SPU-sim : A cycle accurate simulator for the Stencil Processing Unit

Yohann Uguen
Advisor: Sanjay Rajopadhye
From May 25 to August 15, 2015

Colorado State University, Computer Science Department,
80521 Fort Collins, Colorado, USA
`Yo.Uguen@gmail.com,Sanjay.Rajopadhye@colostate.edu`

Abstract. In this report, we describe the implementation of a simulator for a GPGPU extension model called the Stencil Processing Unit (SPU). We explain the features of the SPU that are currently implemented as an extension to an existing GPGPU simulator called *gpgpu-sim*. To ensure the correctness of our simulator, we used the tiled matrix multiplication program as a simplified stencil kernel.

Keywords: GPU architecture, stencil computation, energy consumption, tiling

1 Introduction

As we move to the exascale era, the challenge is not only increasing computation speed but also reducing energy consumption. Graphics Processing Units (GPUs) originally targeted a specialized domain and were therefore more efficient than Central Processing Unit (CPUs) in terms of speed and energy consumption; however, they have recently evolved towards general purpose computing. This trend created a new type of coprocessors, called GPGPUs or General Purpose GPU. They combine features of hardware accelerators and CPUs.

Although accelerators are already more energy efficient than CPUs for compute bound programs, there remains significant room for improvement. It has been shown [1,2] that changing parallelization strategy on CPUs can greatly reduce the energy consumption with almost no loss in terms of speed. As GPGPUs include some of CPUs' tools (e.g., synchronization mechanisms, caches, memory hierarchy), the optimizations made on CPUs could also apply to GPGPUs. These optimizations cannot be directly applied on GPU, but some work-around solutions have been proposed recently [3,4].

Rajopadhye et al. [1] recently proposed a GPGPU extension, called the Stencil Processing Unit (SPU), an extension of GPUs to compute a class of programs called dense stencils computations. This model allows communications between adjacent streaming processors, which is not the case in current GPUs. This feature is proposed to reduce the off chip memory accesses as the computed data is going to be reused by another thread block running on a neighboring streaming processor. However, their claims were presented in the form of analytical formulas, that were not validated empirically.

The goal of this internship was to validate the claimed advantages of the SPU architecture. The specific contributions, are:

- **Implementation of a simulator for the Stencil Processing Unit** as an extension of an existing GPGPU simulator
- **Possible improvements**: We describe different improvements that can be done to the SPU architecture and run time

This report is organized as follows: Section 2 introduces necessary notions to understand the rest of the report; Section 3 describes the architecture simulated by our SPU simulator; Section 4 shows how we ensured the correctness of the simulator; Section 5 shows what can be improved; and Section 6 concludes.

2 Background

In this section, we will introduce the general notions needed to understand the rest of this report. First, we will describe GPGPU programming and architecture, then, we will introduce dense stencil computations, the class of programs targeted by the SPU.

2.1 Programming model for GPGPU

There are two major GPGPU providers and each of them uses a different framework and different architectures. The programming model is shared across vendors with slight differences over naming. In this report, we only use NVIDIA nomenclature, and we consider the kernel to be executed a CUDA kernel.

During the execution of a kernel, the work is distributed over thread blocks. Each thread block will be assigned to a Streaming Multiprocessor (SM). The number of thread blocks and the number of threads per thread block is set by the programmer, regardless of the actual architecture. The compiler will ensure that the parameters are correct, or produce an error.

Every SM has its own shared memory, only accessible within this SM. All the threads within a thread block can communicate through this shared memory but the other thread blocks cannot access the same data, even if they are assigned to the same SM. The next memory level is the global memory, accessible by any SM, but is slower than the shared memory.

Each thread block is divided into warps containing 32 threads each. These warps can synchronize, using barriers, in order to communicate through the shared memory. The thread blocks are completely independent and cannot be synchronized.

The GPGPU run-time system manages the scheduling of the thread blocks, in a way that is totally hidden to the programmer. It will try to concurrently execute multiple thread blocks on the same SM in order to increase performance. It is not always possible due to hardware limitations such as the shared memory size needed by a thread block, the number of threads per thread block, etc.

In the next section, we describe stencil computations and their execution on GPGPUs.

2.2 Stencil computations and its implementation on GPGPUs

In our case, a stencil defines the value of a rectangular grid point in a 2-dimensional spatial grid at time t as a function of neighboring grid points at recent times before t or during t. A stencil computation computes the stencil for each grid point over time steps.

The computation of such kernels can be split into tiles where each tile operates on a subset of the complete domain. Multiple tiles can be executed concurrently after mathematical transformations and proper tiling. Tiles may depend on values produced by other tiles; care must be taken to ensure that these inter-tile dependencies are acyclic. Each tile will then load all the data needed for its execution, which involves writing back the computed data in the closest memory that is shared across all the threads. On CPUs, this level of memory will be the Random Access Memory (RAM), and on GPGPUs it will be the global memory.

When executing these tiles as thread blocks on GPGPUs, synchronization is necessary between thread blocks to ensure that the scheduling of the blocks is correct [3,5]. Current GPUs do not allow synchronization between thread blocks, meaning that the type of synchronization used is not claimed to be supported by the current generations of GPGPUs and might not even work on future generations. In addition to that, the global memory used to store the data for the computation and the synchronization between thread blocks is slow, meaning that a lot of time might be wasted waiting for the load/store unit to perform the global memory accesses.

Rather than use a stencil computation for our tests we used a tiled matrix multiplication algorithm, which exercises the same features of the simulator but is simpler to create and run.

Figure 1 shows how matrix C is tiled with indices $(i, j) \in [0, 3]^2$. Each tile computes a subset of the final matrix C and all tiles are independent. In this example, tile $(0, 0)$ will first multiply the green tile from matrix A and the red tile from matrix B. The yellow tiles from matrix C represent the tiles that need the same tile from A and B during their computation. We can see that these two input tiles are used by six other tiles than $(0, 0)$.

Assigning each tile to a thread block on GPGPUs will make every thread block load from global memory, some data that is also loaded by a neighboring thread block. This introduces long delays and thus, a performance loss, in terms of speed and energy. On modern GPUs this performance and energy degradation is considerably mitigated by the presence of caches, but for simplicity of the discussion, we assume a GPU platform without caches. Rajopadhye et al. claim that even this would have a speed and energy cost that would be further reduced by the SPU, but this is also not empirically justified. The simulator developed in this project is intended to allow all these three cases to be tested.

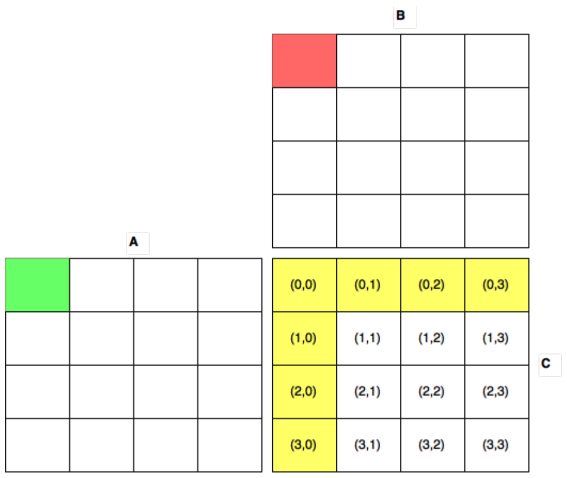

Fig. 1: Data usage in tiled matrix-matrix multiplication

In the next section, we introduce the architecture that we simulate: a stencil specialized GPGPU extension.

3 The simulated architecture

This section is organized as follows: In the first part we will introduce the global architecture simulated by the SPU simulator (SPU-sim); In the second section, we discuss several programming challenges caused by obfuscation of SPU architecture from the programmer; Finally, we will describe one possible optimization for our simulator.

3.1 Principle

The only way for multiple thread blocks to share data on current GPGPUs is to use the global memory. If the data shared between these thread blocks does not fit in a shared level of cache, the accesses to these memory locations will result in global memory accesses. This memory is an off chip memory that introduces long delays for each access.

In order to enable inter-SM communication, the SPU has a new kind of memory called Communication Buffers (CBs). These CBs are used as producer/consumer buffers between adjacent SMs. Each SM can read from its north and west buffers, and may write to its south and east buffers so that the data propagation is limited to one direction (from NW to SE). Every SM is then related to four CBs, one in each direction. Figure 2 illustrates the data propagation between adjacent SMs on GPGPUs. The data produced by the top left SM is written to global memory for the other neighboring SMs to be able to load it. Even though the data was locally close to the loading SMs, global memory accesses are issued. Figure 3 illustrates how the communication buffers are attached to the SMs and with arrows representing the data propagation on the SPU. Dotted boxes represent the SMs and CBs in the neighborhood of the solid line SM.

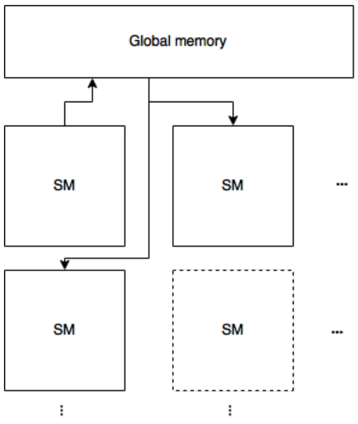

Fig. 2: Data propagation for stencil computation on GPGPU

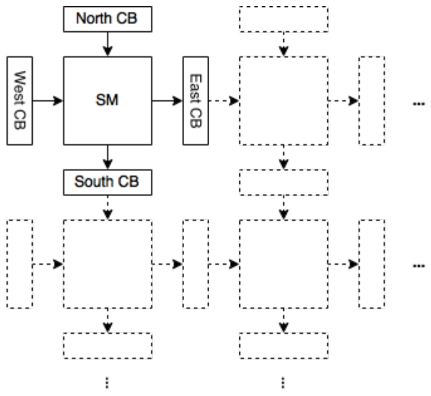

Fig. 3: CBs attachment to SMs and data propagation on the SPU

To ensure that the data is written by the producer before the consumer reads it, a SPU-wide barrier through all the SMs is introduced. We call it a *block synchronization*. We refer to *time step* as the computation between two block synchronizations. Figure 4 shows how synchronization issues are avoided by splitting the CBs into two distinct memories. At time step N the SM1 writes in CB1 while SM2 reads from CB2. A block synchronization make the SMs to switch buffers so that at time step $N + 1$, SM1 writes in CB2 and SM2 reads from CB1. This synchronization ensures that the data is produced before it is consumed. We consider this pair of buffers to be the single, unique CB for the rest of this report.

For the rest of this example, let's consider that we have a four by four grid of SMs. As GPGPU programming hides the actual architecture from the programmer through virtualization, so does the SPU. The programmer might launch an 8×8 grid of threads blocks of the SPU, which means that there are more thread blocks than the number of SMs. In order to perform the computation correctly, the run-time system of the SPU will then have to go through multiple passes. Figure 5 shows how the grid of thread blocks is split into four passes for this example. We identify a thread block by three coordinates i, j, which are its row

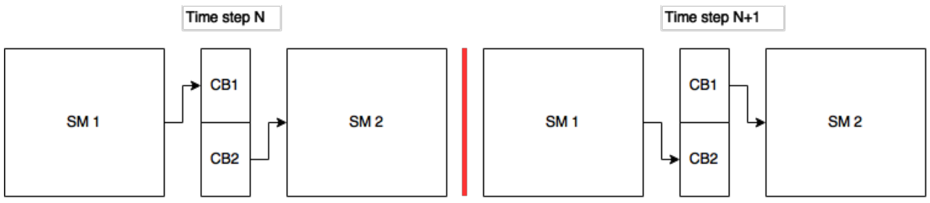

Fig. 4: CBs Accesses by two SMs at two consecutive time steps (separated by the red bar)

and column number in its containing pass and by k, which determines its pass number. The thread block A is then $(0, 0, 0)$, B is $(3, 0, 1)$, C is $(1, 2, 2)$ and D is $(0, 0, 3)$.

The programmer would then select the blocks that are on the frontiers of its grid to load and store data from global memory. All the interior blocks would just transfer data to their neighbors through the CBs. Meaning that the blocks accessing global memory would be the following:

$$(i, j, k), \begin{cases} k = 0, (j = 0, i \in [0, 3]) \;||\; (i = 0, j \in [0, 3]) \\ k = 1, (j = 0, i \in [0, 3]) \;||\; (i = 3, j \in [0, 3]) \\ k = 2, (j = 3, i \in [0, 3]) \;||\; (i = 0, j \in [0, 3]) \\ k = 3, (j = 3, i \in [0, 3]) \;||\; (i = 3, j \in [0, 3]) \end{cases}$$

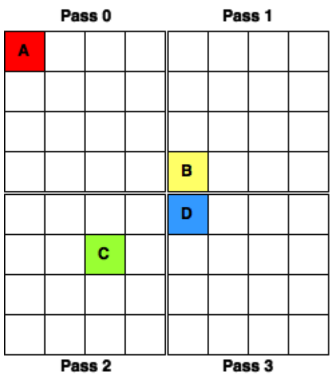

Fig. 5: Grid of thread blocks split into multiple passes on SPU-sim

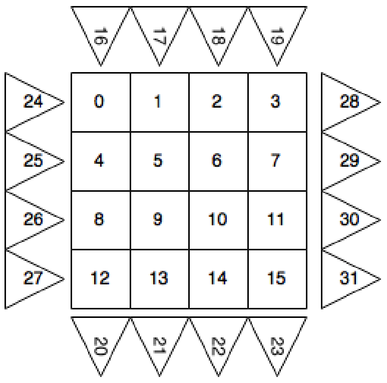

Fig. 6: Hardware placement of SMs and IUs (SMs are squares, IUs are triangles)

During the execution of pass 1, the thread block $(1, 0, 1)$ for example, would expect to read its data from its west neighbor, but it won't be able to read the correct data as $(1, 3, 0)$ already finished its execution. The SPU virtualization mechanism shields the programmer from having to determine which thread blocks are on the borders of the grid so that they load data from global memory and thus, execute a different code than the rest of the SMs. This is why the SPU model introduces the Interface Units (IUs). These IUs are simplified SMs, placed on all sides of the hardware grid of SMs, and their purpose is only to load/store data from global memory and write it into their CBs. Figure 6, shows their placement and their identification number. Each IU is identified by its direction (north, east, south, west) and an *id* starting from the north-west. For example, the *id* of IU number 22 is 2 at pass 0, and the *id* of IU 29 is 5 at pass 3 and 1 at pass 1. As IUs are simplified SMs, if a pass doesn't fill the entire hardware grid, then a SM can act as an IU.

The programmer therefore assumes an unlimited grid of thread blocks, surrounded by IUs. This way, every SM expects the data to come from a CB and not from global memory. When $(0, 0, 3)$ is executed, it expects data from its north and west CBs. And theses CBs will be filled by IUs. The problem is that the programmer was not expecting any IUs between passes as she is not aware of the architecture. In order to ensure the correctness of the results, we need to spill (store to global memory) the output, and restore (load from global memory) the inputs of passes that are not on the borders. This is why SPU-sim introduces Virtual IUs (VIUs).

3.2 Virtual IUs (VIUs)

In order to allow communication between two thread blocks that are not in the same pass, we introduce the VIUs. During a pass, if an IU is inactive because the thread block grid is larger than the actual architecture, this IU is turned into a VIU for the all pass. For example, at pass 0, IUs 28..31 and 20..23 are turned into VIUs. A VIU can then have two different purposes. It can be a:

- **Spilling Unit**: If the VIU direction is south or east, then it is transformed into a spilling unit. It will then, at each time step, store the entire CB that is next to it to global memory.
- **Restore Unit**: If the VIU direction is north or west, then it is transformed into a spilling unit. It will then, at each time step, load an entire CB data, previously spilled, from global memory and store it to its next CB.

System generated code (i.e., not written by the programmer) should be added at compilation time, in order to manage the VIUs. This instruction, would only execute if a special register, within any SM or IU is set by the run time system at every pass. Algorithm 1 shows the type of code that would be produced by the eventual compiler (there is no existing compiler for the SPU yet). In this algorithm, VIU represents a register, private to every SM and IU, which is a boolean value specifying whether it should act like a VIU. The *iu_dir* variable represents is a register that is also present in every SM and IU that is set by the run time system at every pass that can contain four values representing the VIU direction (NORTH, EAST, WEST or SOUTH). A *Spill* operation loads the entire CB and store it to global memory, and a *Restore* reads from global memory and store the corresponding data to the next CB. A *block synchronization* means a SPU-wide synchronization. Finally, *LAST* represents a SPU register that is set to true by the run time system when all the thread blocks from the current pass are done.

```
if VIU then
    if iu_dir is NORTH or WEST then
        do
            Restore;
            block synchronization;
        while !LAST;
    else
        // iu_dir is SOUTH or EAST
        do
            block synchronization;
            Spill;
        while !LAST;
    end
else
    // Normal Code
end
```

Algorithm 1: Code for Virtual IUs

The VIUs will then work as long as the current pass is not completely done. The address calculation for the spilling and restoring is described in detail in the appendix. They might do some useless work by spilling CBs to global memory even though there is no data to save.

Even though the VIUs introduces some accesses to global memory, which is what the SPU tries to avoid, these accesses are negligible, and also unavoidable for the class of computation targeted. To summarize, the IUs are the interface between the global memory and the CBs; the VIUs are the interface (hidden to the programmer) between two neighboring thread blocks that are executed in different passes. The next section describes how to avoid redundant restoring and spilling introduced by the VIUs.

3.3 Avoiding useless restoring and spilling

By reconsidering the example of figure 5, we can see that $(0, 0, 3)$ will wait for some data coming from $(0, 3, 1)$ and $(3, 0, 2)$ by doing a certain number of block synchronization but no effective work. As the entire pass 3 depend on $(0, 0, 3)$, the entire pass of thread blocks will wait for a certain number N of block synchronization to wait for $(0, 0, 3)$ to start its computation, plus a certain number M of block synchronization for the data to come from $(0, 0, 3)$ to its CBs. When waiting for these N block synchronizations, the VIUs will spill and restore data, which involves a lot of wasted time and energy because they are not needed and transfer useless data. In order to avoid this phenomenon, we introduce one new register that contains a boolean value called *regstart*, specifying when one of the loading IU (north or west) has started its computation. When it is the case, *regstart* is set to true and the VIUs are unleashed so that they start their computation at the same time step than the loading IUs. The VIUs won't start spilling and restoring before any computation. In the case of a pass that does not contains any loading Ius, the VIUs must start their computation one time step before any other SM. We avoid that by keeping track of another boolean value, private to each thread block that makes the thread block setting *regstart* to *true*, unleashing the VIUs and executing one more block synchronization when the current pass does not contain any restoring VIUs.

Next section will present our experiment's method to ensure the correctness of the simulator.

4 Ensuring the correctness of the simulator

First, we will describe the main modifications applied to the existing GPU simulator, then, we will present our experiments to ensure the correctness of the simulator.

4.1 Modifications to gpgpu-sim

In order to validate the SPU-sim, we extended a cycle accurate GPGPU simulator called *gpgpu-sim* [7]. This simulator is able to simulate multiple GPGPU architectures and execute native GPGPU assembly. We used the NVIDIA GTX480, a Fermi class architecture, and the NVIDIA API, by compiling our CUDA kernels with *nvcc* and giving PTX assembly[1] to gpgpu-sim. Because there is currently no SPU compiler available, we compiled a CUDA program in order to generate a PTX file and hand modified this file with SPU instructions. We compiled the program using *nvcc* without optimization in order to make the assembly easier to read and modify. Our CUDA code was compiled with the following command which disables optimizations. Whenever *nvcc* uses another tool in its chain, that tool will also have optimizations disabled by the -O0 flag.

```
nvcc -O0 -Xopencc -O0 -Xptxas -O0 -ptx ourFile.cu
```

A list of all our PTX extensions, and a description of how the CBs are implemented in the simulator can be found in the appendix.

Figure 7 represents the execution flow of SPU-sim. When an executable launched by the user includes a kernel call, this one is given to the GPGPU. When SPU-sim is configured, the kernel call and the parameters given to the kernel are catch by the simulator. The simulator then analyses a environment variable specifying if the user wants to execute a different PTX file than the one that would be extracted from the kernel. It then simulates the corresponding PTX file with the given kernel parameters.

[1] NVIDIA's PTX documentation, `http://docs.nvidia.com/cuda/parallel-thread-execution`

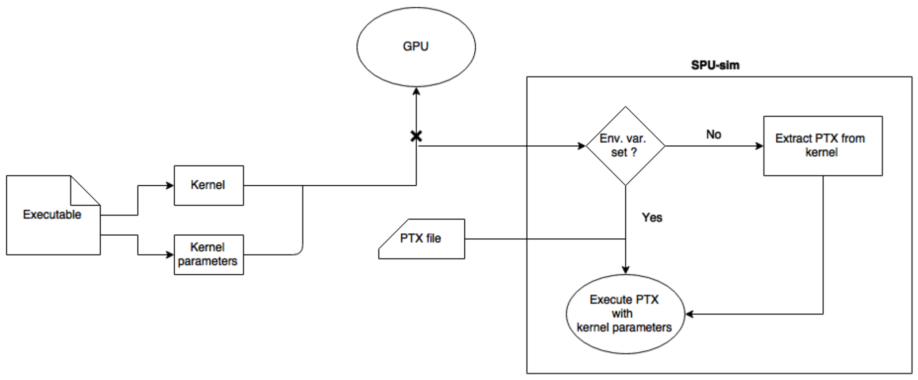

Fig. 7: Execution flow of SPU-sim

Whenever a call to gpgpu-sim is made, it will retrieve some information from the executable such as the thread block grid size or the number of threads per thread block. It will then execute the PTX provided instead of extracting it from the executable by setting an environment variable.

We will briefly describe the modifications made to the cycle-accurate simulator, and then to the power analysis tool.

Modifications to the cycle-accurate simulator The GPGPU architecture that we modified is the NVIDIA Fermi architecture [2]. It contains 16 SMs, where each SM contains one Single Instruction Multiple Threads (SIMT) core. The architecture is a 8×2 grid of SMs, but we modified it to be a 4×4 grid to obtain a square grid of SMs in order to reduce the numbers of IUs necessary. As the GTX480 is a 15 SMs GPU, we modified the already existing configuration file, in order to have 16 SMs plus 16 IUs. All of the SMs and IUs beneficiates from the new instruction allowing a SPU-wise synchronzation. This synchronization first occurs within the SM by invoking a SM-wide synchronization allong threads. It then invokes the SPU-wide synchronization, using the same type of mechanism as the one used for a thread synchronization.

The CBs are implemented as an array of memories. Acting exactly as shared memory. When SPU-sim is called, the modified run time system sets the new registers of all the SMs and IUs. Regarding if they are IUs, VIUs, their direction and attach them to their CBs. It then launch a pass and wait for its completion. During a pass, each SM executes a single thread block, when all the SMs are done, then a new pass is scheduled.

Modifications to the power analysis tool The simulator collects power consumption data using GPUWattch [8]. We modified the collection so that it

[2] NVIDIA's Fermi architecture whitepaper, `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf`

also collects data from the IUs and CBs. GPUWattch is an interface for using McPat [9] in gpgpu-sim. McPat doesn't currently support heterogeneous cores, which are used in the SPU (IUs are different from SMs). As we have a grid of 4×4 SMs, we also have 16 IUs. We then compute the power by assuming that every SM is associated with an IU. We then use the usual way to compute the total energy with GPUWattch, by multiplying by the number of cores by the energy computed for one couple SM+IU.

4.2 Validating SPU-sim

To demonstrate the correctness of the simulator, we wrote a PTX-like kernel code for tiled matrix-matrix multiplication on SPU. The results produced by SPU-sim are the same as the one produced by a CPU, regardless the size of the matrix, the size of the tiles, the size of the grid of SMs, the data footprint or the number of threads.

For our experiments, we considered tiled matrix-matrix multiplication, with a square matrix of size 192×192 and square tiles of size 16×16. This stencil does not introduce any dependency between tiles. In order to spread data through the SMs instead of the global memory, we first execute $(0, 0)$ from figure 1 loading one tile from A and one tile from B from its north and west CBs (previously loaded by the IUs). After its computation, it stores the tiles from A and B into its south and east CBs, it then invokes a block synchronization. At this time step, the thread blocks $(0, 1)$ and $(1, 0)$ will load the needed data from their CBs, etc.

We ensured that the VIUs spilling and restoring was not a bottleneck. Figure 8 shows the number of cycles needed to terminate the kernel, and the average power consumption of the architectures considered. Every code was compiled without any optimization. Our experiments were considering the execution using the SPU without avoiding the useless spilling and restoring before the computation starts as decribed in section 3.3; the SPU with this optimization; gpgpu-sim with only 1 thread block per SM at the time and the native execution using gpgpu-sim.

We used both gpgpu-sim and gpgpu-sim with only one thread block per SM at the time to be able to give a fair comparison with the SPU that only allows one thread block at the time per SM in the current imlementation.

The little number of benchmarks is due to the lack of compiler. Every PTX files executed by the SPU is handwritten.

We can see that the number of cycles needed to perform the computation on SPU-sim is much larger that the one on gpgpu-sim. Even with the optimization by avoiding useless restoring and spilling before the computation starts. And the power consumption is larger too. These results can be explained by the fact that tiled matrix multiplication might not be the best application to consider. As this program does not include any type of dependency across tiles, its execution on GPGPU can be split into thread blocks and benefit from its performances without any need for synchronizations. On the other side, the execution of tiled matrix multiplication on the SPU introduces a lot of block synchronization to

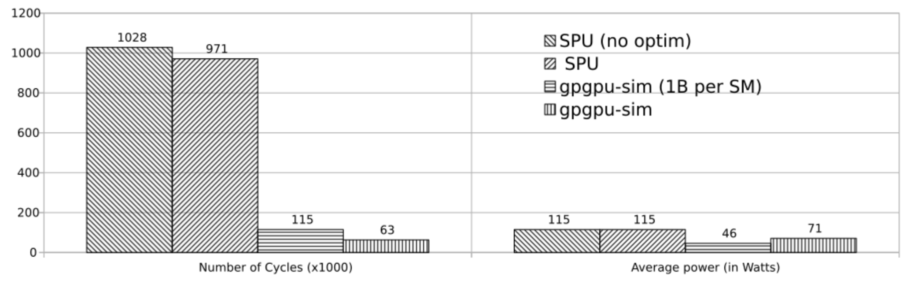

Fig. 8: Larger number of cycles needed and power consumption for tiled matrix-matrix multiplication computation on the SPU than on gpgpu-sim

allow data propagation between SMs. As the SPU is designed to perform well on stencils codes with data dependencies across tiles, matrix multiplication do not benefit from any advantage from the SPU.

The current implementation SPU-sim can be greatly optimized, in the next section, we describe some of them that can be done.

5 Further optimizations

As SPU-sim is at its very beginning state, we describe several optimizations that can be applied to the current implementation.

Don't use reduced SMs for IUs

Our current IUs are SMs in which we removed the following parts:

- **The shared memory**: The shared memory is no used to transfer data from global memory to CB or from CB to global memory
- **L1 cache**: As the L1 cache is associated to the shared memory and that the shared memory is not present in the IUs anymore, we removed the L1 caches
- **Special Function Unit (SFU)**: The SFU is made to perform complex operations in 1 clock cycle (e.g. cosinus, sinus, square root) with a loss in precision. None of the operations provided by the SFU is needed by the IUs so we removed it too
- **Floating Point Unit (FPU)**: The FPU is the unit that allows to do complex and precise calculations over floating points operations. There is no need of FPU when computing memory addresses.

Even with these parts removed, there is still a lot of parts, like the SIMT units that deals with a large area on chip and some useless energy consumption. A further optimization would be to use DMAs[6] instead of our IUs as the DMAs are less complex and thus consume less power.

Dummy block synchronization

We introduced in section 3.3 the mechanism in order to avoid dummy spill and restores. But this can be further optimized. We could avoid doing the block synchronizations involving no computation. This would involve determining the schedule of the thread blocks statically or to ask for a thread blocks schedule to the programmer, and thus, to launch pass 3 at block synchronization number $N + 1$ to avoid many dummy block synchronization.

We could also try to use the same mechanism than in section 3.3 at the end of the thread block computation. Every thread block will perform the total number of block synchronization needed for the entire execution. So, even if we avoid the useless spilling and restoring before the computation starts, we still have not yet implemented the same mechanism for the end of computation. This is more complex to avoid because the first pass determines the total number of block synchronization needed for the entire computation (at the hardware level). And this number is today needed to perform the address calculation for the VIUs. If this address calculatio is done differently, we could keep track of a counter that would be incremented every time a thread block finishes, and when this value is greater than the total number of thread blocks launched, then the VIUs would only perform block synchronizations.

1D-Grid support

Our current implementation of SPU-sim only supports 2D-Grid of thread blocks, but a modification to the run time system could allow 1D-grid of thread blocks. It would give more flexibility to the programmer. Figure 9 shows one example of data propagation for a 1D-grid of thread blocks. Whenever the programmer decides that a thread block will write to a CB, the run time system redirect the write operation to the correct CB considering the implementation.

Another advantage of using 1D grid of thread blocks is to be able to map multiple thread blocks to the same SM. This technique is widely used in GPGPUs to hide memory latencies and to overlap instructions in order to get better performance. This run time decision has been completely disabled for SPU.

Figure 10 shows how two thread blocks can be mapped to the same SM, using a part of shared memory to act like a CB. In this example, we only consider two thread blocks, but this number can be greater as long as the architecture constraints are not met and that there is enough shared memory available to be used as virtual CBs. The run time system would then redirect the writes operations from the thread block A to its CB to shared memory. Following the same mechanism, read operation from thread block B to its CBs are redirected to shared memory.

6 Conclusion & Future work

In this report, we described an implementation of a simulator for the SPU, a hardware accelerator based on GPGPU architectures to compute stencils. We

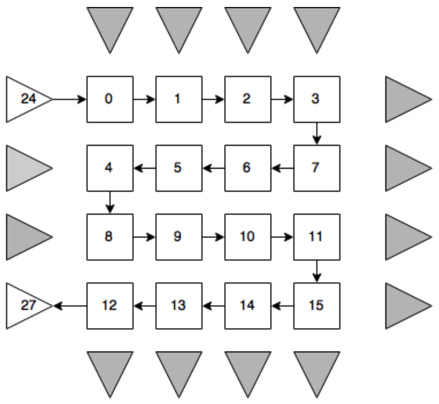

Fig. 9: Run time system decisions for 1D grid of thread blocks

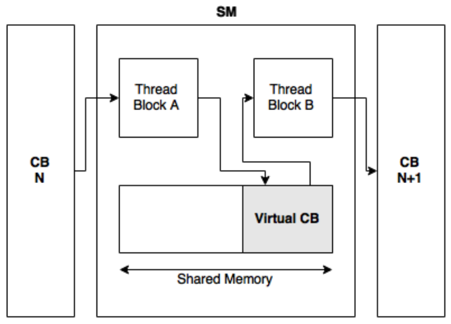

Fig. 10: Multiple thread blocks mapped to the same SM on the SPU

ensured the correctness of the simulator by executing a simple stencil kernel using different parameters. We also discussed about many optimizations that can be applied to the simulator in order to increase its performances.

The current version of the simulator lacks of a lot of optimizations that can greatly increase its performance. As it is very new, the hope for it to become better is not to be neglected. As we described a few optimizations, many others are to come.

The first step towards validating the SPU as proposed by Rajopadhye et al.[1] is to execute a stencil kernel, such as Jocobi, Heat, Smith-Watermann, etc. To simulate one of these kernel on SPU-sim, it is needed to write an assembly file by hand, or to create a compiler for it.

References

1. S Rajopadhye, G Iooss, T Yuki, D Connors, The Stencil Processing Unit: GPGPU Done Right, Technical Report, 1 March 2013
2. Yun Zou and Sanjay V. Rajopadhye, Automatic Energy Efficient Parallelization of Uniform Dependence Computations, Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015
3. Ranasinghe, W. (2014). Reducing off-chip memory accesses of wavefront parallel programs in Graphics Processing Units. Master's Thesis. Colorado State University, Fort Collins, CO
4. Mehmet E. Belviranli, Peng Deng, Laxmi N. Bhuyan, Rajiv Gupta, Qi Zhu, Peer-Wave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization, Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015
5. Holewinski, Justin, Pouchet, Louis-Noël and Sadayappan, P. High-performance Code Generation for Stencil Computations on GPU Architectures. Proceedings of the 26th ACM International Conference on Supercomputing, ICS '12

6. Listanti, M. and Mascitelli, F. and Mobilia, A. D2MA: a distributed access protocol for wireless ATM networks. INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE
7. Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, Tor M. Aamodt, Analyzing CUDA Workloads Using a Detailed GPU Simulator, in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, April 19-21, 2009.
8. Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, Vijay Janapa Reddi, GPUWattch: Enabling Energy Optimizations in GPGPUs, In proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA 2013), Tel-Aviv, Israel, June 23-27, 2013.
9. Sheng Li; Ahn, Jung Ho; Strong, R.D.; Brockman, J.B.; Tullsen, D.M.; Jouppi, N.P., "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on , vol., no., pp.469,480, 12-16 Dec. 2009
10. Shucai Xiao; Wu-chun Feng, "Inter-block GPU communication via fast barrier synchronization", Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on , vol., no., pp.1,12, 19-23 April 2010

Appendix

Address calculation for spill and restore operations

The VIUs spill and restore CBs by computing an address in global memory that is completely hidden to the programmer. The kernel is expecting to receive a pointer to global memory where some space has been allocated to spill the CBs. The pointer must be passed as a kernel argument. The spilling address is then computed as describes in Algorithm 2.

Each VIU will store one entire CB to global memory, so every thread will store a part of the considered CB to global memory. Each thread will then store a *TILE* of elements. *H* is then the starting element to store for each thread. In this algorithm, *CBV* represent the verticals CBs and *CBH* the horizontals CBs. We consider that a read from *CBV* is a load from the north CB, a write to *CBV* is a store to the south CB, a load from *CBH* is a load from the west CB and a write to *CBH* is store to the east CB.

As the grid of thread blocks will be executed in multiple passes, we identify them with registers, managed by the run time system. In this algorithm, *CPassx* represent the index of the current pass following the horizontal direction, and *CPassy* represent the index of the current pass following the vertical direction. *Passx* represent the total number of passes that needs to be done in the horizontal direction.

Two other registers are managed by the run time system, one that gives the current number of block synchronization within the current pass (*CspillNB*) and one that contains, at the end of the first pass, the total number of block synchronization for a pass (*spillNB*).

The pointer to the spilling area in global memory is given by *g_CB*. Lets consider *CB_SIZE* as the size of one CB, then *CBS_SIZE* is the size of four CBs, representing the size of the four border CBs of one direction.

PTX modifications

The modifications made to PTX are the following:

- **New hardware registers**
  - *regiu*: Identify if it is a IU
  - *regviriu*: Identify if it is a VIU
  - *regiudir*: Identify direction (north, west, south or east)
  - *regiuid*: Identify IU id
  - *regcpassx*: Contain the horizontal index of the current pass
  - *regpassx*: Contain the total number of horizontal passes required for the execution
  - *regcpassy*: Contain the vertical index of the current pass
  - *regspillnb*: Contain the total number of block synchronization of one pass
  - *regcspillnb* : Contain the number of block synchronization during the current pass

```
if VIU then
    int TILE = CB_SIZE/(blockDim.x*blockDim.y);
    int H = (threadIdx.y*blockDim.x + threadIdx.x) * TILE;
    if iu_dir is NORTH or WEST then
        while True do
            if iu_dir is NORTH then
                for int i = 0; i < TILE; i++ do
                    CBV[H + i] = g_CB[(Passx*(CPassy-1) +
                    CPassx)*CBS_SIZE*2*spillNB + CspillNB*CBS_SIZE*2 +
                    CB_SIZE*iu_id + H + i];
                end
            else
                for int i = 0; i < TILE; i++ do
                    CBH[H + i] = g_CB[((CPassx-1) + (Passx *
                    CPassy))*CBS_SIZE*2*spillNB + CspillNB*CBS_SIZE*2 +
                    CBS_SIZE + CB_SIZE*iu_id + H + i];
                end
            end
            block synchronization;
            if LAST then
                Break;
            end
        end
    else
        // iu_dir is SOUTH or EAST
        while True do
            block synchronization;
            if iu_dir is SOUTH then
                for int i = 0; i < TILE; i++ do
                    g_CB[(CPassx + CPassy*Passx)*CBS_SIZE*2*spillNB +
                    CspillNB*CBS_SIZE*2 + CB_SIZE*iu_id + H + i] = CBV[H
                    + i];
                end
            else
                for int i = 0; i < TILE; i++ do
                    g_CB[(CPassx + CPassy*Passx)*CBS_SIZE*2*spillNB +
                    CspillNB*CBS_SIZE*2 + CBS_SIZE + CB_SIZE*iu_id + H +
                    i] = CBH[H + i];
                end
            end
            if LAST then
                Break;
            end
        end
    end
else
    // Normal Code
end
```

Algorithm 2: Address calculation to spill and restore CBs by VIUs

  - *regstart*: Identify if the computation has started in the SMs or IUs
  - *reglast*: Identify if at least one SM has finished its current thread block for the current pass
- **New instructions**
  - *bar.blocksync 0;*: Invoke a SPU-wide block synchronization
  - *bar.unleashvius 0;*: Unleash only the VIUs that reached a block Synchronization and set *regstart* to *true*
  - *[st/ld].[cbv/cbh]*: Invoke a store (or load) to a vertical (or horizontal) CB.

Communication buffers implementation

The CBs are implemented as array of memory spaces. In order to cover the all grid of SMs, we need 40 CBS, but in order to implement the CB switching method described in section 3.1, we allocate an array of size 80.

Every SM has four pointers to its current CBs, *prevv* (top CB), *prevh* (left CB), *nextv* (bottom CB) and *nexth* (right CB). The initialisation of these four values is the following where i represents the SM id:

$$prevv = \begin{cases} i+40, & 0 <= i <= 15 \\ -1, & 16 <= i <= 19 \\ i-4+40, & 20 <= i <= 23 \\ -1, & 24 <= i <= 31 \end{cases}$$

$$prevh = \begin{cases} i/4+20+i+40, & 0 <= i <= 15 \\ -1, & 16 <= i <= 27 \\ 24+40, & i = 28 \\ 29+40, & i = 29 \\ 34+40, & i = 30 \\ 39+40, & i = 31 \end{cases}$$

$$nextv = \begin{cases} i+4, & 0 <= i <= 15 \\ i-16, & 16 <= i <= 19 \\ -1, & 20 <= i <= 31 \end{cases}$$

$$nexth = \begin{cases} i/4+20+i+1, & 0 <= i <= 15 \\ -1, & 16 <= i <= 23 \\ 20, & i = 24 \\ 25, & i = 25 \\ 30, & i = 26 \\ 35, & i = 27 \\ -1, & 28 <= i <= 31 \end{cases}$$

When any of these indexes is used, we compute the following index in the CBs array such that the SMs switch buffers every time step.

$$value \equiv ([nextv|nexth|prevv|prevh] + 40 \times regcspillnb) \mod 80$$