respects.

1.1 PROBLEM WITH STENCILS ON GPGPU

Recently, Graphics Processing Units (GPUs) have been gaining popularity as accelerators for general-purpose applications. Their increasing use led to the invention of the CUDA programming model (by Nvidia) that is specifically targeted at aiding general-purpose programming on GPUs. Compute-intensive kernels like stencils can benefit heavily from the massive compute power of these accelerators. However, the dependence between computations, that exist in stencils, introduce some problems on GPGPUs. Tiling is an important transformation applied to stencils in order to exploit data locality and parallelism, and

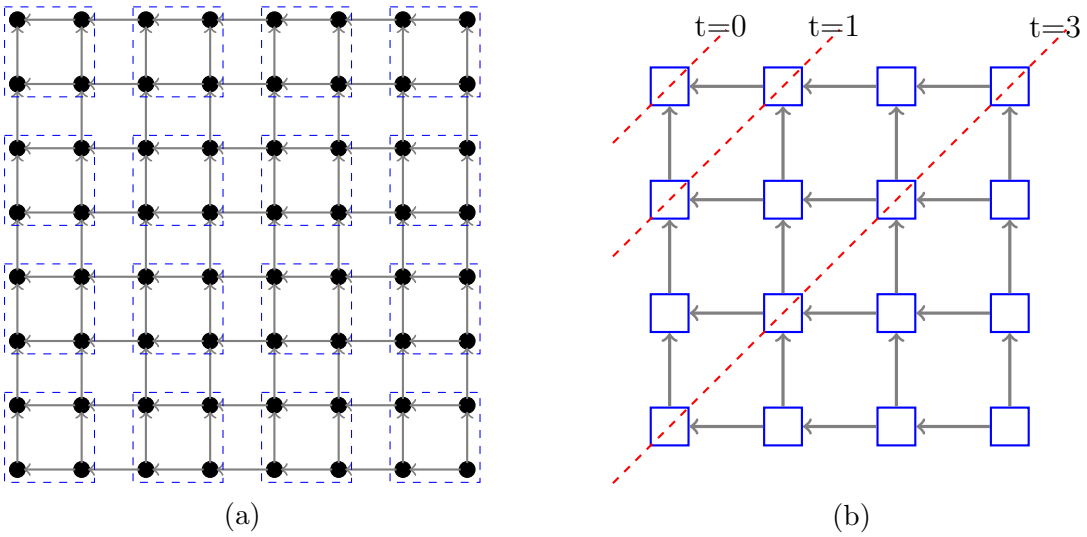


FIGURE 1.1: (a):Dependence graph of data points in a stencil. The nodes indicate data points and the dashed blue squares represent tiles. (b): The tiled dependence graph, with the 45° wavefronts indicating the timestep at which a tile can be computed (all wavefronts are not shown to avoid clutter).

thereby achieve better performance. Consider a stencil computation with dependences as shown by the arrows in Figure 1.1(a). If this computation is tiled as indicated by the dashed blue rectangles, then we can see that the dependences now transform into dependences between adjacent tiles. In order to satisfy these dependences, the tiles need to be executed in a wavefront pattern as shown in Figure 1.1(b). Parallel implementation of most stencils require wavefront scheduling of tiles. In the CUDA implementation of such stencils, a threadblock may be responsible for computing one or more such tiles; hence, the data dependence between these tiles essentially translate into dependences between threadblocks, and these blocks will need to synchronize with each other to enforce the wavefront schedule of tile computation. However, the GPU architecture/CUDA model considers threadblocks as independent blocks of work executing on a single multiprocessor and does not support any explicit mechanism to communicate between them except via global memory and in successive kernel calls. Consequently, any data sharing/synchronization between them needs to use off-chip resources. Although off-chip memory accesses may not have any impact on performance for compute-bound kernels, they are very expensive in terms of energy.

1.2 THE SPU APPROACH

The Stencil Processing Unit (SPU) is a GPGPU-based accelerator for dense stencil computations, proposed by Rajopadhye et al. [10]. Its main goal is to execute such computations more energy-efficiently without degrading performance. The key idea behind SPU is to enable on-chip communication between neighboring processors on a GPU, by introducing some simple innovations in the architecture. Since the dependences in stencils are always between adjacent tiles, an on-chip data transfer mechanism can help reduce the number of off-chip memory accesses in the GPU implementation of stencils and thereby, decrease the dynamic energy overhead associated with their execution. In their proposal, the authors show analytically that $\sqrt{P}$-fold reduction in off-chip accesses is possible with such on-chip communication support, where P is the number of processors on the platform. More details about the architecture of the Stencil Processing Unit is described in Chapter 2. This thesis evaluates the SPU architecture using a stencil benchmark to identify different bottlenecks and optimizes the initially proposed SPU architecture in order to accomplish its goal of energy-efficiency.

1.3 CONTRIBUTIONS

The main contributions of this work are:

1. A comprehensive evaluation of the SPU architecture using a more insightful 2-D stencil benchmark (prior work [13] used a matrix multiplication kernel for this purpose, which was unrealistic).
2. An optimized inter-block synchronization mechanism that eliminates the redundant I/O mentioned previously.
3. A load-balanced and more energy-efficient multi-pass execution model for the SPU runtime system, implemented on the SPU simulator (SPU-Sim).
4. A refined power model for the SPU architecture incorporated into the simulator to

estimate both dynamic and static power consumption.

5. We also provide an analytical estimate of the energy gain if we used specialized spill/restore DMA engines.

1.4 RELATED WORK

Many prior work propose techniques to achieve synchronization/communication between threadblocks on the GPU. Most of them involve using the global/off-chip memory in one way or the other. Xiao et al. [14] have proposed three techniques to achieve inter-block communication, two of which use global memory atomics and the third one uses a lock-free global memory array in order to avoid the overhead of atomic operations. Ranasinghe [11] has also developed a producer-consumer synchronization technique between threadblocks which uses a global memory array to record the status of every threadblock. This was inspired by the idea in the original SPU proposal. The Peer-SM synchronization on GPUs, for wavefront parallel programs, proposed by Belviranli et al. [4] is also similar to, but less efficient than Ranasinghe's technique.

Improving energy-efficiency of stencils by reducing off-chip accesses has been addressed in various prior works. Zou and Rajopadhye [15] have developed a programming technique for CPU platforms that employs *Flattened Multi Pass* strategy for energy efficient parallelization of compute-bound stencils which is very similar to our overlapped pass execution model on SPU. The aforementioned technique from Ranasinghe [11] also aims to reduce the total number of off-chip accesses for wavefront parallel programs on GPUs and is found to be more energy-efficient than the traditional approach, which uses multiple kernel calls to synchronize between wavefronts. Therefore, in our evaluation, we use Ranasinghe's method as our baseline GPU implementation for comparison with the SPU. A survey, by Mittal and Vetter [8], on the techniques employed for improving energy efficiency on GPU identified that these can be broadly classified into following five categories:

1. DVFS-based
2. CPU-GPU workload division-based
3. Architectural based
4. Dynamic Resource Allocation-based, and
5. Application-specific and programming-level techniques

The previously noted methods used by Belviranli et al. and Ranasinghe fit into the programming-level technique class, while the SPU approach better aligns with architectural-based class. However, the examples listed by Mittal and Vetter, for the architecture-based category, mainly involve modifying existing components in the GPU architecture to save energy. But, SPU is slightly different from those methods in the sense that it introduces certain new elements into the existing architecture rather than modifying it. Another architecture-based approach, recently put forth by Braak and Corporaal, called the R-GPU is also a GPGPU-based accelerator. It mainly targets programs with limited instruction-level parallelism due to data dependences and stencils also fall under their targeted class. The R-GPU approach involves replacing the fine-grained SIMT execution model, within the SMs of a GPU, with a pipelined execution of dependent instructions on separate CUDA cores. However, SPU does not make any alterations to the microarchitecture of the SMs on the GPU. Besides, R-GPU is expected to provide gains only for memory bound kernels while SPU mainly targets compute-bound stencil kernels.

CHAPTER 2

BACKGROUND

This chapter provides an overview of GPGPU architecture, the CUDA programming model and the SPU architecture proposed in [10]. We have included only details that need to be understood in order to follow the remaining chapters in this document. More comprehensive explanation can be obtained by consulting the cited references.

2.1 GPGPU ARCHITECTURE AND PROGRAMMING MODEL

In this work, we will be closely following the Nvidia Fermi [9] architecture for GPU Computing. This GPGPU architecture consists of a set of coarse-grain processors, known as Streaming Multiprocessors (SM), which are arranged in a 2-D grid. Inside each SM, there are 32 compute cores (referred to as CUDA cores) capable of integer/floating point operations, 16 load/store units, 4 special function units (SFU) and other structures for instruction issue. The GPU follows a Single Instruction Multiple Thread (SIMT) execution model within each SM, while at the coarse-grain level each SM works independently, executing a set of fine-grain threads. The threads within an SM can collaborate with each other using the shared memory (located inside an SM) or using the thread-level barrier instruction provided by CUDA. The GPU also has a large off-chip memory, also called the *global memory*, which can be used to share data between the host processor and the GPU device or between the SMs within a GPU. The Fermi architecture also includes a last-level L2 cache shared among all SMs and a small L1 cache which is private to each SM. Figure 2.1 shows a coarse-grained outline of the GPU architecture. There are also other kinds of on-chip memory residing within each SM such as the constant cache and the texture cache.

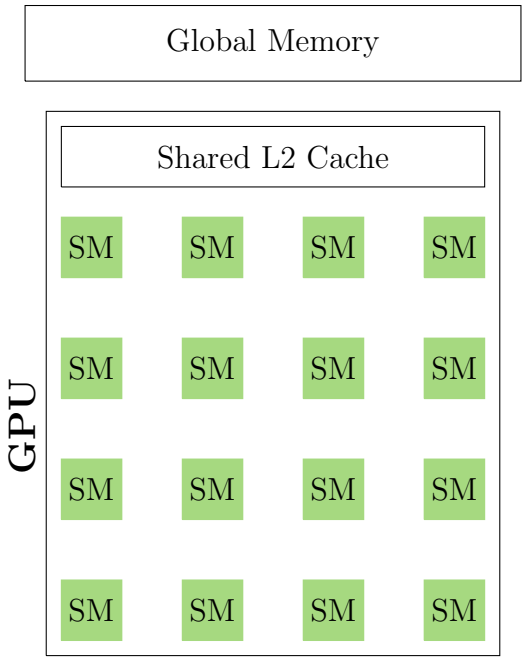


FIGURE 2.1: GPGPU Architecture

The CUDA programming model, used for Nvidia GPGPUs, employs two levels of parallelism just like the architecture. Multiple threads are grouped into coarse-grain units called *threadblocks*, and each threadblock runs on a single SM. There maybe multiple threadblocks running concurrently on one SM, depending on the resource utilization of each block. The threads within each block is grouped into units called warps. A warp consists of 32 threads and forms the fundamental unit of dispatch within an SM. Like mentioned before, CUDA supports two ways to communicate between threads within a threadblock. However, it considers the threadblocks to be independent blocks of work so that the CUDA runtime system can schedule any block on any SM without any ordering constraints. Consequently, it does not provide any explicit mechanism to communicate between the threadblocks/SMs; hence, any such communication has to go through the global memory. The availability of shared L2 cache helps reduce the latency associated with such communication (note that the L1 cache is private and incoherent; hence, not useful for sharing data between SMs). The part of a CUDA program which runs on the GPU is generally referred to as a *kernel*, and the number of threadblocks and threads per block launched by a kernel is chosen by the programmer.

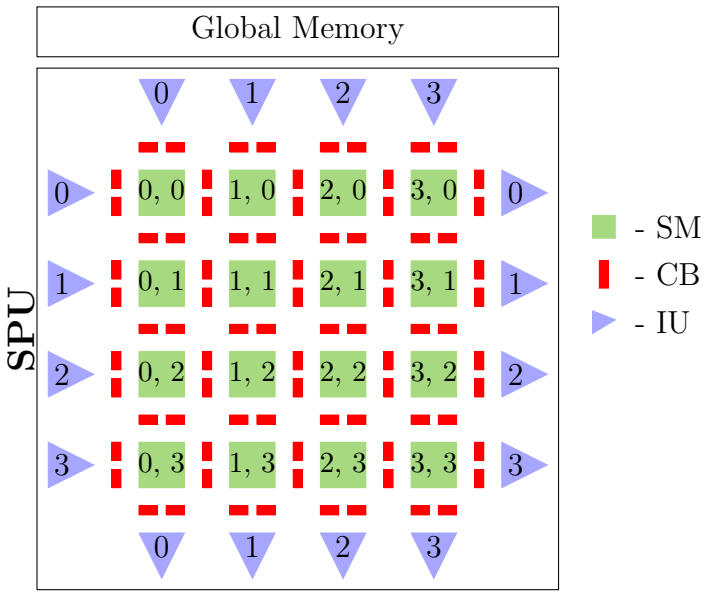


FIGURE 2.2: Stencil Processing Unit Architecture

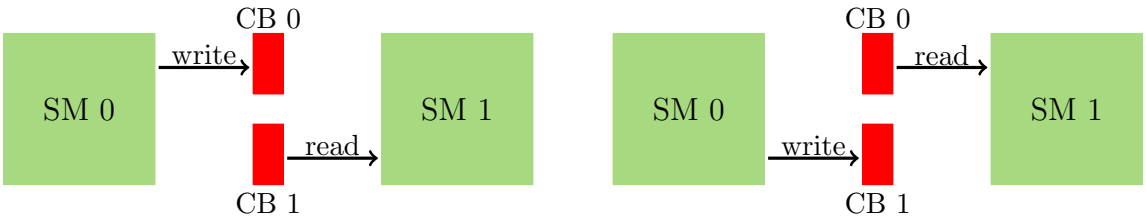
2.2 THE STENCIL PROCESSING UNIT

Due to the absence of any form of on-chip communication between the coarse-grain processors on a GPU, any data sharing/synchronization between the dependent tiles in stencil computations has to happen through the off-chip memory or L2 cache which has been found to be quite energy-expensive [11]. Rajopadhye et al. [10] proposed the Stencil Processing Unit (SPU) architecture for energy-efficient execution of dense stencil computations by enabling a simple form of local communication support on GPGPUs.

2.2.1 Architecture

The SPU architecture introduces the following three elements to allow data transfer between neighboring SMs:

1. it exposes the 2-D grid topology of SMs on the device and restricts the threadblock-to-SM mapping,
2. includes a pair of small memory structures (called Communication Buffers) between neighboring SMs to act as the medium of storage for the data to be shared, and
3. supports an explicit way to synchronize between SMs using a *blocksync* instruction.



(a) Before *blocksync* (b) After *blocksync*

FIGURE 2.3: Data transfer between SMs by switching of CBs at *blocksync*

Figure 2.2 shows a high level overview of the SPU architecture. The coordinates within each green block represent the (x, y) coordinates of the respective SM. The red blocks in this figure represent the communication buffers (CB). The *blocksync* instruction inserts a global barrier that stalls any threadblock from advancing until all other active blocks reach this barrier. The pair of CBs together with *blocksync* instruction use a ping-pong technique to transfer data between the multiprocessors, as shown in Figure 2.3. Say, before reaching a *blocksync* barrier, SM 0 is writing to CB 0 while SM 1 is reading from CB 1 (as shown in Figure 2.3(a)). When this barrier is released, the runtime system switches the CBs accessed by each SM. That is, now SM 0 will be writing the next tile of data to CB 1 while SM 1 can consume the previously written tile in CB 0 (shown in Figure 2.3(b)). Thus, data from SM 0 gets transferred to SM 1. To keep the programming simple, currently SPU allows this kind of data propagation between SMs only from west to east along the horizontal axis and north to south along the vertical axis. Besides supporting this on-chip communication between SMs, the SPU also includes special load/store units around the SPU processor grid in order improve the overlap of computation with communication. These units are called the Interface Units (IU) and are represented by the blue triangles in Figure 2.2. The IUs are responsible for loading/storing to/from the CBs along the boundary of the grid with tiles of data from off-chip memory and they also participate in the global *blocksync* barrier. Due to the restriction imposed on the direction of data propagation, the north and west IUs act only as data loaders while the south and east IUs perform only data stores.

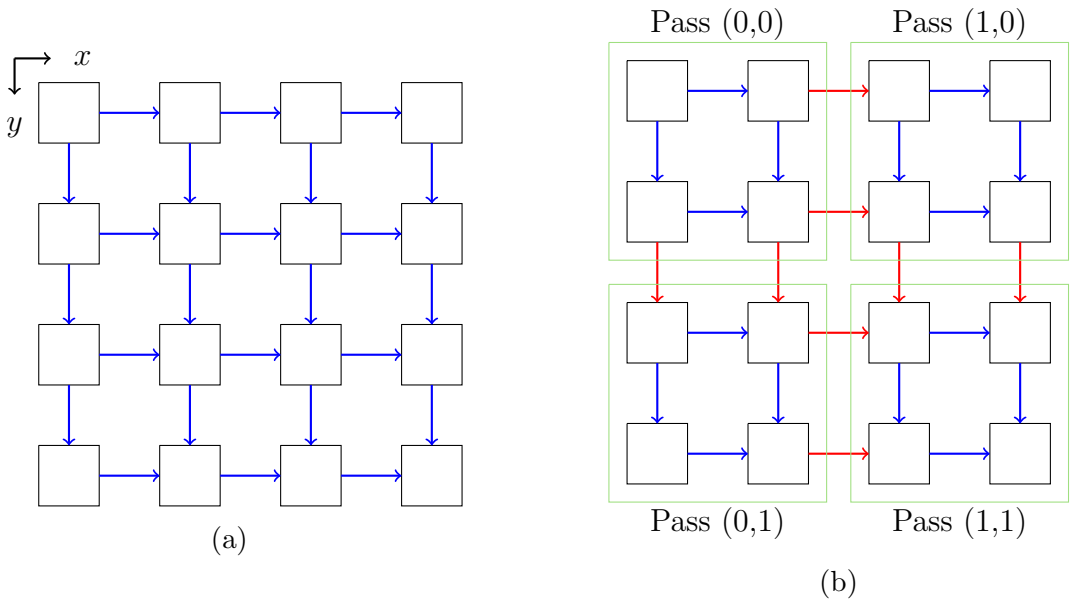


FIGURE 2.4: (a):4×4 grid of threadblocks launched by SPU kernel with dataflow along the direction indicated by the arrows. (b):Division of the threadblocks into passes that fit a 2×2 SPU grid. The blue arrows indicate data propagation through CBs, red arrows represent data that needs to be transferred across passes.

2.2.2 Programming Model & Abstraction

Basically, the SPU uses the CUDA programming model but the programmer is responsible for including the code for the IUs and SMs in the kernel to take full advantage of the SPU architecture. Some special registers have been exposed to the programmer for this purpose (explained in detail by Uguen [13]). Due to the dependences that exist in stencil computations, they always require a wavefront scheduling of tiles. The tiles within a wavefront can be executed in parallel but, in order to satisfy the dependences, there has to be a synchronization barrier between adjacent wavefronts. On the SPU, the global *blocksync* barrier can be used for this purpose. However, the programmer has to take care that all threadblocks participate in this barrier to avoid deadlock.

Like in CUDA, a user may deploy a large number of threadblocks for a SPU kernel as well. But, since the SPU's underlying physical grid has a fixed size, the large grid of threadblocks launched by the user is divided into smaller grids, called *passes*, that fit the actual physical

grid. Thus, the SPU runtime system executes the threadblocks in a pass-by-pass manner. For instance, consider a kernel that is launched with a 4×4 grid of threadblocks, as shown in Figure 2.4(a), with dataflow as indicated by the arrows. Suppose the SPU has only a 2×2 grid of processors, the thread blocks are divided into 4 passes of size 2×2 as shown in Figure 2.4(b). Thus, now we have a grid of passes and the runtime system follows a row-wise ordering of these passes on the actual grid. The capability to launch a large grid of threadblocks is a virtualization provided to the programmer by the SPU runtime system and the actual pass-by-pass execution model is completely hidden from the programmer. This makes the job of the programmer easier and also allows the SPU kernel to be portable from one platform to another. In Figure 2.4(b), the blue arrows indicate data propagation between threadblocks through the CBs. The red arrows indicate the data that needs to be passed on from one pass to another. If the data produced by one pass is needed by a later pass, it is stored to a pre-allocated region in the global memory. This process is referred to as *spilling*. And, when the consumer pass is scheduled, this data is loaded into the CBs from the global memory; this is referred to as *restoring*. The spilling and restoring of data between passes is taken care of by the runtime system. In the figure, all red arrows are satisfied using this spill/restore mechanism. In order to orchestrate spill/restore between passes, the runtime system sets the IUs at those borders to act as what are called *Virtual IUs* (VIU) and it inserts special code, for the spill/restore loads and stores by the VIUs, into the SPU kernel at runtime. Thus, the IUs at the east and south act as spilling VIUs during Pass (0,0) while those at the west and north act as restoring VIUs during Pass (1,0) and Pass(0,1) respectively. The existence of VIUs are also not visible to the programmer. More details about the management of VIUs and the VIU-specific code inserted by the runtime are discussed in [13].

CHAPTER 3

A STENCIL BENCHMARK FOR SPU

In the prior work on SPU and its simulator by Uguen [13], the platform was evaluated using a matrix multiplication (MM) benchmark. But, the MM benchmark is not a suitable application for the SPU as it does not involve dependences and scheduling constraints existent in real stencil computations. Therefore, the experiments with MM only served to functionally verify the simulator, but did not help build insightful inferences about the SPU architecture. In this thesis, we will be using a 2-D stencil benchmark to evaluate the architecture. This chapter provides description of this benchmark and analytical prediction of the reduction in off-chip memory footprint achievable on SPU for this stencil.

3.1 A SIMPLE 2-D STENCIL DEFINITION

Consider a Smith-Waterman like simple 2-dimensional stencil with dependences along the 3 canonic directions implying that the iteration space is 3-D. Each point in the 3-dimensional iteration space is computed as the arithmetic mean of its three canonic neighbors. For any point (i,j,k) in the iteration space, the computation is defined as:

$$H[i,j,k] = Mean(H[i-1,j,k], H[i,j-1,k], H[i,j,k-1]) \tag{3.1}$$

where, $0 \leq i,j,k < N$. At the boundaries of this cubic space, the dependences are on the $N \times N$ input matrices. That is, for $k = 0$: $H[i,j,k-1] = A[i,j]$, for $j = 0$: $H[i,j-1,k] = B[i,k]$, and for $i = 0$: $H[i-1,j,k] = C[j,k]$. A, B and C are the input matrices. $H[i,j,N-1]$ is considered as the primary output matrix.

3.1.1 A Lower bound on Memory Accesses

This 2-D stencil has 3 input matrices and 3 output matrices. Let the problem size parameter be N. For simplicity, lets consider square problem sizes. Then, we have three $N \times N$ inputs and three $N \times N$ outputs. Hence, irrespective of the target architecture, the inevitable input/output (I/O) for this problem is $6N^2$. By I/O we mean the total number of load/store instructions in the program. Hence, $6N^2$ is a lower bound on the total off-chip memory accesses for this simple 2-D stencil problem.

3.2 ANALYSIS OF 2-D STENCIL ON GPU

Using standard tiling, let the N^3 iteration space of our stencil be divided into cubic tiles of size $b \times b \times b$. Consider a CUDA implementation with a 2-dimensional grid of thread blocks. Then, each thread block is responsible for a sequence tiles (say, along the k-axis).

$$\#\text{ tiles executed by a single CTA} = \frac{N}{b}$$

$$\text{Memory footprint per tile} = 3b^2 + 3b^2 = 6b^2$$

However, since the tiles along the k-axis are computed by the same thread block, the data shared between these tiles can reside within the shared memory (by choosing appropriate tile size) and hence, need not be fetched from global memory, thereby avoiding b^2 reads and b^2 writes. Hence,

$$\text{Total memory footprint of a single CTA} = 4Nb + 2b^2 \tag{3.2}$$

$$\text{Total \# of CTAs} = \frac{N}{b}.\frac{N}{b} = \frac{N^2}{b^2}$$

$$\begin{aligned} \text{Total memory footprint on GPU} &= \frac{N^2}{b^2}.(4Nb + 2b^2) \\ &= \left(\frac{4N^3}{b} + 2N^2\right) \\ &\approx \frac{4N^3}{b} \end{aligned} \tag{3.3}$$

This stencil involves $3N^3$ FLOPs. Hence,

$$\text{Compute - to - communication ratio} = \frac{3b}{4} \tag{3.4}$$

Note that due to the kind of dependences present in this 2-D stencil, the tiles need to be executed in a wavefront manner, which means some kind of synchronization is required between adjacent wavefronts. In CUDA, this is usually achieved by using multiple kernel calls. That is, one kernel call per wavefront, implying that the synchronization is done explicitly by the host CPU. Apart from the overhead of multiple kernel launches, this method does not allow the reuse of data within the shared memory of the SMs as well.

But a more efficient way of implementing wavefront parallelization on GPU was developed by Ranasinghe [11], which involves only one CUDA kernel call per stencil and uses a global array-based synchronization between producer-consumer CTA pairs. It also uses global memory atomics to override the CUDA runtime system's ordering of CTA execution. When using this technique for implementing our 2-D stencil, the analytical value in Equation(3.3) would be an upper bound on the total memory accesses on GPU (assuming that the arrays for synchronization stay in the last level cache). The following equation shows the total energy consumed by this stencil on GPU.

$$\begin{aligned} \text{Total energy used on GPU} &= \text{Dynamic energy} + \text{Static Energy} \\ &= \left(3N^3.e_{comp}\right) + \left(\frac{4N^3}{b}.e_{mem}\right) + \text{Static Energy} \end{aligned} \tag{3.5}$$

where, e_{comp} is the energy consumed per FLOP, and

e_{mem} is the energy consumed per memory access.

Prior studies [11, 15] have shown that the second component (due to off-chip memory access) in the above equation is the dominating part of the dynamic energy overhead. Besides, as noted in [10], the energy due to the computation is unavoidable. Therefore, the part of dynamic energy that can be optimized is the factor $\frac{4N^3}{b}$.

$$\text{Static energy} = (\text{Average Static power}) \times (\text{Execution time})$$

Since static power is a constant for a certain architecture/platform, static energy can be reduced only by decreasing the execution time.

3.3 ANALYSIS OF 2-D STENCIL ON SPU

As mentioned before, the lower bound on the memory footprint of this 2-D stencil is $6N^2$. But it has an $O(N^3)$ complexity on GPU mainly due to the absence of a on-chip data sharing mechanism between the Streaming Multiprocessors (SMs). Enabling some form of communication between these SMs can bring in some reduction in this overhead. Being able to locally share the dependent data between thread blocks completely on-chip means the total off-chip accesses will be reduced to the lower bound. However, since the platform can have only a fixed number of SMs and the number of CTAs launched can be arbitrarily large, this is not always achievable.

Let there be a $p \times p$ grid of SMs on the platform and if only one CTA is executed on an SM at a time, the launched grid of thread blocks will have to be executed in passes that fit the physical grid. Meaning, the $\frac{N}{b} \times \frac{N}{b}$ grid of CTAs launched needs to be executed in passes of size $p \times p$. Therefore, the tiles of data to be shared between thread blocks in

adjacent passes have to go through the off-chip memory.

$$\begin{aligned} \text{Total \# of passes} &= \left(\frac{N}{pb}\right).\left(\frac{N}{pb}\right) \\ &= \left(\frac{N^2}{p^2b^2}\right) \end{aligned} \tag{3.6}$$

$$\begin{aligned} \text{Memory footprint of a single pass} &= 2.\left(2Npb + p^2b^2\right) \\ &= 4Npb + 2p^2b^2 \end{aligned} \tag{3.7}$$

$$\begin{aligned} \text{Total off-chip accesses on SPU} &= \left(4Npb + 2p^2b^2\right).\left(\frac{N^2}{p^2b^2}\right) \\ &= \frac{4N^3}{pb} + 2N^2 \approx \frac{4N^3}{pb} \end{aligned} \tag{3.8}$$

$$\begin{aligned} \text{Overhead due to spill-restore between passes} &= \left(\frac{4N^3}{pb} + 2N^2\right) - 6N^2 \\ &= \frac{4N^3}{pb} - 4N^2 \end{aligned} \tag{3.9}$$

$$\text{Compute - to - communication ration on SPU} = \frac{3pb}{4} \tag{3.10}$$

This shows that there is a p-fold improvement in balance possible on SPU over GPU.

CHAPTER 4

Optimized SPU programming model and runtime system

In this chapter, we discuss bottlenecks that were observed while evaluating the SPU architecture and describe how we resolved them.

4.1 PROBLEM OF USELESS SPILL-RESTORE

Uguen [13] noted that there are some unwanted spills and restores to and from global memory occurring between passes on the SPU. Let us try to understand this issue using an example. Consider a 2-D stencil problem with a tiled iteration space of dimension $4 \times 4 \times 4$, meaning, there are 4 tiles along each dimension. Say, the SPU kernel is launched with a 4×4 grid of threadblocks such that each threadblock is responsible for computing 4 tiles. Algorithm 1 shows the pseudo-code executed by each threadblock. Here, $(x, y, z) \rightarrow (x, y)$ is the tile-to-threadblock mapping. If the SPU on which this code executes, has a 4×4 grid of SMs, then the SPU runtime will map the $(x, y)^{th}$ threadblock to the $(x, y)^{th}$ SM on the SPU grid. Due to the dependences in this stencil, the tiles have to be executed in a wavefront manner. Although all SMs start executing their respective threadblocks simultaneously, each SM(x, y) has to wait idly at the *syncblocks* barrier for $x + y$ iterations of the loop before it can start computing a tile. The wavefronts in Figure 4.1(a) shows the timestep at which each SM in this SPU starts computing its assigned tiles. Figure 4.1(b) shows the progression of activity on each SM. Because of the diagonal wavefronts, every SM(x, y) such that $x + y = c$ (where, $c \geq 0$ is a constant) shows similar activity. So, we plot the SMs along the vertical axis using $x + y$, and the horizontal axis represents the timesteps. For this problem, the wavefront progresses from timesteps 0 to 9 (as indicated by the 'for' loop in Algorithm 1). The *syncblocks()* in line 11 ensures the necessary synchronization between wavefronts and

Algorithm 1 Wavefront time-loop in SPU kernel for a 4×4×4 tiled iteration space

```
1: i ← blockIdx.x
2: j ← blockIdx.y
3: k ← 0

4: for t from 0 to 9 do
5:     if (t ≥ i + j) AND (k < 4) then
6:         Load inputs from north and west CBs
7:         Compute Tile(i,j,k)
8:         Store output to south and east CBs
9:         k ← k + 1;
10:    end if
11:    syncblocks();
12: end for
```

allows the output of each tile to be passed on to the neighboring tile (or stored to the global memory in case of a boundary tile) through the CBs. Note that all the iterations of this 'for' loop are executed by every threadblock; hence, every threadblock executes 10 block synchronizations on the SPU in this kernel. Therefore, when a threadblock is not computing a tile, it will be idly waiting at a barrier. The filled nodes in Figure 4.1(b) represent a tile being computed while the unfilled nodes are when the SM/block is waiting at a *blocksync*. As far as the SPU programmer is concerned, the activity of the SMs on the SPU is as shown by this figure.

Now suppose that the SPU executing this kernel has only a 2×2 grid of physical processors. This means that only 4 threadblocks can be running concurrently on the entire SPU at any time (assuming that only one threadblock can be concurrently executed on a single SM). So, when a large grid of threadblocks is launched, the SPU's runtime system splits it into 4 passes of size 2×2 that fit the physical grid, as shown in Figure 4.2(a) (the coordinates marked within each block in a pass indicate the SM on which it is executed). The SPU runtime system employs a spill-restore technique between passes to ensure correctness of results. The output produced by the tiles along the east boundary of Pass 0 are spilled to a pre-defined location in the global memory by the Virtual IUs (VIUs) along the east

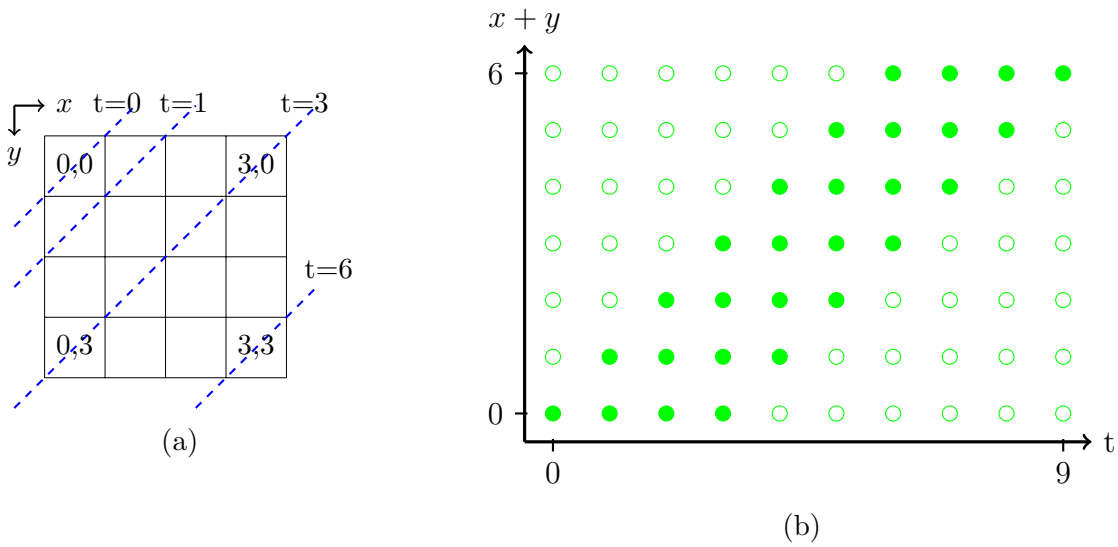


FIGURE 4.1: (a): shows a $4x4$ SPU grid with the wavefronts indicating the timestep at which each SM starts computing a tile as per Algorithm 1. The coordinates marked in the cells indicate (x, y) coordinates of the SM. (b): Shows the activity of each SM on the 4×4 grid. Since every SM(x, y) such that $x + y = c$ (where, $c \geq 0$ is a constant) shows similar activity, vertical axis plots the sum of the coordinates of an SM. The filled nodes indicate that the SM is computing a tile and the unfilled nodes represent the timesteps when the SMs are idly waiting at a barrier.

boundary of the grid and when Pass 1 starts executing this spilled data is restored from global memory by the IUs acting as VIUs along the west boundary of the SPU grid. This spilling and restoring to and from global memory is fully managed by the runtime system and the programmer is unaware of it. The runtime system accomplishes this by inserting some VIU-specific global/load store code into the SPU kernel and this inserted code is independent of any program-specific parameters.

Algorithm 2 gives a pseudocode that is inserted into a SPU kernel by the runtime system to ensure that the values produced by one pass is appropriately made available to a succeeding consumer pass. Here, the variables preceded by a '$'sign indicate registers that are part of the SPU architecture and whose values are managed by the runtime system. Lines 3 to 6 are executed on a spilling VIU that stores data to global memory. Therefore, the number of spills will be equal to the number of *syncblocks()* encountered by it. However,

Algorithm 2 Code inserted for spill-restore between passes

```
1: if $regviu then
2:     if $regiudir == (south OR east) then                    ▷ Spilling VIU
3:         while !($reglast) do
4:             syncblocks();
5:             Spill current tile in CB
6:         end while
7:     end if

8:     if $regiudir == (north OR west) then                    ▷ Restoring VIU
9:         while !($reglast) do
10:            Restore next tile to CB
11:            syncblocks();
12:        end while
13:    end if
14: end if
```

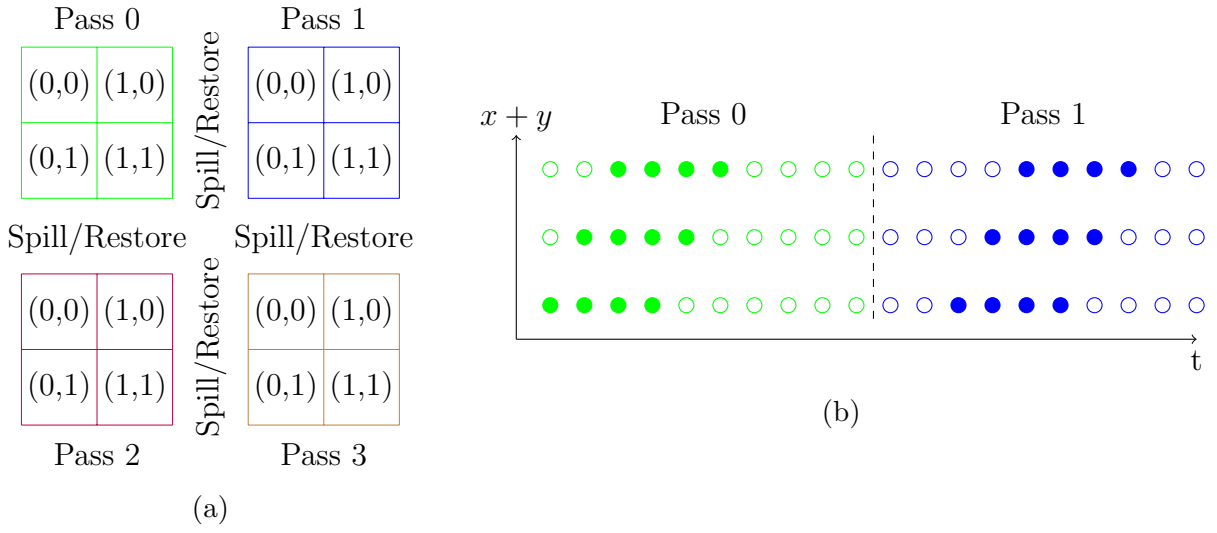


FIGURE 4.2: (a): shows the number of passes on a 2 x 2 SPU grid for a problem that launches 4 x 4 threadblocks, each computing 4 tiles; and the coloring scheme for each pass. (b): shows a timeline of when Pass 0 and Pass 1 are executed on the SMs. The horizontal axis is the time-step and the vertical axis is the sum of coordinates of an SM on the grid. The filled nodes denote that a tile is computed and unfilled nodes denote steps where an SM has to wait at a blocksync without computing any tile.

the number of restores depends on the number of spills that took place in the producer pass. Now coming back to Algorithm 1, though every block does 10 iterations of the loop, it computes a tile and produces useful output only when the condition in line 5 is satisfied (which happens only 4 out of 10 times in our example). Figure 4.2(b) shows the work done by

each SM on the 2×2 SPU grid during every iteration of the loop. As per our example, every threadblock computes 4 tiles. Therefore, there are only 4 shaded circles per SM per pass. Nonetheless, every threadblock has to go through 10 *syncblocks()*, irrespective of the number of tiles computed by each threadblock. This in turn implies that every SM encounters 10 *syncblocks()* in each pass where it is scheduled with a threadblock. The runtime system does not have any information on which *blocksyncs* are useful and which are not. Consequently, each spilling VIU between Pass 0 and Pass 1 makes 10 spills to the global memory when only 4 of those are useful. Since the number of restores depends on the number of spills, the restoring VIUs also end up doing redundant loads from the global memory.

4.2 PRODUCER-CONSUMER SYNCHRONIZATION ON SPU

The major reason for these unwanted spills/restores, between passes on the SPU, is the fact that the inter-block synchronization on the SPU is defined as a global barrier, where all active threadblocks have to reach this barrier in order for the threadblocks to advance. Also, the multi-pass execution is invisible to the programmer. As a result, in order to avoid any deadlock, the programmer is forced to design the wavefront time loop in such a way so that every spawned threadblock executes every required inter-block synchronization barrier in the program. This abstraction supported by the runtime system eases the task of programming the SPU (to some extent), but it negatively impacts the energy consumption of the system. Apart from the useless spill/restore issue, there is one more redundancy resulting from this abstraction. In figure 4.2(b), you can see that though the last tile in Pass 0 is computed at $t = 5$, the pass does not finish executing until $t = 9$. From time-steps 6 to 9, no SM is doing any useful computation. Similarly, in Pass 1, there are periods of no useful work at the beginning and end of the pass. Such redundant time-steps will appear at the start and/or end of other passes as well. This extends the overall execution time of the kernel which in turn leads to increase in both dynamic and static energy consumption.

Algorithm 3 SPU kernel with decoupled synchronization for a 4×4×4 tiled iteration space

```
1: i ← blockIdx.x
2: j ← blockIdx.y

3: for k from 0 to 3 do
4:     syncblocks(NORTH)
5:     syncblocks(WEST)

6:     Load inputs from north and west CBs
7:     Compute Tile(i,j,k)
8:     Store output to south and east CBs

9:     syncblocks(SOUTH)
10:    syncblocks(EAST)
11: end for
```

The fact that the inter-tile dependences in stencil programs are always between adjacent tiles makes us question the necessity of a global barrier. An effective solution to both the aforementioned problems is to replace the global synchronization barrier across all SMs with a localized variant which allows to synchronize only between two neighboring processors. This enables the runtime system to still support the programming abstraction while avoiding the redundancies. Thus, we introduce a producer-consumer based synchronization on the SPU. This is done by simply modifying the existing *bar.blocksync* instruction to take in an extra argument that specifies the direction in which to synchronize. Now, every SM has to synchronize with each of its neighbors in a decoupled fashion, meaning the kernel code needs to include separate instructions to synchronize with each neighbor. However, this leads to further simplification of the SPU programming model.

With this point-to-point synchronization technique, the wavefront-time loop in Algorithm 1 is no longer needed. The modified kernel loop executed by a threadblock, running on an SM, is as shown in Algorithm 3. The k-loop just iterates over each tile that the threadblock is responsible for computing. The decoupled blocksyncs at the beginning of the loop, along the NORTH and WEST directions, ensure that the threadblock is stalled from advancing

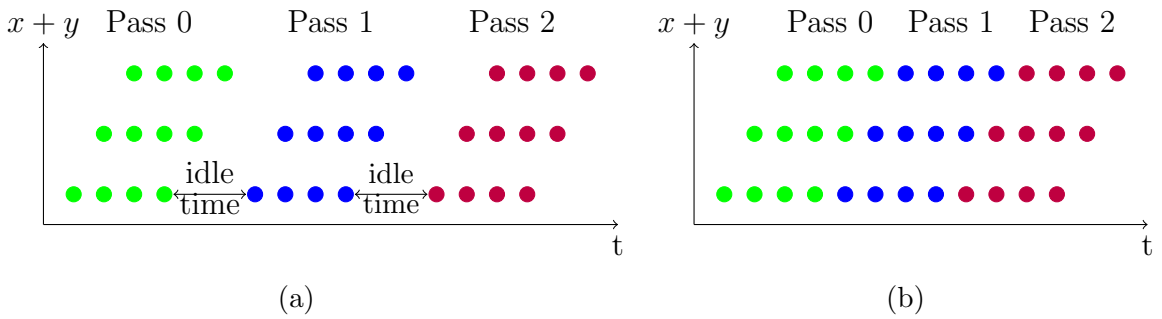


FIGURE 4.3: (a): With point-to-point synchronization, the SMs do not have to go through any more unwanted blocksyncs. Number of *blocksync* instructions per thread-block is equal to the number of tiles computed by it. But, since the thread-blocks are scheduled on the SPU grid in a Pass-by-Pass manner, SMs suffer an idle period between passes. (b): With overlapped execution of thread-blocks from different passes, the idle period in SMs is eliminated leading to better load balance.

until the input data is ready. Likewise, after computation of a tile, the blocksyncs along the output directions (i.e., SOUTH and EAST) stall the threadblock until its dependent blocks are ready to use the currently produced output. Thus, now each block needs to execute only as many iterations as the number of tiles it computes which means the redundant iterations within each pass (seen in Figure 4.2(b)) are not present any more. Also, the number of spills executed by each VIU is now equal to the number of blocksyncs it resolves with its producer SM. For instance, the number of spills by an east VIU is equal to the number of *syncblocks(EAST)* executed by its neighboring SM along the east boundary. Figure 4.3(a) illustrates the work done by each SM on a 2×2 SPU grid, for our previous example, when implemented using point-to-point synchronization. Here, you see that each pass finishes as soon as its last tile is computed.

4.3 OVERLAPPED EXECUTION OF PASSES

In Figure 4.3(a) we can see that each pass now finishes as soon as its last tile is computed (and output stored to the global memory). However, the runtime system follows a pass-by-pass scheduling of threadblocks to SMs. Consequently, threadblocks from Pass 1 are scheduled only after Pass 0 completely finishes execution and all its threadblocks exit. The

last threadblock in Pass 0 exits after $t = 5$; hence, Pass 1 starts only at $t = 6$. However, there are other threadblocks in Pass 0 which exit before $t = 5$ and the corresponding SMs stay idle until the next pass is scheduled. For example, SM$(0, 0)$ finishes its tiles in Pass 0 by $t = 3$ and stays idle until $t = 6$. Such gaps of idle period are seen on every SM when advancing from one pass to another. During such idle periods, though there is no dynamic activity on the SMs they add to the static energy overhead of the SPU.

In order to enable the multi-pass execution of threadblocks on the SPU grid, some special registers have been included in the SPU architecture [13]. These registers hold information related to the index of the passes, spill/restore counts of VIUs, etc. This information is necessary to aid the runtime system in scheduling the passes and also for memory address calculation related to spills and restores between passes. However, each piece of information is maintained at a global scope, meaning there is only one register available per datum for the entire SPU. For example, there is only one register '*cpassx*' which holds the x-coordinate of the currently executing pass; hence, all SMs have to be in the same pass at any time. Maintaining the pass and spill status information at a global level limits the runtime system's capability of executing threadblocks belonging to different passes simultaneously on the SPU grid.

However, in order to eliminate the idling of SMs between passes, seen in Figure 4.3(a), it is necessary to allow threadblocks from a subsequent pass to be scheduled on an SM as soon as it finishes its threadblock in the current pass. We refer to such an execution model as the 'overlapped execution of passes' since threadblocks from different passes may be concurrently running on the SPU grid. In order to enable this overlapped pass execution model, the SPU hardware registers that store the status of passes and spill/restore are now moved to a local scope. This means that every SM and/or IU has its own register to hold these status information thereby allowing each processor to progress its state without waiting for others.

TABLE 4.1: SPU-specific registers that hold the Pass and Spill/restore related information

Register	**Description**	**Scope**
regpassx	Total number of passes along the horizontal dimension for current kernel	Global
regcpassx	x-coordinate of the currently executing pass	Local
regcpassy	y-coordinate of the currently executing pass	Local
regspillnb	total spills/restores (blocksyncs) in one pass (used only by IUs)	Local
regcspillnb	count of the completed spill/restore in current pass	Local

Table 4.1 shows the SPU's registers that hold status of passes and inter-pass spill/restore. The ones with 'Local' scope are the ones that have been modified (localized) to support the overlapped pass execution model. With this optimized execution model, the activity of the SMs on a 2×2 SPU grid for our example is as shown in Figure 4.3(b).

CHAPTER 5

POWER MODEL FOR SPU

This chapter describes how we extended an existing GPU power model to estimate both dynamic and static power consumption of the Stencil Processing Unit. First, we provide an overview of the existing GPUWattch power model and then explain the modifications included to extract a refined power estimate for the SPU architecture.

5.1 OVERVIEW OF GPUWATTCH

Leng et al. [6] has developed a power model for GPGPUs, called *GPUWattch*, that is capable of cycle-level power calculations. This model was incorporated into the GPGPU-Sim [3] simulator. The total power consumption of any platform comprises two elements, namely, dynamic power and static power. In GPUWattch, the dynamic power model is driven by 30 different performance counters whose values are updated periodically by the performance simulator in GPGPU-Sim. These counters are associated with different microarchitectural components of the GPU architecture and the dynamic activity collected by them are sent to the power model at regular intervals (approximately every 500 cycles). This enables GPUWattch to build a trace of the instantaneous dynamic power consumption for a simulated kernel and thereby estimate the average dynamic power consumed by the kernel at the end of the simulation.

At the back-end, GPUWattch uses McPAT [7] for modeling the different microarchitectural components in the GPU architecture. Since the fundamental blocks available in McPAT are mostly suited for general-purpose CPUs, Leng et al. adapted these to conform to the NVIDIA GPGPU processor design and followed an iterative approach of empirical valida-

TABLE 5.1: Components of Dynamic Power in GPUWattch model

Component	**Description**
IBP	Instruction Buffer Power
ICP	Instruction Cache Power
DCP	L1 Data Cache Power
TCP	Texture Cache Power
CCP	Constant Cache Power
SHRDP	Shared Memory Power
RFP	Register File Power
SPP	Streaming Processor execution unit Power
SFUP	Special Function Unit Power
FPUP	Floating Point Unit Power
SCHEDP	Warp Scheduler Power
L2CP	Shared L2 cache power
MCP	Memory Controller power
NOCP	Network-on-chip power
DRAMP	DRAM power
PIPEP	Power associated with the Pipeline
IDLE_COREP	Dynamic power associated with cores that are idle due to load imbalance [6]
CONST_DYNAMICP	Constant component of dynamic power [6]

tion and correction to refine the GPUWattch power model. The static power of the GPU is estimated by McPAT. It takes into consideration the different parameters, provided through the GPUWattch configuration file, to model the different architectural blocks. For dynamic power estimation, it uses the values collected by the performance counters from GPGPU-Sim. The value collected within each performance counter is described in the GPUWattch manual [1]. The total dynamic power is computed as the sum of runtime power consumed by 18 different modeled components. These components of dynamic power are summarized in Table 5.1.

5.2 SPU LEAKAGE POWER MODEL

The Stencil Processing Unit has been derived from the NVIDIA GPGPUs by adding a few microarchitectural blocks to the existing architecture; but, it does not alter the design of any of the current components. Consequently, the existing GPUWattch model can be refined

to estimate SPU's power consumption by simply adding necessary parts and/or counters to model these newly included components. The two additions, introduced by SPU, in the architecture are: (1) Communication Buffers (CBs) and (2) Interface Units (IUs). Therefore, we incorporated models for these two units into the existing GPUWattch.

5.2.1 Modeling Leakage Power of Communication Buffers

The Communication Buffers (CB) are the units of storage shared by a pair of neighboring processors on the SPU included to enable on-chip data sharing between neighbors. Since the CBs have been conceived as a simple extension to the shared memory within an SM, they are modeled exactly like the shared memory as pure RAM with multi-banked structure. The size of the CB is configurable through the input file to GPUWattch. Since there is a pair of CBs between every pair of neighboring coarse-grain processors (both SMs and IUs are considered as coarse-grain units on the SPU), we view them as four CB memory spaces associated with each SM and 1 CB space associated with each IU, and these are included into the respective models in GPUWattch. Thus, the total CB storage space in a SPU is given by:

$$\text{Total CB space} = (4 \times (\#\text{SMs}) + 1 \times (\#\text{IUs})) \times (\text{Size of single CB}) \tag{5.1}$$

5.2.2 Modeling Leakage Power of Interface Units

The Interface Units (IU) have been added along the boundary of the SPU grid to orchestrate overlapping of off-chip memory access with computation. Their purpose is to perform set of load/stores between global memory and communication buffers along the border. This being an early study on the SPU architecture, we would like to keep the conception of IU simple and easy to understand. So, currently the IU is modeled as a truncated SM, that is, an SM with limited capabilities. Since their primary function is to perform loads and stores, the instructions executed by these units are likely to a subset of integer arithmetic

TABLE 5.2: Comparison of SM and IU microarchitecture

Component	**SM**	**IU**
Instruction Fetch Unit	✓	✓
Load Store Unit:		
Shared Memory	✓	×
Constant Cache	✓	✓
Texture Cache	✓	×
Communication Buffer	✓	✓
Execution Unit:		
Register File	✓	✓
Instruction Scheduler	✓	✓
Integer ALU	✓	✓
Floating Point Unit	✓	×
Special Function Unit	✓	×
Complex ALU (mul/div)	✓	✓

instructions for address calculation, branches and of course, load/store instructions to global and CB memory spaces. Therefore, some of the microarchitectural blocks in an SM are not going to be used by the IUs. Table 5.2 lists the microarchitectural components within an SM and IU. Those marked with a '×' symbol indicate the parts that are not present in the IU. This model of IU has been incorporated into the existing GPUWattch, to get a more realistic estimate of leakage power for the SPU.

We currently do not have a compiler for the SPU; hence, we use hand-written assembly (PTX) benchmarks for the SPU. In our benchmarks, we make sure that the code for the IU does not contain instructions that are beyond its capability. The SPU programming model calls for the programmer (or a compiler) to include the code for the IUs in the SPU kernel. Eventually, it would be the compiler's responsibility to ensure that the instructions compiled for the IU conform to its limited capability. It should also be noted that since the Interface Units are merely specialized load/store units, they can be implemented as optimized DMA engines. We discuss this in the final chapter and its implementation will be part of our future work.

5.3 EXTENDING THE DYNAMIC POWER MODEL

As noted before, the dynamic power estimation in GPUWattch is driven by a set of 30 performance counters which record the activity pertaining to different architectural blocks within the GPU. These values are collected from GPGPU-Sim and passed on to GPUWattch through a software interface. The available counters can be broadly classified into two types: the ones that are related to resources shared by all SMs, and those collecting activity of components private to each SM. In the case of the latter set, the activity across all SMs is nevertheless accumulated into a single counter and sent to the power model. For instance, the shared memory is a resource that is private to every SM on the GPU and the accesses to this resource occur within each SM independently. However, the total number of shared memory accesses, across all SMs, is accumulated into a single performance counter, namely SHRD_ACC. Since all the shared memory regions present in the GPU are architecturally uniform, the power consumed per access of this region is same, irrespective of whether it occurs in SM(0,0) or SM(2,2). Therefore, it is sufficient to track the total number of accesses in order to compute the dynamic power spent in performing them.

Following this principle, we add an extra performance counter, called CB_ACC, in the SPUSim to record the total number of accesses to the Communication Buffers across all coarse-grain processors (which includes both SMs and IUs). Using this counter, the new CBP (Communication Buffer Power) component of dynamic power is calculated in the GPUWattch. This is the extra power component added, to the existing list given in Table 5.1, so as to adjust the estimated dynamic power for the SPU. Note that this new performance counter was included in the SPUSim by Uguen [13], nonetheless it was not being used in the actual power estimation due to absence of a proper model for the CBs. With the inclusion of a shared memory-like model for the CBs into GPUWattch, a better estimation of runtime power consumed by these storage units is now in place.

The other addition in SPU architecture is the IUs. In order to account for the dynamic power consumed by these units no add-ons are necessary. As the microarchitectural blocks in the IUs are a subset of those in the SMs, the activity within the IUs will be accumulated into the appropriate existing counters by the original GPGPU-Sim implementation. Hence, though we do not isolate the dynamic power used by the IUs, it is completely accounted for in the power report generated by the extended GPUWattch model in SPU-Sim.

CHAPTER 6

EXPERIMENTAL EVALUATION AND RESULTS

We implemented the previously described optimizations for the SPU architecture as well as the refined power model into the SPU-Sim. With this in place, we conducted experiments using the 2-D stencil benchmark in order to study its benefits. We now present these results.

6.1 EXPERIMENTAL SETUP

First, let us look at the details of the platforms and test cases we use in our experiments.

6.1.1 Test Platforms & Benchmark

In this study, we compare the efficiency (in terms of different metrics) of the SPU platform with that of an existing GPGPU platform. We use SPU-Sim to simulate the SPU architecture while the chosen GPU is simulated on GPGPU-Sim [3]. The configuration of the two simulated architectures is given in Table 6.1. Since GPGPU-Sim is capable of simulating only the Nvidia Fermi architecture, we choose to use the GTX 480 device as our reference GPU platform, while for the SPU we use a 4×4 grid on SPU-Sim. We keep the L1 and L2 caches as it is on the original GTX 480, with L1 cache configured to be 16KB. On the other hand, we do not want to use any data cache on the SPU. Therefore, these units are turned off for the SPU in the performance simulator and power model configurations. All CUDA code compilation has been done using CUDA 4 toolkit (as this is the highest version compatible with both the simulators).

The code for real-world stencils like the Jacobi-2D that has non-canonic dependences, using the synchronizations we have on SPU, can be quite complex. Even on CPU, the generated C

TABLE 6.1: Configuration of test platforms on the simulators

Target	**Number of SMs**	**L1/L2 cache**	**Core frequency**	**Simulator**
GTX 480	15	ON	700 MHz	GPGPU-Sim
SPU	4×4	OFF	700 MHz	SPU-Sim

code for multi-pass execution with similar synchronizations [15] contains more than 300 lines and 6 nested loops. Writing such code for the SPU with the extended instruction set would require significant effort. As no code generator tools or compilers are currently available for the SPU, we limit our evaluation to only one benchmark. Of course, additional benchmarks will eventually be necessary, but are beyond the scope of this thesis.

All results presented in this study use the simple 2-D stencil benchmark described in Chapter 3. On GPU, we use the energy-efficient multi-pass implementation proposed by Ranasinghe [11] which uses a global array based synchronization between dependent blocks and global memory atomics to ensure deadlock-free execution. The SPU kernel conforms to the defined SPU programming model.

6.1.2 Test Cases

The results presented in the following sections use a subset of the problem sizes listed in Table 6.2 for evaluation. In the benchmark description given in Chapter 3, we used only one problem size parameter (N) as we were considering square input matrices; hence cubic iteration space. In order to represent non-cubic problem shapes in our evaluation, we refer to the third dimension of iteration space as K. The size along this dimension essentially controls the number of tiles computed per threadblock. The size of a single CB on the SPU is set to 4 KB for all our experiments except in Section 6.3.4. Therefore, we use a tile size of 32 with all test cases, on both SPU and GPU. This may not be the optimal tile size, but finding and using the optimal tiles is not in the scope of this work. As the simulation time is very high for large problem sizes, we limit our tests to deploy only up to 4 passes on the SPU grid.

TABLE 6.2: List of Test Cases

N × N × K	Tile size	Subtile size	# thread-blocks	# threads per block	Passes on SPU grid	CB Size (KB)	Passes with partial grids
128 × 128 × 128	32	4	4 × 4	8 × 8	1 × 1	4	No
256 × 256 × 256	32	4	8 × 8	8 × 8	2 × 2	4	No
320 × 320 × 320	32	4	10 × 10	8 × 8	3 × 3	4	Yes
384 × 384 × 384	32	4	12 × 12	8 × 8	3 × 3	4	No
512 × 512 × 512	32	4	16 × 16	8 × 8	4 × 4	4	No
256 × 256 × 512	32	4	8 × 8	8 × 8	2 × 2	4	No
320 × 320 × 512	32	4	10 × 10	8 × 8	3 × 3	4	Yes
384 × 384 × 512	32	4	12 × 12	8 × 8	3 × 3	4	No

The case $512 \times 512 \times 512$ takes about 4 hours and the case $1024 \times 1024 \times 1024$ (mentioned in Table A.1) takes upto 10 hours to complete. We also include special cases where a subset of the passes do not completely occupy the processors on the SPU grid. For instance, when $N = 320$, there are only 10 threadblocks along each dimension; hence, the passes along the east and south boundary will occupy $\leq \frac{3}{4}^{th}$ of the processors on the SPU grid. These are also the cases where some of the SMs will take up the role of IUs during the execution of the partial passes.

6.2 EFFECT OF OPTIMIZATIONS ON THE SPU

In Chapter 4, we discussed some improvements beyond the originally proposed design. This section presents the results that empirically validate their benefits.

6.2.1 Impact of producer-consumer synchronization

Ideally, the total number of off-chip accesses on SPU for the simple 2-D stencil problem is given by Equation 3.8 in Chapter 3. The global inter-block synchronization, proposed for the SPU in the earlier works, resulted in many useless global memory accesses. However, as explained before, replacing this global synchronization mechanism with producer-consumer synchronization (also referred to as localized sync) between neighboring blocks helped resolve

TABLE 6.3: Number of off-chip accesses on SPU for different block synchronizations

N × N × K	**Number of Off-chip Accesses on SPU**	
	with global sync	**with localized sync**
128 × 128 × 128	98304	98304
256 × 256 × 256	1179648	655360

this problem. Table 6.3 shows the number of global accesses observed with the two inter-block synchronization techniques. With localized sync, the values exactly match those predicted by the analytical formula in Equation 3.8. Here, $p = 4$ (as our target SPU has a grid of 4×4 SMs) and we use tile size $b = 32$. Note that when $N = 128$, the off-chip accesses is same for both cases because the number of threadblocks are such that only one pass needs to be executed on the SPU. This test case essentially helps validate the functional correctness of the SPU simulator.

6.2.2 Impact of overlapped pass execution model

The idle activity time on SMs, that was occurring between passes with the pass-by-pass scheduling, was eliminated by allowing overlapped pass execution model. This helped reduce the execution time of the kernel as well as the dynamic power consumption of the SMs. For a 2-D stencil of size 256 × 256 × 256, the optimized pass execution model resulted in 31% lower execution time and 17% lower dynamic power over the prior non-overlapped model. Figure 6.1 shows the components of dynamic power (described in Table 5.1) measured with the two models on SPU-Sim. Some of the components are not shown in this chart because either the corresponding part is turned off on the SPU or their average recorded value is less than 0.1 W. It is clearly seen that the 'IDLE_COREP' component is high in the case of non-overlapped model which is a direct impact of the SMs consistently staying idle between passes.

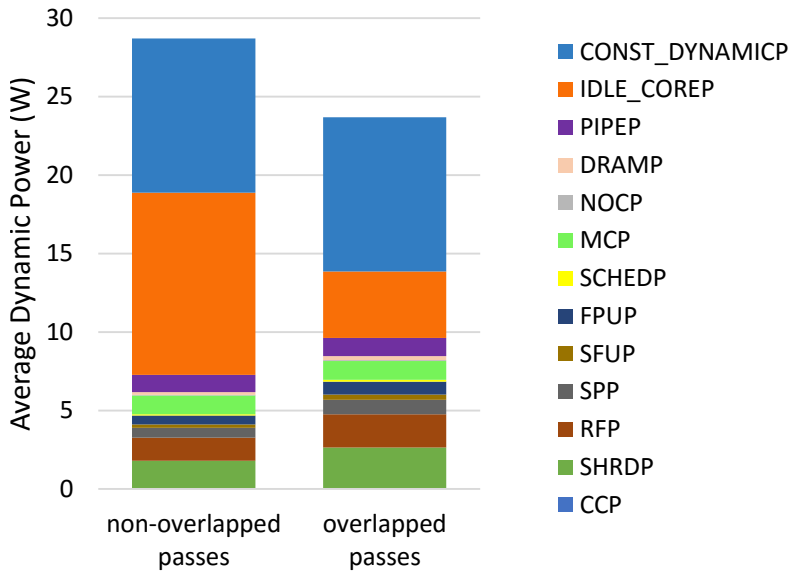


FIGURE 6.1: Dynamic Power for $256 \times 256 \times 256$ problem size with different pass execution models

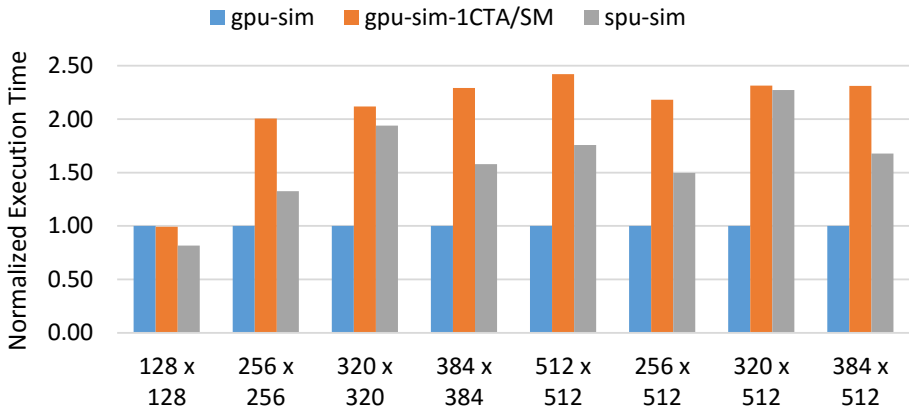


FIGURE 6.2: Comparison of Execution time on GPU and SPU. (The values are normalized to the *gpu-sim*)

6.3 COMPARISON OF SPU WITH GPU

Now, let's compare the performance of our benchmark on SPU with that on GPU. All SPU results presented in this section use point-to-point synchronization between threadblocks and overlapped pass execution model.

6.3.1 Execution Time

The number of core clock cycles, obtained from the simulator(s), is used as the measure of execution time. Figure 6.2 shows the execution times on the two platforms for all the test cases listed in Table 6.2. On the GPU, the runtime system is capable of scheduling multiple threadblocks concurrently on a single SM (depending on the resource utilization per threadblock). But, SPU-Sim currently does not support this feature. Therefore, in order to do a fair comparison, we run the test cases on GPGPU-Sim with and without concurrent threadblocks placed on an SM. The bars corresponding to '*gpu-sim-1CTA/SM*' indicate the case where only one threadblock is scheduled on any SM at a time.

Remember that the SPU proposes to improve the energy consumption by reducing the total off-chip accesses. This is not likely to have much effect on the execution time as our chosen benchmark is a compute bound problem. Nonetheless, we see an average 1.31× speedup on SPU with respect to *gpu-sim-1CTA/SM*. This is due to the difference in the number of processing elements on the two platforms. GTX 480 has only 15 SMs while our simulated SPU has 16 SMs. Apparently, we also see that with concurrent threadblock execution enabled on GPU, it executes much faster than the SPU. This feature is expected to have a positive impact on the SPU performance as well, since it will result in fewer passes on the SPU grid. However, as noted before, this is currently not supported on SPU-Sim and hence, not explored in this paper.

6.3.2 Power Consumption

Both dynamic and static power consumption estimates are collected from GPUWattch on GPGPU-Sim and the extended power model on SPU-Sim. The leakage power reported by the respective tools are shown in Table 6.4. The SPU consumes nearly 70% more leakage power than the GPU. This is mainly due to the addition of the special load/store units (namely, IUs) around the SPU grid. Because the IU has been modeled as a truncated SM, the leakage

TABLE 6.4: Leakage Power on the two platforms. (Note that changing the size of the CBs on SPU from 4 KB to 16 KB increased leakage power only by about 4%.)

	Modelled Leakage Power
GTX 480	46.4 W
SPU (16 SMs + 16 IUs + 4KB CB)	79.1 W
SPU (16 SMs + 16 IUs + 16KB CB)	82.4 W

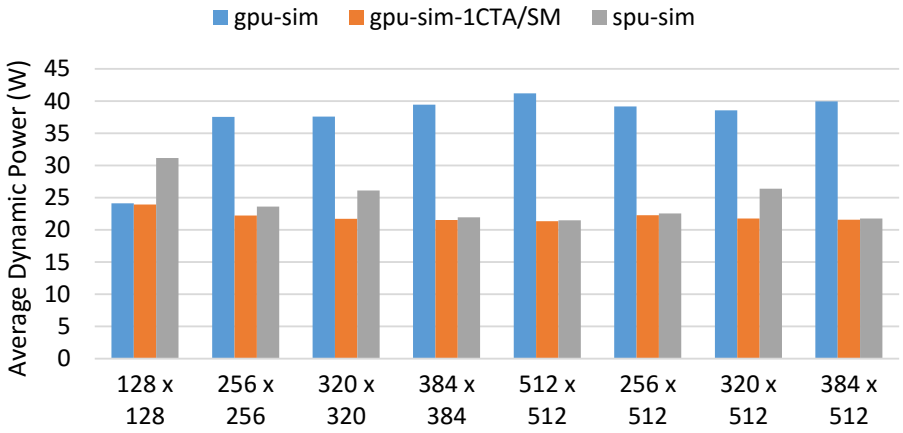


FIGURE 6.3: Comparison of Dynamic Power consumed by 2-D stencil kernel

power per IU is very close to that of an SM. Moreover, for the size of grid that we simulated the number of IUs is equal to the number of SMs, whereas this is a diminishing number as the architecture scales up.

Figure 6.3 shows the dynamic power consumed by the 2-D stencil kernel on GPU and SPU. For all cases, except $N = 128$, there is more power being consumed by *gpu-sim*, while *gpu-sim-1CTA/SM* and *spu-sim* consumes comparable dynamic power. The higher power on *gpu-sim* is on account of multiple threadblocks being executed on each SM which results in more dynamic power being drawn by each multiprocessor. In the case of $N = 128$, there is only 1 pass being executed on both GPU and SPU. And because of the extra dynamically active units present on the SPU, the dynamic power consumed by SPU is more in this case.

6.3.3 Energy Consumption

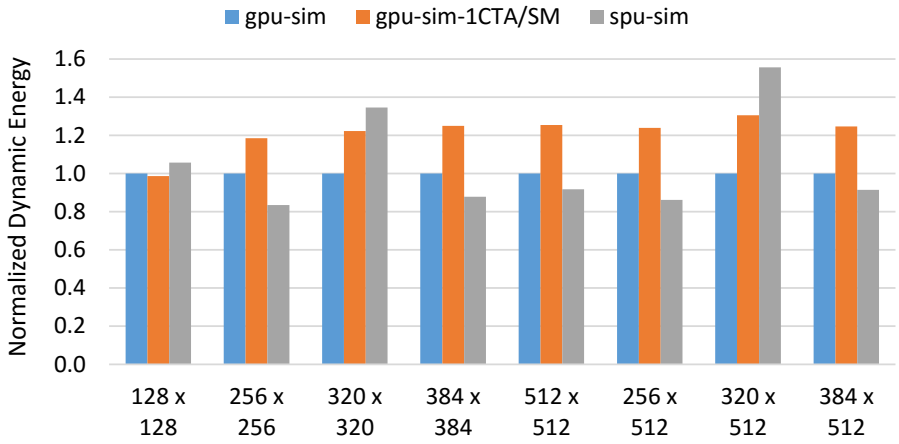
Energy is the product of power and time. Using the statistics collected from the simulator, the total energy is computed as:

$$\text{Total Energy in mJ} = \frac{(\text{Number of core clock cycles}) \times (\text{Total Power in W})}{(\text{Core clock frquency in MHz}) \times 1000} \tag{6.1}$$

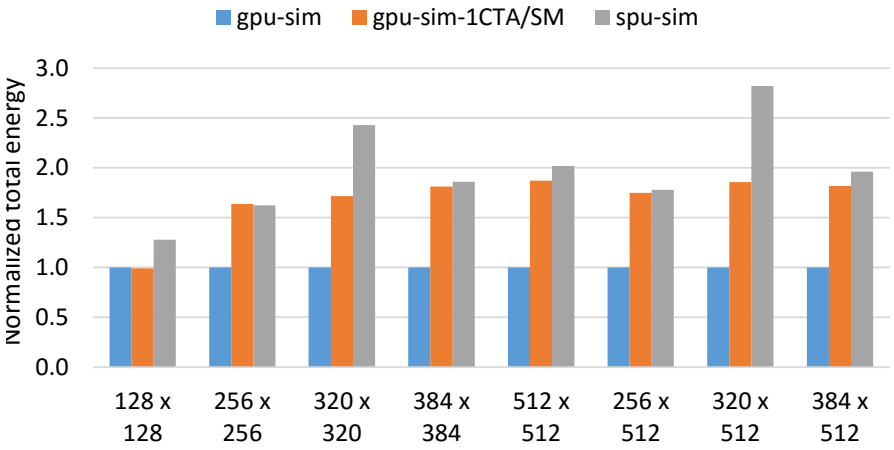
where, Total Power = Leakage Power + Average Dynamic Power.

The core clock frequency for the platforms is listed in Table 6.1. Since the static/leakage power for a platform is constant, irrespective of the code it executes, the values listed in Table 6.4 is used to compute the energy. Figure 6.4(a) gives a comparison of dynamic energy consumed by the stencil kernel on both GPU and SPU. In majority of the multi-pass test cases, the dynamic energy consumed on SPU is less than that on GPU. There is 12% average reduction in dynamic energy on SPU. The only anomaly is in the case of $N = 320$ which exhibits passes with partial grids. Remember that the SPU architecture claims to achieve energy-efficiency by reducing the dynamic energy overhead associated with stencil computation on GPGPU. Overall, this result is an indicator that it indeed achieves this claim.

That being said, when considering the total energy consumption (Figure 6.4(b)), SPU fares much worse than GPU. On an average, the total energy on SPU is about 85% more than that on GPU (excluding the cases with incomplete passes). In spite of comparatively lower dynamic energy, the higher net energy on SPU apparently points a finger at the static energy. From Figure 6.5(a), we can understand that static energy accounts for 70-80% of total energy consumed on the Stencil Processing Unit. Figure 6.5(b) gives a comparison of the energy parts on SPU and GPU for the $512 \times 512 \times 512$ case. It is evident that dynamic energy is lesser on SPU but the total energy is higher only because of the static energy. This is a direct impact of the high leakage power observed on SPU.

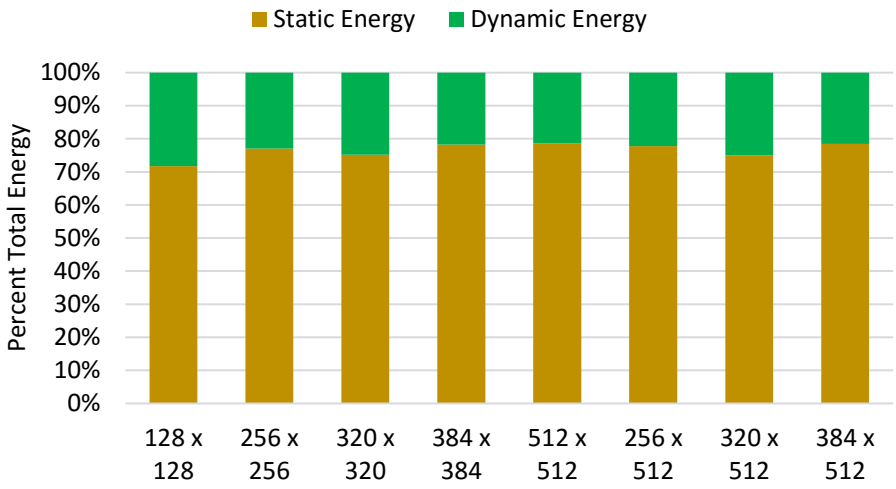


(a) Comparison of Dynamic Energy

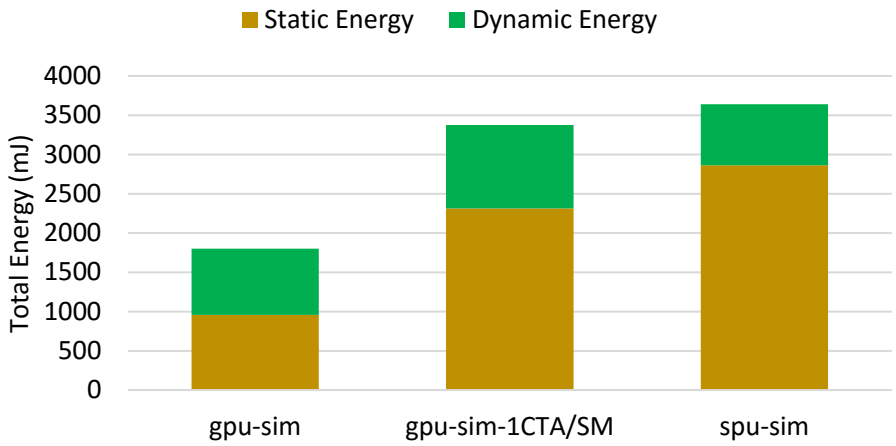


(b) Comparison of Total Energy

FIGURE 6.4



(a) Composition of Total Energy on SPU



(b) Division of energy on different platforms for $N = 512$

FIGURE 6.5

6.3.4 Effect of changing size of CB on SPU

The tile size, for our 2-D stencil, on the SPU is limited by the size of a single CB, which is set to 4 KB (shown in Table 6.2). This implies it can hold only up to 1K of 4-byte ints or floats. Hence, 32×32 is the maximum tile size we can use for this problem. With this tile size, the occupancy a threadblock on a single SM is low. On the GPU, however, the net occupancy on an SM is improved by placing multiple threadblocks concurrently. Since this feature is not available on SPU-Sim, all the test cases used so far suffers poor resource utilization on the SPU. To improve this situation, it is necessary to increase the size of CB on SPU.

We also conducted experiments with the size of CB set to 16 KB. This allows for the tile size to be increased to 64 which in turn increases the shared memory usage in the 2-D stencil kernel for a single threadblock. The shared memory size required per threadblock is such that it prevents more than 1 block being concurrently executed on an SM. This happens on both GPU and SPU. Quadrupling the size of CBs raised the leakage power of the SPU by only about 4% (refer to Table 6.4), indicating that the contribution of these storage units to the net leakage of SPU is not as high as that of IUs. The observations from these experiments are very similar to what we have already seen. Excluding the single pass test case, there is 16% savings in dynamic energy while the total energy consumed on SPU is still 1-10% percent more than on GPU. Details of the test cases and charts related to execution time and energy for this experiment are presented in Appendix A.

CHAPTER 7

Conclusion & future work

Our experiments indicate the viability and benefits of the proposed Stencil Processing Unit. Yet, further optimizations/extensions and more elaborate study on the architecture is necessary to fully justify the gains from this platform. In this final chapter, we give an overview of certain limitations prevailing in our tool, discuss an architectural improvement that can help the SPU outperform GPU in terms of energy, and summarize all inferences from this study.

7.1 LIMITATIONS OF SPU-SIM

The study on the SPU is still in an early stage and so is our simulator (SPU-Sim) for exploring this architecture. This tool currently has the following shortcomings which will be resolved in our future development work.

1. *No Concurrent Placement of multiple threadblocks on an SM*:

   This point was noted in our results section. On the GPGPU, the runtime system tries to schedule multiple threadblocks concurrently on a single SM depending on the block's resource utilization. This feature is currently not implemented on SPU-Sim. The availability of multiple independent warps to a scheduler helps improve memory latency hiding and this is important to GPGPU-like accelerators with small on-chip caches. However, the possible gains from it for the SPU may not be same as that on GPU. Supporting this on SPU-Sim requires careful formulation of strategies for how to emulate the CBs shared between threadblocks executing on the same SM, how to share the CBs (attached to the SM) between these concurrently placed blocks, and when a

CB has to be shared between 2 blocks on same SM, how should the point-to-point block synchronization be handled.

2. *Non-Configurable grid of processors on SPU*:

   Currently, the SPU-Sim can simulate an SPU with a 4 x 4 processor grid only. This means 16 SMs and a total of 16 IUs around it. Many duties of the SPU runtime system including the threadblock to SM mapping, dynamic assignment of values to the *iuid* register in IUs/VIUs, identifying neighboring pairs of processors for block synchronization are all dependent on the grid co-ordinates of the SMs and IUs. These are currently not implemented using parametric functions; hence, this limitation.

3. *Interface Units modeled as truncated SMs*:

   The Interface Units around the SPU grid are currently implemented as limited capability SMs that execute instructions to realize their functionality. However, their primary duty is only to perform sets of loads and stores from the off-chip memory. Therefore, a rather better design would be to have them as specialized DMA units. In the next section, we discuss this in more detail.

7.2 POSSIBLE IMPROVEMENT IN SPU MICROARCHITECTURE

As per our experimental results, the major roadblock in achieving SPU's energy efficiency goal is its high static energy consumption.

$$\text{Static Energy} = \text{Execution Time} \times \text{Leakage Power} \tag{7.1}$$

This component of energy is directly proportional to both execution time and leakage power, and reducing it requires reducing one or both of these factors. Given that the execution time is already well-optimized, the focus should be on how to minimize the leakage power, and this calls for changes in the microarchitecture. As shown earlier, it is the addition of the IUs that acutely raised the leakage power of the SPU over GPU.

TABLE 7.1: Microarchitectural components within IU and DMA

Component	**IU**	**DMA**
Instruction Fetch Unit	✓	×
Load Store Unit:		
Constant Cache	✓	×
Communication Buffer	✓	✓
Execution Unit:		
Register File	✓	✓
Instruction Scheduler	✓	×
Integer ALU	✓	✓
Complex ALU (mul/div)	✓	×

The way the IUs are currently implemented in SPU-Sim is like a general purpose multiprocessor that has to do software work (i.e., fetch, decode and execute instructions) in order to realize the data transfers between off-chip and on-chip memory. A Direct Memory Access (DMA) engine can essentially do the same work completely on hardware. Jamshidi et al. [5] proposed the idea of augmenting the multiprocessors on a GPGPU with special units called Data-Parallel DMA or D^2MA to provide a direct path for data transfers between global and shared memory. A similar design would be the right candidate for our SPU as well, except that in addition to adding them to the SMs, we also need stand-alone units surrounding the SPU's processor grid allowing direct transfers between the global memory and the CBs along the boundary.

Now, we will try to estimate the reduction in leakage power that may be obtained by replacing the IUs with such DMA engines. A DMA unit is like a finite state machine that performs a set of contiguous memory transfers directly between the source and destination spaces without involving intermediate registers. Consequently, such an engine does not require any Instruction Fetch Unit (IFU), Instruction Scheduler or even a complex ALU. However, it will need a minimal register file and integer ALU to enable address calculation. Table 7.1 shows the components, within the current IU design, that can be avoided when us-

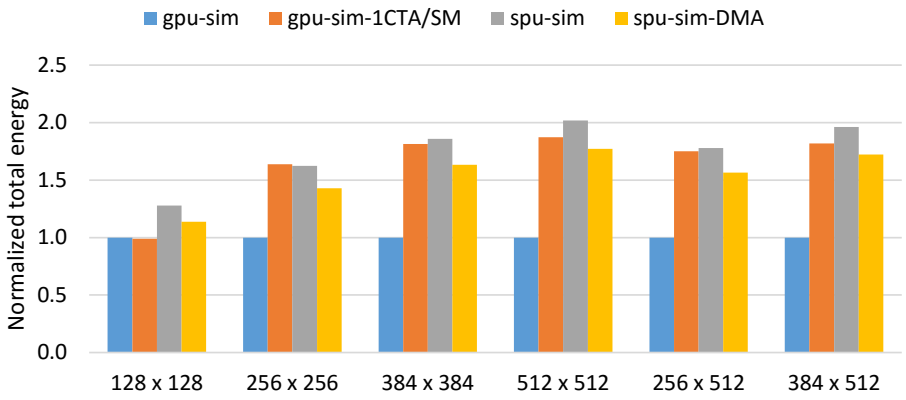


FIGURE 7.1: Estimated Total Energy on SPU with DMA in place of IU

ing a DMA engine and the resulting leakage power for our SPU platform (with CBs of 4 KB) will be about 66.8 W which is 15% less than that with current IU design. In Figure 7.1, *spu-sim-DMA* shows an estimate of the total energy consumed by SPU, for the test cases in Table 6.2, with this reduced leakage power value. We see that, with this improvement, the total energy on SPU can be about 12% lesser than *gpu-sim-1CTA/SM*. Remember, this is a very coarse estimate which does not take into consideration any impact of the design change on execution time or dynamic power.

7.3 CONCLUSION

In this thesis, we introduced two important optimization to the SPU's runtime system, namely, (i) producer-consumer synchronization between threadblocks, and (ii) overlapped pass execution, and also developed a power model for the architecture that provides a refined estimate of the dynamic and static power consumed by a SPU kernel. This improved architecture was evaluated using a simple 2-D stencil benchmark to empirically show the gains from the optimizations. We also compared the execution time, energy and power consumption of the SPU with that of a known GPU platform for the same benchmark. It is seen that the SPU uses an average of 12% lower dynamic energy than GPU, which bolsters the claim of saving energy by reducing the dynamic energy overhead. However, the inclusion of

the special load/store units (IUs) on the SPU increased its leakage (static) power drastically. Therefore, the total energy consumption on SPU is observed to be higher than on GPU with static energy accounting for more than 70% of the net energy for all test cases.

We noted the inefficiency in our current IU design and propose to replace them with DMA engines in future. Our conservative estimates indicate a possible 15% reduction in SPU's leakage power with the use of DMA, and this can help the SPU outperform GPU in terms of energy consumption. In order to fully establish the benefits of the Stencil Processing Unit, more extensive architectural exploration is necessary, for which the inflexibilities in the existing SPU-Sim simulator need to be resolved. Further, it is also important to evaluate SPU with more realistic stencil benchmarks to show its gains in real-world applications. The development of benchmarks for SPU is currently a tedious task due to the lack of a compiler or code generator for its extended ISA. Addressing these identified limitations will be part of our future work.

REFERENCES

[1] *GPUWattch Energy Model Manual Version 1.0.*

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[3] Ali Bakhoda, George L. Yuan, Wilson W.L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 163–174, April 2009.

[4] Mehmet E. Belviranli, Peng Deng, Laxmi N.Bhuyan, Rajiv Gupta, and Qi Zhu. PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15, pages 25–35, 2015.

[5] D. Anoushe Jamshidi, Mehrzad Samadi, and Scott Mahlke. D^{2}MA: Accelerating Coarse-Grained Data Transfer for GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 431–442, 2014.

[6] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, 2013.

[7] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and

manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec 2009.

[8] Sparsh Mittal and Jeffrey S. Vetter. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. *ACM Computing Surveys*, 47(2):19:1–19:23.

[9] Nvidia. NVIDIA's Next Generation CUDA Compute Architecture: Fermi (Whitepaper). Technical report.

[10] Sanjay Rajopadhye, Guillaume Iooss, Tomofumi Yuki, and Dan Connors. The Stencil Processing Unit: GPGPU Done Right. Technical Report CS-13-103, Computer Science Department, Colorado State University, Fort Collins, CO 80523-1873, March 2013.

[11] Waruna Ranasinghe. Reducing off-chip memory accesses of wavefront parallel programs in Graphics Processing Units. Master's thesis, Fort Collins, CO 80523-1873, 2014.

[12] DOE ASCAC Subcommittee Report. Top Ten Exascale Research Challenges. Technical report, February 2014.

[13] Yohann Uguen. SPU-sim: A cycle accurate simulator for the stencil processing unit. Internship Report, August 2015.

[14] Shucai Xiao and Wu chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

[15] Yun Zou and Sanjay Rajopadhye. Automatic Energy Efficient Parallelization of Uniform Dependence Computations. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15, pages 373–382, 2015.

APPENDIX A

ADDITIONAL RESULTS

This Appendix supplements the details of the results discussed in Section 6.3.4. The platforms used in this experiment are same as that shown in Table 6.1. The only difference is that we set the size of the CBs on the SPU to 16 KB. (All other experiments used only 4 KB CBs). As noted earlier, this increase in size of CBs led to a 4% rise in the leakage power of SPU. Table A.1 lists the test cases used in this experiment. The main point to note here is that, the tile size has been increased to 64 due to which the shared memory usage per threadblock is such that only one threadblock can be executed on an SM at a time.

TABLE A.1: Test Cases for the experiment with increased CB size

N x N x K	Tile size	Subtile size	# thread-blocks	# threads per block	Passes on SPU grid	CB Size (KB)	Incomplete passes
256 x 256 x 256	64	4	4 x 4	16 x 16	1 x 1	16	No
512 x 512 x 512	64	4	8 x 8	16 x 16	2 x 2	16	No
768 x 768 x 768	64	4	12 x 12	16 x 16	3 x 3	16	No
1024 x 1024 x 1024	64	4	16 x 16	16 x 16	4 x 4	16	No

Figure A.1 shows the speedup on the SPU with respect to the GPU device. Since there is a mismatch in the number of processors on our two test platforms, we adjusted the observed execution time on the GPU to get the best case time with 16 processors (by multiplying the observed time with a factor of $\frac{15}{16}$). One curve in the figure shows the speedup with the actual observed time while the other curve indicates the speedup on SPU with respect to the aforementioned adjusted execution time. As you can see, the SPU shows 1.2× - 1.4× speedup over GPU even with the adjusted time. This could be because the GPU is now not able to take advantage of hyperthreading, which is possible only with concurrent

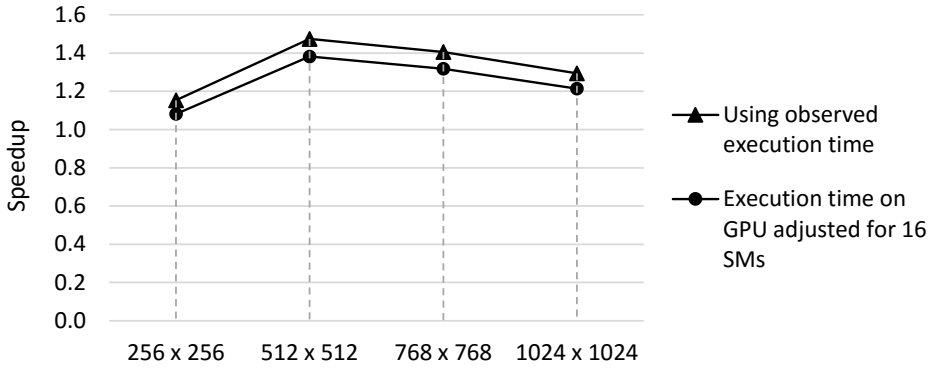
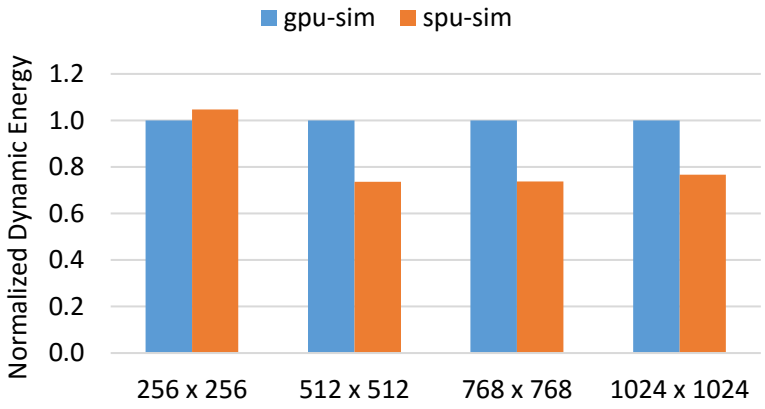
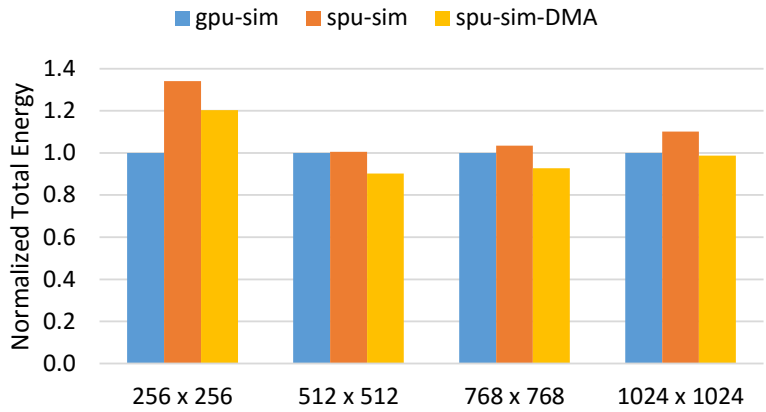


FIGURE A.1: Speedup on SPU over GPU with Tile size 64 x 64

threadblocks executing on an SM. The observed dynamic energy consumption on the two platforms is compared in Figure A.2(a). Similar to what we have seen earlier, problem sizes which require only single pass on the SPU grid do not see any gains. The gains from SPU manifest only with large problem sizes. For all other cases, we see about 25% reduction in dynamic energy. Nonetheless, the total energy (shown in Figure A.2(b)) consumed on SPU is still slightly higher than that on GPU (the difference is not as high as observed earlier in Figure 6.4(b)). But, we can also see that SPU can be more favorable if the IUs are replaced by DMA engines.



(a)



(b)

FIGURE A.2: Comparison of energy consumption with Tile size 64×64. (a): Dynamic Energy, (b): Total Energy. (All values normalized to *gpu-sim*)