Computer Science
Technical Report

Colorado State University

The Stencil Processing Unit: GPGPU Done Right

Sanjay Rajopadhye
Colorado State University
Sanjay.Rajopadhye@colostate.edu

Guillaume Iooss
Colorado State University
iooss@cs.colostate.edu

Tomofumi Yuki
INRIA, Rennes, France
tomofumi.yuki@inria.fr

Dan Connors
University of Colorado, Denver
dan.connors@ucdenver.edu

1 March 2013

Colorado State University Technical Report CS-13-103

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

The Stencil Processing Unit: GPGPU Done Right

Sanjay Rajopadhye
Colorado State University
Sanjay.Rajopadhye@colostate.edu

Guillaume Iooss
Colorado State University
iooss@cs.colostate.edu

Tomofumi Yuki
INRIA, Rennes, France
tomofumi.yuki@inria.fr

Dan Connors
University of Colorado, Denver
dan.connors@ucdenver.edu

1 March 2013

Abstract

As computing moves to exascale, it will be dominated by energy-efficiency. We propose a new GPU-like accelerator called the Stencil Processing Unit (SPU), for implementing dense stencil computations in an energy-efficient manner. We address all the levels of the programming stack, from architecture, programming API, runtime system and compilation. First, a simple architectural innovation to current GPU architectures enables SPUs to have inter-processor communication between the coarse-grain processors (SMs or TPs). Despite this simplicity, the mere possibility of on-chip communication opens up many challenges, and makes the programming even more difficult than it currently is. We therefore provide a solution to the programming challenge by limiting access to the communication through a disciplined API and with a mechanism that can be statically checked. This allows us to propose simple modifications to existing runtime systems for GPUs to manage the execution of the new API on the SPU architecture. Based on our analytical models, we expect an order of magnitude reductions in the energy cost when stencil codes are implemented on the proposed architecture.

Keywords: co-design, GPGPU, exascale, energy efficiency, tiling

1 Introduction

General Purpose computing on Graphics Processing Units (GPGPU) has helped produce advances in a range of fields, including medicine, biology, chemistry, physics, and computational engineering by enabling the fastest running supercomputers in the world. Yet, this progress has been fortuitous, because of the realization that special purpose architectures (*accelerators*), developed for a very narrow domain, had broader applicability. The future of such accelerator systems represents both a unique opportunity and a challenge for the HPC and general-purpose computing communities. In exascale computing, the dominating cost metric will be energy, rather than speed, or possibly, energy in addition to speed. This paper addresses the evolution of General Purpose Computing on Graphics Processing Units (GPGPU) towards this metric.

1.1 Shoehorning GPGPU onto GPUs

Current GPUs and APIs for GPGPU have a huge drawback under an energy-centric cost metric. GPUs disallow communication/synchronization between the "large grain processors," i.e., the streaming multiprocessors (SMs) of NVIDIA, or ATI's thread processors (TP). Correspondingly, the programming APIs disallow "outer level" communication, e.g., *blocks* in a CUDA *grid* or cannot communicate between each other, and neither can *work-groups* in OpenCL.

For graphics processing—the original goals of GPUs—these choices were just right because (i) the application matches them, (ii) they simplify the programming model, and (iii) they provide portability. The justification is as follows.

First, graphics computations can be divided into "globally independent" parts or tasks, that need minimal or zero communication or synchronization (they are "pleasantly parallel," often called "embarrassingly parallel"). However, these tasks are internally compute-intensive and benefit from fine-grain, multi-threaded parallelism. Second, the corresponding programming model also has two levels of parallelism—just like the architecture—with independent tasks at the "outer" level, and an "inner" level of fine-grain parallelism that may communicate and synchronize. This model can be mastered relatively easily by computing and HPC specialists, as evidenced by the growing popularity of CUDA and OpenCL. It is also effective, as evidenced by the development of the ecosystem and support infrastructure that seeks to provide easier solutions to non-specialists through directives. Finally, a critical advantage of this choice is that the runtime scheduler can be made non-preemptive, leading to portable code that can run on any GPU-enabled card.

The success of GPGPU shows that many applications can be "massaged," (shoehorned) into such a two-level form. Usually, this involves making multiple kernel calls, and using global memory as a mechanism by which different thread-blocks communicate across the kernel calls. However, off-chip communications consume at least two orders of magnitude more energy [3, 10]. Even the energy consumption of on-chip communication grows linearly with distance. Hence, this shoehorning comes at a significant energy cost.

Such costs are unacceptable and avoidable. Evolving GPUs to non-graphics domains will need energy-efficient inter-SM communication/synchronization. One way to do this is to introduce caches and/or other on-chip memory. Efforts like the Echelon project [10] to extend GPUs in this direction are already under way, and Intel's MIC can be viewed as a "general purpose accelerator" to compete directly with GPGPU.

We seek a new niche middle ground—dense stencil computations, for which we extend accelerator functionality without losing many special purpose advantages of GPUs. Not surprisingly, we call our proposed device the SPU: Stencil Processing Unit.

1.2 Why Stencils

We focus on stencils for three important reasons.

- Stencils are *just the right* extension of graphics computing. Beyond shaders, GPUs are also used to efficiently simulate physical phenomena such as cloth and smoke [6], where the computational core is a stencil computation to solve the Navier-Stokes' equation. Although this is currently only a small part of game computation, we expect this to change as (i) architectures become available to do this efficiently and (ii) market demand emerges for more physical realism.[1]

[1] It is a chicken and egg problem, and we propose to lay the first egg.

- For general purpose computing, stencils are extremely important. One of the thirteen Berkeley dwarfs/motifs (see `http://view.eecs.berkeley.edu`), is "structured mesh computations:" nothing but stencils. Many instances of two other motifs, "dynamic programming" and "dense linear algebra," share similar dependence patterns. A recent surge of publications on the broad topic of stencil optimization [12, 2, 7, 13, 11, 16] ranging from optimization methods for implementing stencils on a vareity of target architectures, to domain specific languages (DSLs) and compilation systems for stencils, suggests that this importance has been noted by many researchers.

- We claim, and show in Sec 3 that the SPU can be built with very minimal tweaks to the GPU architecture. Since the tweaks are minimal, the SPU will will subsume GPU functionality as a special case. Moreover, the extensions to the programming API and run-time system are also simple.

There are a number of strategies to parallelize stencil computations for various platforms ranging from distributed memory machines, multi-core platforms, many-core accelerators like GPUs, Cell, etc., to FPGA based dedicated hardware implementations. They include various tiling schemes such as redundant computations through expanded halo regions, overlapped tiling, tiling with concurrent starts, general time-skewing, and cache-oblivious tiling schemes. While there has been much work on choosing the schemes optimally and also on selecting the parameters of the strategies in an optimal manner, the notion of optimality has so far only been speed. Little effort has been devoted to optimizing for energy, and we are not aware of any work that seeks to do this with a quantitative model.

1.3 Contributions

In this paper, we (i) develop energy efficient parallelization for stencils, and propose modifications to (ii) the architecture of current GPUs, (iii) the CUDA programming API, and (iv) the run-time scheduler. Specifically,

- We first propose a parallelization strategy for dense stencil computations over 1D and 2D data arrays (Section 2). We show its energy efficiency by comparing with other competing strategies using a simple quantitative model based on counting the number of off-chip memory accesses.

- We propose the architecture of SPUs (see Section 3), a simple, almost trivial, extension to GPU architectures. The SPU allows *local* communication and synchronization between SMs, enabling an SM to access the shared memory of *neighboring* SMs as cheaply—with comparable energy cost—as its own.

- Unfortunately, this opens Pandora's box in terms of impact on programming model, the run-time system, and the compilation problem. We therefore reopen the box in Section 4 and offer Hope, by insisting that inter-SM communication is exposed to the programming API only through a *disciplined but simple extension* to CUDA/OpenCL.

- We prove, in Section 5 that this indeed is safe by showing how to adapt current GPU run-time systems so that GPU tasks can continue to be non-preemptively scheduled through to completion—a feature that is essential to the portability and success of current GPU programming APIs.

2 Energy Efficient Stencil Parallelization

We first use matrix multiplication to show why energy efficiency requires on-chip communication. For this simple case, such communication can be achieved through caches. Next we show that dependent computations like stencils are much more complicated. We quantify the the energy cost of standard, wavefront based mechanisms for parallelizing such programs, and show how that our proposed strategy reduces the energy overhead by a factor of P, where P is the number of processors on chip, for 1D dependent computations, and by $\sqrt{P}$ for 2D stencils.

2.1 Matrix Multiplication

Consider the $N \times N$ matrix multiplication $C = AB$, using the standard tiled approach of the CUDA programming guide (we could use the Volkov algorithm [17] implemented in **`cublas`**, but the story would be similar). Each thread-block is responsible for computing a $b \times b$ submatrix of the result. This requires an $b \times N$ submatrix of A, and an $N \times b$ submatrix of B. Since the local memory is not large enough, we read, in a loop, smaller "data-blocks" of A and B, i.e., copy these blocks from global memory to shared memory. So the *memory footprint* of each threadblock[2] is $2Nb + b^2$ (the first term dominates). Let $n = \frac{N}{b}$, so we have an $n \times n$ grid of threadblocks. Note that n is, in general, much larger than P, the number of processors, so there is a level of virtualization that the GPU runtime system manages under the hood. Since there is no sharing of data between the threadblocks (indeed, even if two threadblocks were to be allocated to the *same* processor, the runtime system and the programming model *hides* this), the memory footprint of the entire program is $n^2 * 2Nb = \frac{2N^3}{b}$. The program performs $2N^3$ FLOPs, and therefore its *balance*—the compute-to-communicate ratio—is b.

We use a similar analysis to study energy efficiency. The total energy of the program is the sum of the energy spent in the computation, and that in memory accesses. It has been noted [3, 8] that the latter is two to three orders more than the former. Moreover, the energy to perform the computation, once data are on-chip, is unavoidable and essential. Hence the *energy overhead* of the program is, to a first approximation, simply the energy of the off-chip data transfers, i.e., the memory footprint of the entire program, $\frac{2N^3}{b}$.

In order to improve this, we need to reduce the footprint, or equivalently, increase the balance. Note that the matrix multiplication *algorithm* has an arbitrarily scalable balance (N^3 to N^2), but our implementation only had a balance of b. The inability to share data between SMs is a key reason. Assume that there is some architectural support for communication *between* SMs. Say there are P SMs, arranged in a square $p \times p$ grid (so $p = \sqrt{P}$). Now, two processors on the same row need the same elements of A, and those on the same column similarly share the same elements of B. To analyze this, we think of this as making all processors *collectively* responsible for *cooperatively* computing a $pb \times pb$ block of C. Since n is usually much larger than p, we need $\frac{N^2}{Pb^2}$ *passes* to execute the algorithm, each with a footprint of $2Npb$. The total memory footprint now becomes $\frac{2N^3}{pb}$, and the balance improves to pb, a $\sqrt{P}$-fold improvement. If the number of SMs approaches the hundreds, this would yield an order of magnitude reduction in the number of off-chip accesses. In fact, In fact, recent GPUs already provide the necessary architectural support for this in the form of caches.

Fewer accesses to global memory may have an additional secondary benefit. Although this may not happen for a well tuned matrix multiplication program, fewer accesses may even speed up the program, and this further contributes to reduction in energy of all the system

[2]The total number of memory accesses performed by each threadblock.

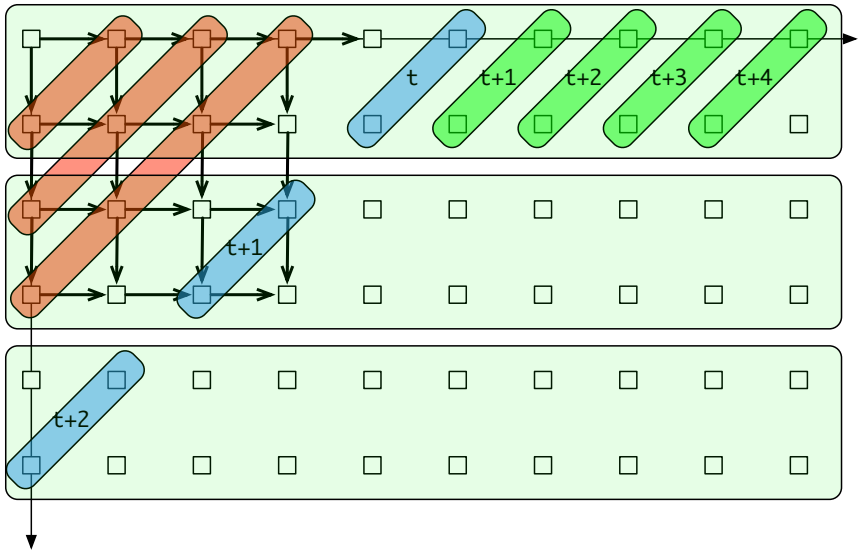


Figure 1: Pipeline parallelization of a simple 2D computation over an $N \times M$ domain. Each square represents an $x \times y$ tile, executed as as single threadblock; all inter-tile edges are not shown, to avoid clutter. The (red) diagonal bands represent sets of tiles executed in a single kernel call. Because the number of physical processors is limited, the GPU run-time system partitions the red bands into blue bands (labeled $t, \ldots t+2$) and serially executes them. The total number of off-chip accesses is $2NM\frac{(x+y)}{xy}$. In an alternative parallelization, the whole computation is divided into passes (light green swaths) and the wavefront is restricted to just the pass: after the blue band at time step t, the green sequence of bands is executed. Now, the total number of off-chip accesses is $\frac{2NM}{Px}$, a *factor* of P reduction.

components (especially that due to static power). There is potential for a quadratic improvement in (a dominant part of) the GFLOps/joule metric. The benefits of reducing memory footprint are thus significant.

2.2 Dependent Computations

Consider (see Fig. 1) the simplest example with inter-iteration dependences: an $N \times M$, 2D iteration space with canonic dependences (the point $\langle i, j \rangle$ depends on its north and west neighbors, $\langle i, j-1 \rangle$ and $\langle i-1, j \rangle$, respectively). To parallelize this, we first tile it using $x \times y$ tiles, yielding an $n \times m$ *tile graph*, where $n = \frac{N}{x}$ and $m = \frac{M}{y}$.

Each tile is executed as a single threadblock, and "wavefronts" of tiles are executed by a sequence of kernel calls (red bands in Fig. 1). However, the GPU has only a limited number of processors, so the run-time system partitions each wavefront and executes them serially, as illustrated by the blue sequence of bands marked $t, t+1 \ldots$ in Fig. 1).

Let us quantify the energy overhead of this. The total number of tiles/threadblocks executed is nm. Each tile performs $x + y$ reads and $x + y$ writes, and therefore has a footprint of $2(x + y)$. Hence, the memory footprint of the program is $2NM\frac{(x+y)}{xy}$.

Now consider an alternative parallelization where we first partition the tile graph into *passes* (light green swaths in Fig. 1) of height P tiles, and execute each pass with a wavefront schedule—darker green bands marked with $t+1, t+2 \ldots$). There are $\frac{N}{xP}$ passes, and each one

has (as we justify below) a memory footprint of $2(xP + M)$, the perimeter of the pass. Hence the memory footprint of the program is $\frac{2NM}{xP} + 2N$, which is dominated by the first term. This yields at least a P *fold* reduction!

To justify our claim that a pass can be completely executed with a footprint just equal to its perimeter, we must provide an on-chip communication mechanism between processors (see Section 3). However, that alone is not enough. We see from Fig. 1 that the parallelization scheme cannot be simply specified by the sequence of red wavefronts, and unfortunately the green wavefronts have a size, P, which is very machine dependent. Hence the challenge is to expose the newly introduced communication capabilities of the architecture to the programmer through a simple API, and yet retain the portability of the program. We explain these extensions, as well as the modifications to the run-time system in Sections 4 and 5. However, before proceeding further, we show why this multiple pass parallelization strategy is also applicable to dense stencils.

2.3 Stencils are not so easy

Given sets $\mathcal{N}$ and $\mathcal{N}'$ of vectors in Z^2 called *neighborhoods* for the previous and current time steps, respectively, and associated sets of *coefficients*, $\mathcal{W}$ and $\mathcal{W}'$, a 2-dimensional dense stencil computation (without convergence tests) consists of the iterative evaluation of an expression for all points in a domain $D = \{i, j, t \mid 0 \leq (i, j) \leq N; 0 \leq t < T\}$

$$X^t[z] = \sum_{a \in \mathcal{N}} \mathcal{W}_a X^{t-1}[z + a] + \sum_{a \in \mathcal{N}'} \mathcal{W}_a X^t[z + a]$$

assuming that appropriate boundary/initial values are provided for points outside D. In the Jacobi stencils $\mathcal{N}'$ is empty, whereas in the Gauss-Seidel stencils $\mathcal{N}$ and $\mathcal{N}'$ have roughly equal cardinality.

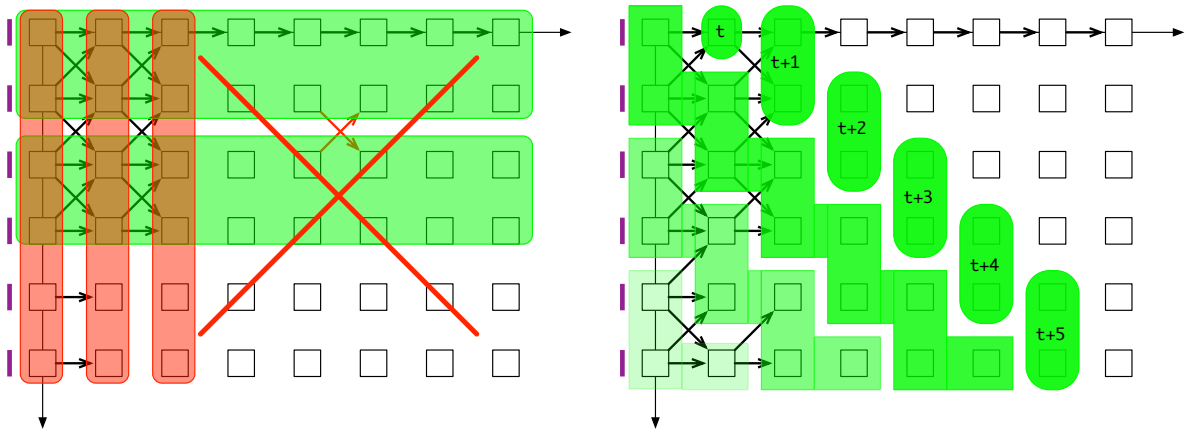
In the remainder of this paper, we assume Jacobi stencils, although our arguments carry over directly to other stencils. We let $\beta = |\mathcal{N}|$. We shall analyze four schemes to parallelize such stencils, from the perspective of off-chip data access. Fig. 2 pictorially illustrates the case for a 1D data array, that is easier to visualize.

2.3.1 Standard Parallelization with Redundant Computation

In the most direct scheme, the $N \times N$ grid of data points is tiled into $n \times n$ grid of $b \times b$ tiles, where $N = nb$. Again, n is much larger than p, so data is copied from global memory for each tile at each iteration of the outer t loop. This has a memory footprint of $2N^2T$, leading to a very poor computation balance of only $\frac{\beta}{2}$. A well known way to improve this is by using larger "halo regions," at the expense of a certain amount of redundant computation [15]. The corresponding gain is usually by a small constant factor before memory capacity and the additional work produce diminishing returns.

2.3.2 Cache Oblivious Tiling

Another important approach for tiling stencil-like computations is cache-oblivious tiling [4, 5]. These methods recursively tile the iteration space such that at some level of the tile, the memory foot print is small enough to fit into a given cache. Therefore, the behavior of the resulting program with respect to data locality is independent of—oblivious to—the cache size. For the purpose of our analysis, we assume cache-oblivious methods to behave the same



(a) Passes are illegal **(b) Legal passes**

Figure 2: Stencil programs are trickier. Each square is a tile which is fine-grain parallelized (many techniques for this are available) by a threadblock. Across tiles, each one needs data from its north-west, west and south-west neighbors. The parallelization and corresponding CUDA kernels (red bands) are now seemingly easier–all the tiles in a column are independent. However, executing this in multiple passes as shown in the diagram on the left is illegal (cyclic dependences between passes). However, oblique passes as shown in the right diagram are legal provided the passes are executed from the bottom to the top. Furthermore, each pass can be easilyparallelized (green ovals marked t, $t+1$. ...). The standard CUDA parallelization (left diagram, red bands) makes $2NM\frac{(x+y)}{xy}$ off-chip accesses, and the parallelization on the right makes only $\frac{2NM}{Px}$, again, a *factor* of P reduction.

as tiled programs using the perfect (i.e., optimal) tile sizes. There are other parameters and potential inefficiencies in cache-oblivious methods, e.g., recursive program structure, and this serves as an upper bound estimate of its performance. Hence they will be no worse than the nest case scenario of the time-skewed approach described below, both of which suffer from the inherent architectural limitations.

2.3.3 Time Skewing with Wavefront Parallelization

Time skewing [20, 19, 1] is a well known technique that allows tiling to have scalable granularity. Applying it to our stencil consists of first skewing the two spatial dimensions, i and j of the iteration space, by a factor d with respect to the t dimension[3], where d is the maximum component of the vectors $a \in \mathcal{N}$. After this preprocessing transformation, the iteration space becomes a parallelepiped, and it is legal to tile it with tile boundaries perpendicular to the unit vectors (orthogonal tiling). We assume, for the sake of simplicity, cubic, $b \times b \times b$ tiles. Furthermore, it is also well known that dependences in the resulting *tile graph*—the graph whose nodes are the tiles and whose edges are dependences between the tiles—are binary linear combinations of the unit vectors, and the tile graph admits a simple *wavefront* parallelization [9].

Each tile has a volume b^3 and performs βb^2 multiply-add operations. Along the four "spa-

[3]There may be different factors, d_1 and d_2, in each of the spatial dimensions. This just complicates the analysis, without contributing anything substantial.

tial" boundaries it has dependences of length d and along the t dimension, the dependences are of unit length, so the amount of data it needs is $(4d+2)b^2$ and the computational balance of each tile is αb for a constant $\alpha = \frac{2\beta}{4d+2}$. Just as in matrix multiplication, there is an upper bound on how large b can be made, based on tile-level resource constraints: size of the shared memory, number of available registers, etc.

There is a fundamental reason why the balance is bounded by b. The architecture and programming model prohibit tiles belonging to different wavefronts from communicating directly (they are in *different* kernel calls).

2.3.4 Time Skewing with On-chip Communication

We now explain how to overcome these constraints. First, allowing SMs the ability to communicate will enable the tiles within a wavefront to communicate between themselves, thereby reducing some of the constant terms in α. However, this alone is not sufficient. Inability to communicate *across* kernel calls is the critical limiter here.

We seek to organize the parallelization of the tile graph into *partitioned wavefronts* where a set of (virtual) processors in the parallelization can execute their wavefronts *all the way to completion* before other processors even start. Such a partitioning is well known from the 30 year old literature on systolic array synthesis, and is called LPGS partitioning [14] for "locally parallel, globally sequential." It is also easy to prove that this can be achieved if, after the parallelizing transformation, the projection of the inter-tile dependences on the virtual processor dimensions lie in the first quadrant. Furthermore, this is always possible for a dependence graph whose original dependences are in the first orthant, as is the case for our tile graph.

The parallelization that we propose can therefore be described as follows. Organize the P processors into a $p \times p$ grid and have this processor array sweep through the wavefront-parallelized time-skewed graph in multiple passes. If the parallelization needs an $n \times n$ grid of (virtual) processors, the parallelization uses n^2/P passes. Only the data between passes needs to be communicated to/from global memory. Hence the memory footprint is reduced by a factor of p.

2.4 A Question of Balance

Balance is a critical metric [20] and is used in many analyses. It may be defined for a given program/parallelization, for an algorithm (e.g., matrix multiplication has a balance of N: N^3 FLOPs to N^2 floats, which is arbitrarily scalable), or even for an architecture/machine (peak FLOPS/sec to peak bandwidth). As an example, if program balance can match or exceed the machine balance, the implementation can be made compute bound, rather than memory bound. This is indeed the case for matrix multiplication. This is why appropriately tuned versions of the tiled program can achieve close to the machine peak performance.

Williams et al. [18] suggest how balance (which they call *arithmetic intensity*) can be used to optimize "performance" (i.e., speed) on multi-cores. In their formulation it is viewed as a constraint to be satisfied—if the balance can be improved beyond a certain threshold, the program becomes compute bound—but the objective function of the optimization remains the execution time. Our position is that in exascale, balance must factor into the objective function.

Balance provides us with a good model for energy efficiency. Of course, improving balance by a factor x may not directly translate to x-fold reduction in total energy due to Amdahl's

Law like effects—global memory access is only a part of the total energy budget. However, let us view the total energy cost as the sum of two terms, α for the cost of computing the results once all the necessary data has been copied into shared memory, plus β, the cost of this copy. The β term is a direct measure of the *energy overhead*, and the balance simply reports it relative to the operation cost. Therefore, we claim that our proposed solution yields a $\sqrt{P}$-fold reduction of the energy overhead. With 100-processor GPU-like architectures already available or announced, this is definitely nothing to sneeze at.

Furthermore, static energy is increasingly becoming the critical component. Now, note that the improvement in balance affects not only the energy, but may also affect the speed of the program—local accesses are also one to two orders of magnitude *faster* than global. Since energy is the product of power and time, improving balance may reduce the energy consumed by other components of the energy budget.

3 The SPU Architecture

Two concerns drive our architectural goals. First the "general purpose" goals cannot afford to trump the primary one—graphics. Our architectural extensions should (i) be minimal, and (ii) not interfere with core GPU functions. The second is that the changes must provide extremely energy-efficient communication and synchronization. The processors in the SPU are called *Stencil Processors, SP* and we refer to tightly connected groups of scalar processors within GPU processors as Streaming Multiprocessors (SMs) which borrows from Nvidia original designation.

3.1 Mesh Topology

Guided by these goals, we choose a mesh topology for inter-SP communication. SMs on current GPUs are already laid out in a 2D grid, and it is easy to assign $\langle x, y \rangle$ coordinates to each SM. Thus, local communication in such a mesh translates to low-energy. Because of this, our communication topology will be only nearest neighbor, any "long" communication will have to be achieved through a "store-and-forward" mechanism, and will have to be explicitly programmed.

Figure 3 illustrates this. Each processor has four *communication buffers* (CBs) one for each of the North, East, West, South (NEWS) directions. The SMs in current GPUs already have shared memory for their fine-grain SIMD parallel/vector units, and this memory is already partitioned into multiple banks to provide adequate bandwidth. Therefore, it may be possible to use some of the banks, designated as "boundary" banks, as communication buffers. Alternatively, they may be implemented with separate, dedicated banks of memory blocks. The CBs are shared between the fine-grain threads on the SPU (just like shared memory on current GPUs) and reads/writes from/to the buffers may have to be appropriately synchronized. However, adequate mechanisms already exist on GPUs to provide this, especially if they are logically viewed as specialized regions of the shared memory.

3.2 Instantaneous Communication

To communicate the data in the CBs, we use a simple toggling arbitration mechanism (see Figure 3, right) to instantaneously transfer the "ownership" of a CB from one SP to the next. The mechanism is again, trivially simple: at any time, an *arbitration bit* between two adjacent SPs controls which of them owns each one of a pair of CBs. The memory system is oblivious to

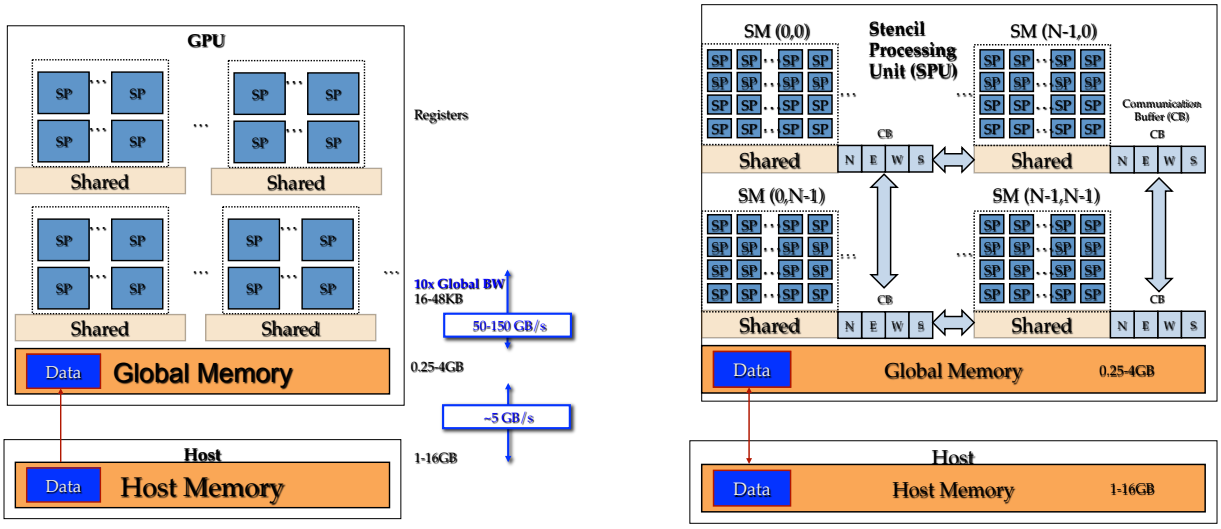


Figure 3: The SPU architecture (right) is a simple extension (lightly shaded blue parts are added) of the standard GPU (left). These additional communication buffers allow a mesh connected mechanism for inter processor communication

this—any memory access is automatically transfered to the bank that the processor "currently owns." When a communicaton is desired, this arbitration bit is toggled, thereby achieving a complete exchange of (all the contents of) the CBs. This happens at some synchronization point during execution. From this time on, until the next synchronization, memory accesses to the CBs refer to values that the neighboring SP may have written into this bank prior to the synchronization.

This mechanism has minimal disruption and is energy efficient. In fact, for standard GPU applications, our mechanism could be completely bypassed, and therefore, if power gating were used, we expect its overhead in time and energy to be essentially, zero, and the area overhead to be small.

Next, we discuss how these mechanism are exposed to the programmer, and later on, the implications of these choices.

4 The programming API

There are essentially two choices in exposing the communication fabric to the programmer: (i) how to access data in the CBs, i.e., how to store to and retrieve from them, and (ii) how to orchestrate the transfers. We now discuss them.

The first choice about access mechanisms is a direct consequence of our architectural choices. Since CBs are logically to be viewed as sub-regions in shared memory, our API allows CUDA variables that are declared as **`shared`** to have an additional **`transferable`** annotation, to indicate that these variables are to be allocated to CBs. An additional attribute/keyword indicates the (NEWS) direction of the transfer.

Orchestrating the transfers: Pandora's box

There are really two choices here. We could allow individual threads to dynamically toggle the arbitration bits in a possibly asynchronous, completely dynamic manner. This would lead to

anarchy. Alternatively, and this is our choice, we could allow the bits to be toggled through a "global" command involving all the threads. We therefore have a synchronized transfer, where the threads in a block must first all arrive at a common synchronization point. This simple extension to a CUDA **`syncthreads`** is called **`syncblocks`**. When the control flow arrives at a **`syncblocks`**, the thread

- waits until *all the threads in all the blocks* arrive at this point as in a **`syncthreads`**,
- Next, the arbitration bits are toggled so that the CBs get exchanged, and
- finally, all threads continue execution.

Note first, that this is a *global synchronization* across all the threadblocks in the grid. And that this opens Pandora's box. This simple and seemingly minimal extension breaks a key GPU requirement.

Let us return to our stencil computation, consider our *first* standard parallelization, possibly with increased halo and redundant computation. To execute this on the SPU, we would want to tweak the standard GPGPU code so that we make, instead of $\frac{T}{h}$ kernel calls communicating through global memory, a *single kernel call*, that makes $\frac{T}{h}$ calls to **`syncblocks`**. We would expect to use the CBs of the SPU to do the typical "halo exchange," and avoid using the global memory for this communication.

However, this would either lead to deadlock or defeat the entire purpose!! The main motivation behind GPGPU and the abstraction provided by CUDA and/or OpenCL is that a GPGPU program must port to any CUDA-enabled card with no changes. In order to achieve this, the threadblocks in a grid are actually viewed as virtual processors, that are mapped to the physical SMs on the card by a runtime system. For maximum efficiency, the scheduler almost always *non-preemptively* executes individual threadblocks through to completion. This is efficient and perfectly legal in the CUDA/OpenCL programming model.

If the SPU code proposed above is executed non-preemptively by such a run time, it would produce incorrect answers–the halo regions read by a threadblock whose neighbor is not executing concurrently, will be garbage!

5 Hope and Extended Runtime

This section provides hope through discipline.

There are two ways to solve the problem. The first is to extend the runtime with non-preemptive scheduling, possibly by interleaving the execution of threadblocks. This is itself a daunting task, made especially difficult by the fact that considerable prior GPGPU infrastructure has been developed, much of it proprietary, and details are often unavailable. Second, the potential gains of on-chip communication may be lost with a heavy runtime system. Moreover, it is likely to lead to significant programming difficulties. Finally, this is really not a solution, because it reverts back to global memory transfers that we are seeking to avert—the threadblocks would have to be swapped in and out at each **`syncblocks`** and this would require their state to be saved and restored.

Our solution is to impose a discipline to ensure that a non-preemptive scheduler remains safe, by what we call *unidirectional communication*. This gives sufficient conditions for safety—in exchange for limiting the communications to be unidirectional, we can guarantee the legality of non-preemptive scheduling, with a few simple modifications.

```
__shared__ transferable N2S float Btrans[BLOCK_SIZE][BLOCK_SIZE];
__shared__ transferable W2E float Atrans[BLOCK_SIZE][BLOCK_SIZE];

 for (t=0; t<bx+by; t++)
 __syncblocks;                    // Dummy blocks (most can be optimized away)
 for (int t=bx+by; t <= Maxbz+bx+by; t++) {
    // Computation: code for internal blocks only
    // (special case for boundary blocks are not shown)
    for (int k=0; k<BLOCK_SIZE; ++k){
      Csub += Atrans.in (ty,k)*Btrans.in(k,tx);
     }
    // Transmitting data
    __synchthreads();
    Atrans.out(ty,tx) = Atrans.in(ty,tx);
    Btrans.out(ty,tx) = Btrans.in (ty,tx);
 __syncblocks;                              // Subsumes a syncthreads
 }
 for (t=Maxbz+bx+by+1; t < Maxt; t++)
 __syncblocks;                     // Dummy blocks (most can be optimized away)
```

Figure 4: Matrix Multiplication in extended CUDA for the SPU

In order to achieve this, we impose a further restriction on the annotation that we allow for **`transferrable`** variables. The direction for the transfer is not an arbitrary one of the four NEWS, values, but a specific *pair* that indicates the direction. Moreover, all accesses to `transferrable` variables is with additional qualifiers—**`.in`** to read the variables in neighboring grid blocks, and **`.out`** to write the values that neighboring blocks will access later. For example, the following declaration indicates that `Btrans` is an array that will be transferred *from* north neighbor *to* the south.

```
__shared__ transferable N2S float Btrans[BSIZE][BSIZE];
```

The compiler must check that these constraints, across all the variables declared in the program do not lead to any cycles. For example, in a CUDA program with a 1-D grid, no other direction should be allowed: the following program will be flagged as erroneous:

```
__shared__ transferable N2S float Btrans[BSIZE][BSIZE];
__shared__ transferable S2N float Atrans[BSIZE][BSIZE];
```

Even if the declaration of `Atrans` had the annotation, `E2W` it would be incorrect. In a CUDA program with a 2-D grid of threadblocks, exactly two orthogonal and non-conflicting communication directions should be present. Given a legal set of annotations, the runtime system can now determine a non-preemptive schedule. Of course a number of different choices are possible (e.g., row-major, column-major or even blocked or wavefront orders, but this can be predetermined by the runtime system.

The second important additional aspect that the runtime has to do is to manage the communication buffers and transferrable arrays through *spilling to global memory*. At any time during the execution, the system maintains a set $\mathcal{S}$ of the currently executing threadblocks. Whenever a **`syncblocks`** is executed, all the arbitration bits are toggled. In addition, for all threadblocks that need to *write to* another (using `Var.out` on the left hand side of an assignment) that is not in $\mathcal{S}$, the CB is *spilled* to global memory in a system-reserved area.

Similarly, a read (`Var.in`) access from a threadblock that is not in $\mathcal{S}$ causes the runtime system to refresh the buffers from previously spilled data.

We expect that in most cases the system memory for spills can be statically predicted, and can even be optimized and tuned through prefefetching.

6 Conclusions

In this paper we have essentially argued that GPUs are not right for GPGPU (*general purpose* computing). The moment there are dependences between the computations that are being parallelized, the lack of on-chip inter-processor communication, means that all dependent data that must be communicated must pass through global memory, and this is hugely energy inefficient, and will not be acceptable in the exascale era.

The common wisdom is that these problems can be resolved by adding caches, but this too has a significant energy cost—although they avoid off-chip accesses, they are nevertheless shared resources on the chip, and imply long communications. We therefore argue that approaches that bolt general purpose solutions to GPUs would "throw away the baby with the bathwater."

The solution that we propose is a special purpose (domain specific) architecture, that is dedicated to a niche domain: dense stencil computations. We have shown that our solution can yield up to an order of magnitude reduction in the energy overhead of such computations, and we have argued how the solution is simple to implement, although it needs us to revisit the entire run-time stack from architecture, to programming API and runtime system. This is our ongoing work.

References

[1] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal semi-oblique tiling. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):944–960, Sept 2003.

[2] Matthias Christen, Olaf Schenk, and Yifeng Cui. Patus for convenient high-performance stencils: evaluation in earthquake simulations. In *Proceedings of the 2012 International Conference on High Performance Computing, Networking, Storage and Analysis*, page 11, 2012.

[3] W. J. Dally, J. Balfour, D. Black-Shaffer, R. C. Chen, J. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *IEEE Computer*, 41(7):27–32, July 2008.

[4] M. Frigo and V. Strumpen. Cache Oblivious Stencil Computations. In *ICS 2005: International Conference on Supercomputing*, pages 361–366, Cambridge, MA, June 2005.

[5] M. Frigo and V. Strumpen. The Cache Complexity of Multithreaded Cache Oblivious Algorithms. *Theory of Computing Systems*, 45(2):203–233, 2009.

[6] M. Harris. *Fast Fluid Dynamics Simulation on the GPU*, chapter 38. Morgan Kauffman, 2008. in GPU Computing Gems.

[7] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320, 2012.

[8] S. Hong and H. Kim. An integrated GPU power and performance model. In *ISCA*, pages 280–289, 2010.

[9] F. Irigoin and R. Triolet. Supernode partitioning. In *15th ACM Symposium on Principles of Programming Languages*, pages 319–328. ACM, Jan 1988.

[10] S. Keckler. GPU Computing and road to Extreme Scale Parallel Systems. Rich Report on YouTube, Feb 2012. A report on the NVIDIA's Director of Architecture Research.

[11] Wang Luzhou, Kentaro Sano, and Satoru Yamamoto. Domain-specific language and compiler for stencil computation on fpga-based systolic computational-memory array. *Reconfigurable Computing: Architectures, Tools and Applications*, pages 26–39, 2012.

[12] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[13] Richard Membarth, Frank Hannig, Jürgen Teich, and Harald Köstler. Towards domain-specific computing for stencil codes in hpc. In *Proceedings of the 2nd International Workshop on Domain-Specfic Languages and High-Level Frameworks for High Performance Computing*, 2012.

[14] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transaction on Computers*, C-35(1):1–12, January 1986.

[15] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill, 2004. ISBN 0-07-282256-2.

[16] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C-K. Luk, and C. E. Leiserson. The Pochoir Stencil Compiler. In *SPAA: ACM Symposium on Parallel Algorithms and Architectures*, San Jose, CA, 2011.

[17] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *ACM/IEEE Conference on Supercomputing (SC08)*, 2008. (best student paper award).

[18] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun.ACM*, 52(4):65–76, April 2009.

[19] D. Wonnacott. Time skewing for parallel computers. In *LCPC 1999: 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 477–480. Springer Verlag, 1999. See `http://ipdps.eece.unm.edu/2000/papers/wonnacott.pdf` for a more detailed version.

[20] D. Wonnacott. Achieving scalable locality with time skewing. *IJPP: International Journal of Parallel Programming*, 30(3):181–221, 2002.