Single-Graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs

Dani Voitsechov[1] and Yoav Etsion[1,2]
Electrical Engineering[1] Computer Science[2]
Technion - Israel Institute of Technology
{dani,yetsion}@tce.technion.ac.il

Abstract

We present the single-graph multiple-flows *(SGMF) architecture that combines coarse-grain reconfigurable computing with dynamic dataflow to deliver massive thread-level parallelism. The CUDA-compatible SGMF architecture is positioned as an energy efficient design alternative for GPGPUs.*

The architecture maps a compute kernel, represented as a dataflow graph, onto a coarse-grain reconfigurable fabric composed of a grid of interconnected functional units. Each unit dynamically schedules instances of the same static instruction originating from different CUDA threads. The dynamically scheduled functional units enable streaming the data of multiple threads (or graph flows, in SGMF parlance) through the grid. The combination of statically mapped instructions and direct communication between functional units obviate the need for a full instruction pipeline and a centralized register file, whose energy overheads burden GPGPUs.

We show that the SGMF architecture delivers performance comparable to that of contemporary GPGPUs while consuming ∼57% less energy on average.

1. Introduction

GPGPUs are gaining track as mainstream processors for scientific computing [20, 28]. These processors deliver high computational throughput and are highly power efficient (in terms of FLOPs/Watt) [21]. Nevertheless, existing GPGPUs employ a von-Neumann compute engine and, therefore, suffer from the model's power inefficiencies.

Control-based von-Neumann architectures are tuned to execute a dynamic, sequential stream of instructions that communicate through explicit storage (register file and memory). For GPGPUs, this means that each dynamic instruction must be fetched and decoded, even though programs mostly iterate over small static portions of the code. Furthermore, because explicit storage is the only channel for communicating data between instructions, intermediate results are thus transferred repeatedly between the functional units and the register file.

These inefficiencies dramatically reduce the energy efficiency of modern GPGPUs (as well as that of general purpose von-Neumann processors [15, 19]). For example, recent GPGPUs spend only about 10-20% of their dynamic energy on computing instructions but spend up to 30% of their power on the instruction pipeline and the register file [16, 25, 31].

Interestingly, the same factors that adversely affect the energy efficiency of GPGPUs make a coarse-grain reconfigurable fabric (CGRF) energy efficient. A CGRF distributes the computation across a fabric of functional units. The compute operations (nodes in the dataflow graph) are statically mapped to functional units, and the interconnect is configured to transfer values between functional units based on the graph's connectivity. These features make a CGRF both energy efficient [11, 27] and able to deliver good single-threaded performance [10, 11, 27, 36] (in comparison with von-Neumann processors). Distributed control and static instruction mapping obviate the instruction pipeline, and direct communication between functional units obviates the centralized register file.

In this paper, we present a GPGPU-compatible architecture that combines a CGRF with a dynamic dataflow execution model to accelerate execution throughput of massively thread-parallel code. This design goal differs from previous CGRF designs, which favored single-threaded performance over multi-threaded throughput [10–12, 18, 27, 36, 39].

The proposed *single-graph multiple-flows* (SGMF) processor core is composed of a grid of interconnected functional units onto which compute kernels are mapped. An SGMF processor can be composed of multiple SGMF cores. The SGMF core employs the tagged-token dynamic dataflow model to concurrently execute multiple threads. Every thread is tagged, and the tag is attached to all in-flight data values associated with the thread. Tagging the values transferred between functional units allows threads to bypass memory-stalled threads and execute out-of-order, while data-dependencies inside each thread are maintained. This increases overall utilization by allowing unblocked threads to use idle functional units.

SGMF cores thus benefits from two of types parallelism: *a)* pipelining the instructions of individual threads through the CGRF; and *b)* simultaneous multi-threading (SMT) [42, 43], achieved by dynamically scheduling instructions and allowing blocked threads to be bypassed. Furthermore, the execution model enables the processor to fetch and decode instructions once, when the program is loaded, and directly communicates values between its functional units. We show that the dynamically scheduled spatial design is more energy efficient than von-Neumann-based GPGPUs.

The paper presents the dynamic SGMF execution model for CGRFs. We examine the challenges of integrating dynamic dataflow into CGRFs. We design and implement an SGMF processor and demonstrate that, on average, is consumes ∼57% less energy than Nvidia's Fermi GPGPU.

The remainder of this paper is organized as follows: Sec-

978-1-4799-4394-4/14/$31.00

tion 2 presents the SGMF execution model and is followed by a discussion of related work in Section 3. Section 4 discussed the challenges in designing a multi-threaded CGRF and how we addressed them, leading to Section 5 that presents the SGMF architecture. Finally, we present the evaluation of the proposed architecture in Section 7 and conclude in Section 8.

2. Execution Model: Spatial Computing with Dynamic Dataflow

The SGMF execution model adapts a CGRF to deliver massive thread-level parallelism and supports the CUDA [29] and OpenCL [32] programming models. The reconfigurable grid comprises compute units (similar in design to CUDA cores [26]), control units, load/store (LDST) units and special compute units (for long latency operations such as division and square root). Units are interconnected using a statically routed interconnect. The architecture is described in Section 5.

The SGMF compiler breaks the CUDA/OpenCL kernels into dataflow graphs and integrates the control flow of the original kernel to produce a control-data-flow-graph (CDFG) [33]. Adding the control into the graph eliminates the need for centralized control logic, because control tokens are passed through the interconnect as arguments to the compute units. The CDFG is mapped to the grid and the interconnect is statically configured to match the CDFG connectivity.

2.1. Pipelined parallelism

Functional units in the grid execute (or *fire*, in dataflow parlance) once their input operands are ready, regardless of the state of the other units. Once the result is sent as a token to the destination node, a functional unit is available to process new input operands, or tokens, belonging to a different thread. This enables pipelining multiple threads through the grid — different threads execute at different levels of the dataflow graph at the same time. The number of threads that can execute in parallel using pipelining is proportional to the length of the critical path in the dataflow graph.

Thread execution begins at the edges of the grid, with units that represent constants in the CDFG. These units feed the functional units, which consist primarily of compute and load/store (LDST) units. These *constant units* initiate the creation of new threads by sending tokens with their constant values to their destination units. The constant units can thereby introduce new threads into the grid at every cycle during which their destination nodes can receive them.

Thread pipelining is illustrated in Figure 1. In the first cycle, the constant units send the tokens that correspond to thread #1, namely a_1, b_1, c_1 and d_1. These tokens are processed by the compute units in the second cycle, at which time the constant units send tokens a_2, b_2, c_2 and d_2 associated with thread #2. Finally, once thread #1's tokens reach the multiplication unit, the constant units send the tokens for thread #3.

Importantly, input tokens may arrive to the units at different times. We therefore add *token buffers* to each node, in which received tokens wait until all the input tokens for an operation are available, and the operation can execute.

Thread pipelining requires that, in the common case, units in the CGRF be able to consume and process new tokens on every cycle. The units must therefore either incur a single-cycle processing latency or be fully pipelined. Some operations, however, cannot provide these guarantees. These include memory operations and compound, non-pipelined computations (division and square root operations). We therefore employ dynamic dataflow to relax this restriction.

2.2. Dynamic dataflow

Non-pipelined operations, and in particular those whose latency is unpredictable, may severely degrade the utilization of the SGMF core. For example, memory operations might stall on cache misses and block all threads in the pipeline. Furthermore, arithmetic operations such as division (both integer and floating point) and square root are inherently non-pipelined. Moreover, their latency varies and depends on the operands. This section describes how the SGMF execution model uses dynamic dataflow to tolerate long latency operations by allowing stalled threads to be bypassed by ready ones.

The SGMF execution model employs dynamic dataflow semantics to dynamically schedule instructions in the functional units. Dynamic dataflow [7, 13], and specifically tagged token dataflow [1], dynamically reorders instructions coming from different threads according to the availability of their input tokens (in a manner that resembles simultaneous multi-threading (SMT) [42, 43]). This enables instructions in one thread to bypass blocked instructions in other threads. Threads thus execute out-of-order to improve the overall grid utilization.

We demonstrate thread bypassing using LDST units. Nevertheless, the same principle are applied in special compute units, which handle non-pipelined arithmetic operations and are described in Section 4.2.

LDST units may return result tokens out-of-order, as demonstrated in Figure 2. In the first cycle, a load operation is executed on behalf of thread #1, which experiences a cache miss. The memory operation then waits until data is retrieved from the next level cache. During that time, the LDST unit can execute a load operation on behalf of thread #2, which hits in the cache. The retrieved memory value for the thread #2 is thus sent to the destination node before the thread #1's value is available. Note that a similar scenario may also occur with hits to a non-uniform cache architecture (NUCA).

Since tokens may arrive out-of-order, CGRF units need to be able to match input tokens associated with a specific thread. Each thread is therefore associated with a unique thread identifier (TID), and tokens are tagged with their respective TID. To obviate associative tag matching in the functional units, the token buffer is implemented as a direct-mapped structure and used as an *explicit token store* [34]. Token buffers are indexed by the lower bits of the TID, and a functional unit can

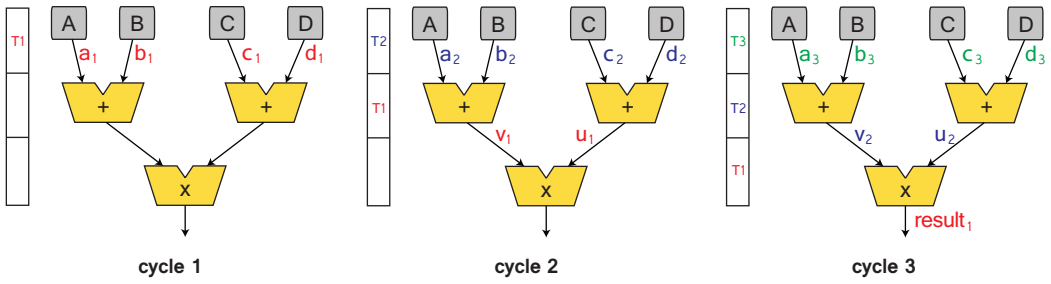


Figure 1: Thread pipelining example

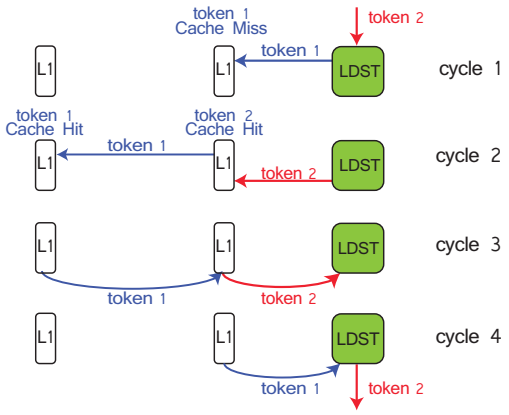


Figure 2: Thread reordering on a load/store event. Thread #1 has a cache miss in 1st cycle and stalls. On the 2nd cycle, thread #2 has a cache hit, and its data token is sent to the destination unit on the 3rd cycle. Thread #2 thus bypasses thread #1, whose data is not yet available.

only execute if all a thread's input tokens are available in its respective token buffer entries.

The SGMF model thus combines thread pipelining with out-of-order thread execution. Figure 3 illustrates the combination. The figure depicts the same scenario shown in Figure 1 with out-of-order execution. While thread #1 hits in the cache and does not stall in the LDST unit (cycle 1), thread #2 misses and stalls (cycle 2). Nevertheless, thread #3 hits (cycle 3) and its token is sent to the adder (cycle 4). Thread #3 will thus reach the multiplication node in the following cycle, and the execution of thread #3 will bypass that of the thread #2.

The amount of thread-level parallelism extracted using dynamic dataflow is proportional to the size of the token buffer. Our RTL implementation, for example, evaluated in Section 7, stores tokens for sixteen threads. The overall thread-level parallelism available using the SGMF execution model is a multiplication of the thread-level parallelism available using each method — i.e., *critical path length* $\times$ *token store size*. For example, the SGMF model can manage up to 48 concurrent threads for the graph in Figure 3, whose critical path of length 3, when using token buffers that hold 16 threads.

In summary, the SGMF execution model combines pipelining and dynamic dataflow to extract thread-level parallelism in CGRFs. The model can yield enough thread-level parallelism to hide medium-latency events such as L1 caches misses.

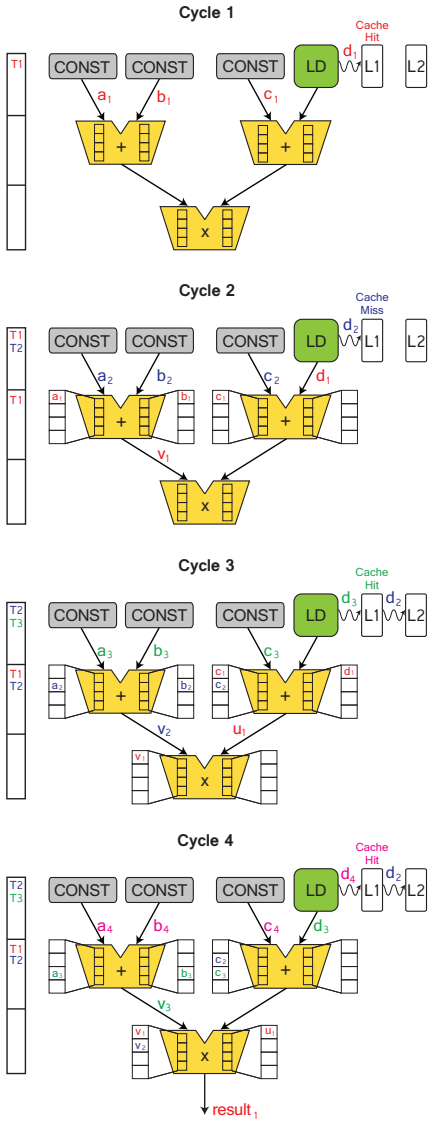


Figure 3: Dynamic dataflow example. Threads can bypass an older thread that is stalled on a cache miss.

3. Related Work

The potential performance and energy efficiency of CGRFs prompted their use to accelerate general-purpose computing. Previous studies, however, focused on single-thread performance rather than executing multiple data parallel threads.

Reconfigurable co-processors Several studies employ reconfigurable fabrics as accelerators that execute CDFGs one at a time. These rely on the compiler to identify frequently executed code blocks and convert them to CDFGs, which can be executed on the reconfigurable fabric.

DySER [11] adds CGRF functional units to an out-of-order processor. Compiler analysis maps frequently executed code blocks to execute on the CGRF functional units. BERET [12], on the other hand, does not tightly couple the CGRF into the processor but rather uses it as a co-processor for frequently executed code blocks. Garp [3] accelerates tight loops by offloading them to a CGRF accelerator. All three designs show substantial energy savings.

Unlike SGMF, these designs do not concurrently execute multiple code instances on the CGRF. DySER and BERET execute code blocks atomically, while Garp only allows streaming of data across multiple loop iterations.

Streaming processors Streaming execution models differ from GPGPUs' SIMT model [26]. SIMT executes multiple instances of the same algorithm, and each executing instance processes a different input set. In contrast, streaming programming models partition an algorithm into a graph of small filters. Streaming processors spatially stream data through the graph by mapping it to a set of interconnected compute elements.

The Imagine processor [22] maps filters onto clusters of floating-point units, using StreamIt [41] as the underlying programming model. A decade later, Sponge [17] applied StreamIt to stream data through multiple GPGPUs.

The Raw processor [24, 40] is composed of tiles, each containing a 5-stage pipelined processor. Raw explicitly exposes its micro-architectural elements to programmers (register mapping; interconnect routing; etc.). This enables programmers to manage the flow of data through the processor at fine granularity and map streaming programs across the Raw fabric.

The increasing popularity of GPGPUs and CUDA/OpenCL arguably suggests that stream-based programming is less intuitive to programmers. Our SGMF design targets the more popular, contemporary programming models.

General-purpose CGRF processors Studies such as TRIPS [36], WaveScalar [38, 39], Tartan [27] and Elastic CGRAs [18] use CGRFs for general purpose processing.

The TRIPS, WaveScalar and Tartan processors employ very different designs but share some common execution characteristics. Programs are partitioned into hyperblocks that are dynamically scheduled to execution nodes. These nodes comprise a grid of simple cores (TRIPS, WaveScalar) or use a CGRF element (the PipeRench [10] accelerator in Tartan). The nodes dynamically schedule hyperblocks according to their data dependencies. The four processors approach simultaneous thread execution using different strategies: WaveScalar pipelines multiple hyperblocks instances originating from a single thread, while its WaveCache extension [38] supports pipelining of hyperblocks originating from different threads; TRIPS dynamically schedules hyperblocks associated with different threads, but each hyperblock executes individually; and Tartan does not support simultaneous execution of hyperblocks. Finally, Elastic CGRAs [18] relaxes the scheduling constraints of operations inside the CGRF. Special circuitry controls the elastic flow of data through the fabric's functional units, which enables flexible data streaming inside a fabric.

In contrast to the aforementioned architectures, the SGMF processor supports simultaneous, out-of-order execution of multiple threads on a single CGRF. Moreover, it explicitly targets a well-established, massively-threaded programming model (while employing several techniques from TRIPS and WaveScalar in its compilation process). The architecture thus provides data-thread parallelism by incorporating both thread pipelining and dynamic instruction scheduling.

4. Design Challenges and Considerations

A CGRF designed for thread-level parallelism is different than one designed for accelerating single-thread performance. In this section we highlight the key challenges when designing a CGRF tuned for massive thread-level parallelism.

4.1. Out-of-order memory responses

The latency of memory operations is not predictable in the presence of caches, and responses from the memory system may arrive at the LDST units in an order that differs from the order in which the requests were issued. A multi-threaded CGRF must correctly handle out-of-order memory responses to guarantee functional correctness and prevent deadlocks.

Correctness: Mapping memory responses to threads Single-threaded CGRFs carry no ambiguity pertaining to the identify of a value returned from memory. A single static instruction is mapped to each LDST, which only executes one thread. A LDST unit can thus directly forward a memory response to its destination functional units. In a multi-threaded CGRF, memory responses must be associated with their issuing thread, or the functional correctness will be impaired (e.g., thread *B* will receive a memory value destined to thread *A*).

We therefore resolve to use a mechanism that resembles the miss-status holding register (MSHR) in non-blocking caches [9]. Specifically, we add a *reservation buffer* to each LDST unit, which stores the TID of each in-flight memory access. The reservation buffer is then used to associate each memory response with its issuing thread. To prevent the reservation buffer from limiting the attainable thread parallelism, we assign it the same number of entries as in the token buffer.

Deadlocks: Thread bypassing and finite token buffers The unpredictability of memory latencies coupled with the finite size of token buffers can deadlock the execution of multiple threads (deadlocks have been characterized by Culler et al. [5] as a key limitation of dynamic dataflow). An example of such a deadlock is depicted in Figure 4, which shows a graph composed of an adder that receives its input from a

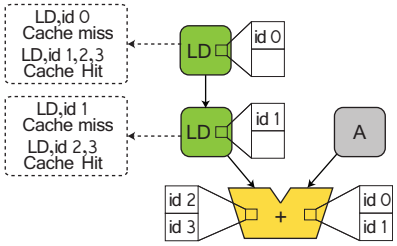


Figure 4: Deadlock example. Load operations for threads #0 and #1 get blocked at the LDST units, while those for threads #2 and #3 flow through (cache hit). The adder thus has tokens for threads #1 and #2 waiting in the right token buffer and tokens for threads #2 and #3 waiting in the left buffer.

constant unit and a chain of two LDST units (representing an addition of a double memory indirection with a constant — e.g., *array[i][j]+5*). In the example, thread #0 reaches the first LDST unit, misses in the L1 cache, and occupies one of the entries in its reservation buffer until the memory value is returned. Then, thread #1 clears the first LDST unit but stalls on the second. While threads #0 and #1 are stalled, threads #2 and #3 both clear the two LDST units (hit in the L1 cache). Their tokens thus reach the adder and wait in its left input's token buffer. However, the right input's token buffer is fully occupied with the tokens for threads #0 and #1 (sent by the constant unit), and the system deadlocks.

To prevent such deadlocks, we introduce the concept of *thread epochs*. The TIDs are sequentially partitioned epochs, each grouping a number of threads in accordance with the size of the token buffer. In our example system, whose token buffers consist of two entries, the first thread epoch will include TIDs #0 and #1, the second epoch will include TIDs #2 and #3, and so on. Thread epochs are used by the LDST units to avoid deadlocks by throttling thread parallelism. Specifically, threads that belongs to a younger (higher) epoch will not be processed by a LDST unit before all threads from previous epochs have left the node.

Returning to Figure 4, we see that thread epochs will prevent the deadlock since threads #2 and #3 will not be processed by the top LDST unit before both threads #0 and #1 have graduated. Both LDST units will therefore guarantee that threads #0 and #1 will reach the adder before threads #2 and #3 and the deadlock will be avoided.

The implementation of thread epochs adds a minimal logic to the LDST units but, as shown above, is critical to the implementation of the SGMF model.

4.2. Virtually pipelining non-pipelined arithmetic

Non-pipelined arithmetic operations may greatly hinder performance of a pipelined CGRF. Specifically, we address three types of non-pipelined operations: integer divisions, floating point divisions and square root computations. For brevity, the text only refers to floating-point division, but the mechanisms described apply to the other operation types).

We use dedicated *special compute units* to process non-pipelined operations. Each special compute units contains multiple floating-point divider blocks and can therefore compute multiple floating-point divisions at any given time. Like the LDST unit, the special compute includes a reservation buffer to track in-flight operations. Whenever the unit identifies matching input tokens in its token buffers, it stores the originating TID in an available slot in the reservation buffer and issues the floating-point division operation to an available divider. When a computation finishes, the result is tagged with its associated TID and forwarded to the destination node. The unit is depicted in Figure 8 (part of the overall architecture).

This design also allows instructions to complete out-of-order. Since the effective latency of floating-point division depends on the input operands, one thread's instruction may complete before that of an earlier thread. The tagged token dynamic dataflow model allows the result associated with the younger thread to bypass the older one's. Consequently, much like LDST units, the reservation buffer enforces thread epochs to prevent deadlocks.

Note that the special compute units hardly increases the overall amount of logic in an SGMF core compared to that of a GPGPU streaming multi-processor. Since each CUDA core already includes a floating-point division block, aggregating the blocks in the special compute units simply reorganizes existing logic. The only logic added to the group of dividers includes the reservation buffer, token buffers and a mux/demux which connect the dividers. Maintaining the area constraint, however, limits the number of special compute units that can be implemented in an SGMF core. For example, if we design the unit to include 8 dividers, and given that a Fermi streaming multi-processor has a total of 32 CUDA cores, the SGMF fabric can only contain 4 special compute units.

The special compute units allow the compiler to assume that all instructions are pipelined. The compiler can thus map a static, non-pipelined instruction to a special compute unit. In turn, the unit will consume new input tokens in (practically) every cycle by distributing their computation to one of its multiple computational blocks.

4.3. Intra- and Inter-thread memory ordering

The data parallel CUDA and OpenCL models leave it to the programmer to partition code into independent threads. Therefore, only memory operations executed by each single thread may be interdependent (we currently do not allow sharing of memory across threads). The SGMF implementation can thus reorder memory operations issued by different threads, but it must guarantee that the memory operations of each thread not violate intra-thread memory dependencies.

Nevertheless, loads may still be reordered if they do not violate load-store dependencies. In the code presented in Figure 5b, for example, the first two loads can be reordered, as long as they complete before the following store is issued. Conversely, the last two loads can also be issued concurrently,

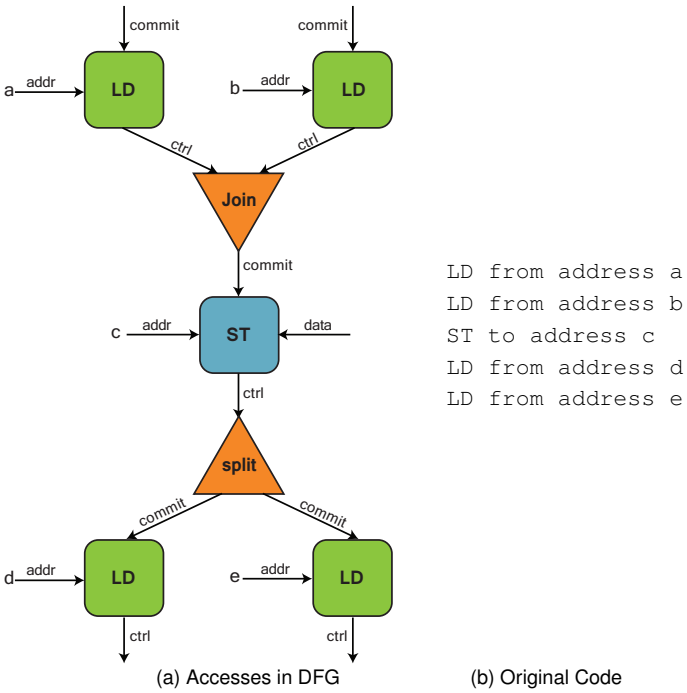


Figure 5: Memory Ordering

as long as they are issued after the preceding store completes.

To preserve memory ordering within a thread, we add control edges to the CDFG that chain memory operations. LDST units must send enable tokens so that subsequent memory operations from the same thread can be issued. Importantly, all tokens are tagged by the respective thread's TID, so parallelism between threads is still supported. Furthermore, the compiler can inject *join* operations to collect tokens from multiple LDST units and thus enable read parallelism.

Figure 5a presents the CDFG resulting from the pseudo-code in Figure 5b. The CDFG preserves read-write ordering yet still supports concurrent load operations. Specifically, the LDST nodes associated with the first two loads send their enable tokens to a join operation. The join operation sends a tagged enable token to the following LDST node only after it receives matching enable tokens from both preceding LDST nodes. Inversely, a split node enables the last two load operations to execute concurrently.

The *join* and *split* operations require additional units in the CGRF. However, as discussed in Section 5, these operations implemented using dedicated control units.

4.4. Balancing graph paths by injecting delays

Pipelining multiple threads through the graphs requires that diverge-merge paths (*hammock* paths) be of equal length. The reason is that if such paths contain different numbers of nodes, they can only pipeline different numbers of threads, thus degrading utilization (akin to pipeline bubbles). Importantly, this problem is not unique to diverging control paths but also affects diverging computation paths.

For example, Figure 6 illustrates the graph for $(a+x)*(b+c+x)$. The left path through the graph comprises two nodes, yet the right path comprises three. When pipelining threads (i.e., in-flight tokens) through this graph, the SGMF processor will only complete a thread computation every other cycle.

We address this problem by introducing conceptual delay nodes to equalize paths through the graph. In effect, we add small skid buffers (implemented as FIFOs in the control nodes described in Section 5). The buffers are configured at compile and map time to pipeline a specific number of in-flight tokens (chaining control nodes, if needed, to generate larger buffers), thereby equalizing the lengths of diverging paths.

4.5. Embedding control in the dataflow graph

Our implementation of control code closely follows the dataflow predication methods used in TRIPS [37]. Specifically, both code paths are mapped to the CGRF and control tokens are used to enable only one path. However, since our design targets thread-level parallelism rather than accelerating a single thread, some of the design tradeoffs related to dataflow predication can be handled in a somewhat different manner.

Smith et al. [37] discuss the tradeoff between the speed of propagating control information vs. edge fanout: in a design that targets single-thread performance, the results of a branch instruction should ideally be propagated directly to each instruction predicated by the branch. However, this strategy yields high graph connectivity and edge fanouts. In their research, Smith et al. propose a mapping scheme that balances these opposing trends.

Designing for thread-level parallelism rather than single-thread performance enables a small relaxation to the algorithm proposed by Smith et al. Specifically, we add label nodes that govern the diverging code paths [33]. A label node collects the control tokens of all preceding branches and therefore aggregates all the conditions that dominate the execution of the basic block it controls. In addition, SSA Phi nodes [6] are translated into multiplexers, whose selection token is governed by the label node and whose data tokens are received from the nodes generating its possible values on both control paths.

Importantly, loads and stores on non-taken code paths must be disabled, so that false accesses will not be sent to the memory system. The non-taken label node must thus send a poisoned control token to its subordinate LDST node. This process is simplified since load and stores are chained (as described above). Consequently, the control nodes need only send values to the first LDST node in the chain, and the control token is forwarded along the chain.

Figure 7b demonstrates the conversion from divergent code to a CDFG. The CDFG includes condition nodes that output *True* or *False* and a label node for each code label. As depicted, the label nodes propagate their control tokens down the code chains to enable/disable the relevant memory operations.

In summary, embedding control information into the dataflow graph is required for correct execution, but may also increase the number of nodes and edges. As discussed in Sec-

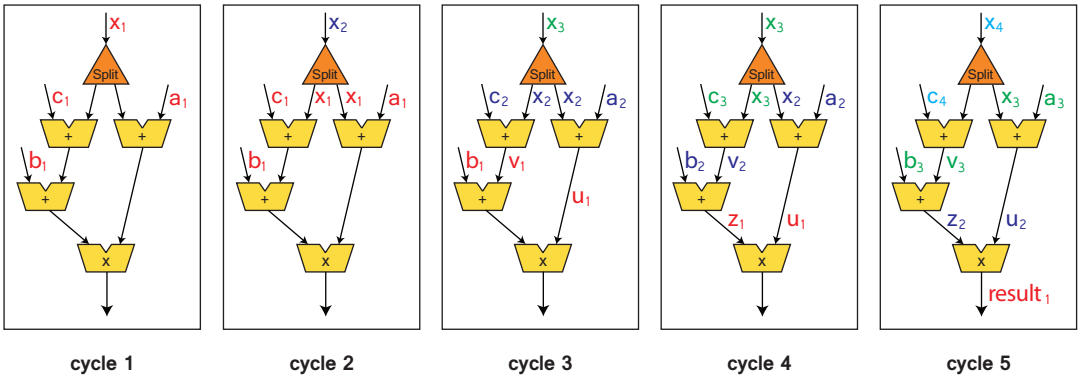
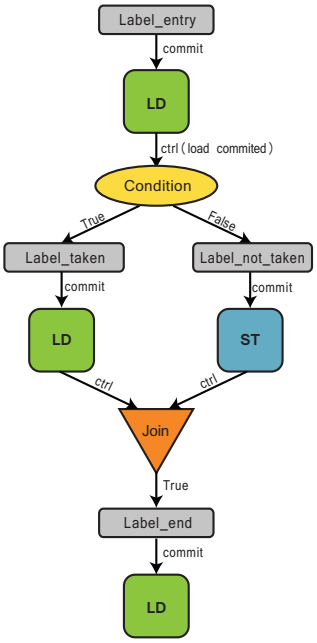


Figure 6: Functional unit stall due to an imbalance in paths' lengths.



(a) Control-Data Flow Graph.

```
entry:
   ...
   LD
   ...
   if (condition)
      jmp to taken

not_taken:
   ...
   ST
   ...

taken:
   ...
   LD
   ...

end:
   LD
   ...
   return
```

(b) Original Control Code.

Figure 7: A Control-data flow graph (CDFG). An ellipsis indicates code without control or memory operations.

tion 5, the SGMF processor includes dedicated control nodes to implement the *label* functionality.

4.6. Large kernels and loops

Since the size of the reconfigurable grid is finite, kernels may need to be partitioned into smaller kernels. Specifically, this is needed if kernels require more functional units than available in the grid or cannot map due to the limited grid connectivity. Kernel partitioning splits a larger kernel into multiple smaller ones that are executed sequentially[1]: *a)* all threads execute the first kernel, storing their intermediate values to memory; *b)* the SGMF processor reconfigures the grid to run the next kernel; *c)* all threads execute the second kernel; and *d)* the processor repeats steps *b* and *c* as needed.

Loops are handled in a similar manner. A kernel that contains a loop is split into an *initialization kernel* that includes all the code that precedes the loop; the *loop kernel*; and an *epilogue* kernel that consists of the code that follows the loop. The three kernels are executed in sequence, as described above, with one difference: the loop kernel is executed repeatedly according to the number of loop iterations. Note that loop iterations execute sequentially for each thread and thereby do not violate loop-carried dependencies. Furthermore, memory operations in the loop kernel are predicated on a thread basis (similar to vector masks [35]), allowing threads to execute different numbers of iterations.

5. The SGMF Architecture

In this section, we present the implementation of the data-parallel SGMF architecture. The SGMF processor is designed as a multi-core architecture that comprises multiple SGMF cores. Each SGMF core is a CGRF composed of interconnected heterogeneous units.

The memory system of the SGMF processor is similar to that of the Nvidia Fermi processor. Each SGMF core is connected to a private, banked L1 cache. A banked L2 cache is shared among all cores. The main memory is composed of a banked, multi-channel GDDR5 DRAM.

5.1. Reservation and operand buffers

The SGMF architecture does not require a central Register-File (RF). Since no temporal data is saved, the large RF can be replaced with small buffers that are spread across the different units. Two types of buffers are used: *a) token buffers* in each unit store input tokens (operands) until all tokens of the same thread have arrived; and *b) reservation buffers* track in-flight memory operations. These buffers enable thread reordering by allowing memory values to flow into the fabric out-of-order.

When a load command is sent to the memory system, the issuing thread's TID is saved in the reservation buffer. When the data returns from memory, it will be tagged with its thread's TID and sent to the next unit. TIDs are grouped into epochs

[1]While the partitioning process will ultimately be automated, our current software toolchain does not support automatic kernel partitioning. This process is currently manual and sub-optimal.

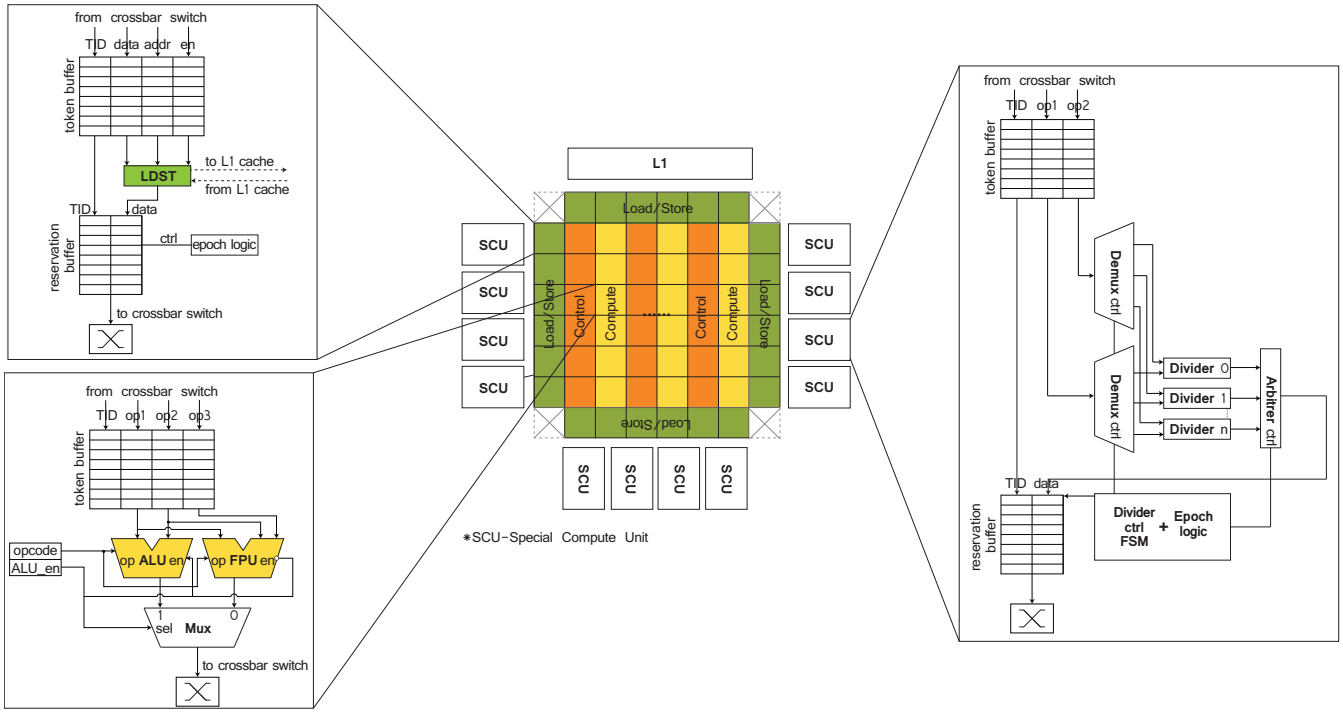


Figure 8: An example SGMF fabric. LDST units and special compute units are located on the perimeter; the internal grid consists of alternating columns of control and compute units.

(Section 4.1), which limit out-of-order execution of memory operations to groups of threads and thereby guarantee buffer space in the following level of the graph. Inside each unit, threads that belong to the same epoch are served in FR-FCFS policy. Once all the tokens from the current epoch have been sent, the epoch counter is advanced by one, and the tokens from the next epoch can be sent on.

The sizes of the token and reservation buffers determine the number of in-flight threads supported by the SGMF core and introduce a power/performance tradeoff. Increasing the buffer sizes provides better utilization in the presence of long latency operations but also consumes more power. Our evaluation (Section 7) shows that setting the number of entries in the token and reservation buffers to 16 provides a good power/performance balance. This brings the total buffering in an SGMF core to $\sim$18 KB, which is $\sim$37% of the 48 KB RF in a Fermi streaming multiprocessor (SM) [30].

The low buffering requirements of the SGMF core are attributed to the SGMF execution model. Since the core concurrently executes multiple instructions from each thread, it can maintain fewer in-flight threads yet still deliver high grid utilization. In contrast, the von-Neumann based Fermi SM must store the temporal state of *all* CUDA threads it is assigned to.

5.2. The SGMF core

Each SGMF core is composed of four types of units: *a)* compute units; *b)* load/store units; *c)* control units; and *d)* special compute units. A schematic figure of the hardware organization is shown in Figure 8. Each unit contains an operand buffer and a reconfigurable switch that connects the tile to its eight nearest neighbors (8-NN). Units also contain a configuration register that specifies their token routing (destination nodes for the tokens) and opcode (for compute units). The register is written once, when the kernel is loaded.

Compute units The compute unit is similar to an Nvidia Fermi CUDA core. It comprises an arithmetical-logical unit (ALU) and a single-precision floating point unit (FPU).

Since compute units are pipelined and repeatedly execute a single static instruction, they do not block. Instructions flowing through a unit thus execute in-order. To boost performance, the fused-multiply-add operation is also supported by the unit.

Load/Store units Load/Store (LDST) units are spread across the perimeter of the fabric and are connected to the banks of the L1 cache using a crossbar. The units include a reservation buffer to allow multiple memory accesses to execute in parallel. The buffer also allows memory accesses to graduate out-of-order and for ready threads to bypass blocked ones.

Control units Control units implement the functionality of control nodes in the CDFG. These include the label nodes and join/split nodes. The control units also include a FIFO, which is used to add delay slots when equalizing graph paths 4.4.

5.2.1. Special compute units The special compute units execute multiple non-pipelined arithmetic operations (*div*, *sqrt*) in parallel using multiple logic blocks. Each of these units can thereby be viewed by the compiler as a single virtually pipelined unit. These units are therefore larger than others and are located on the perimeter of the SGMF core.

Each SGMF core reorganizes the logic blocks available

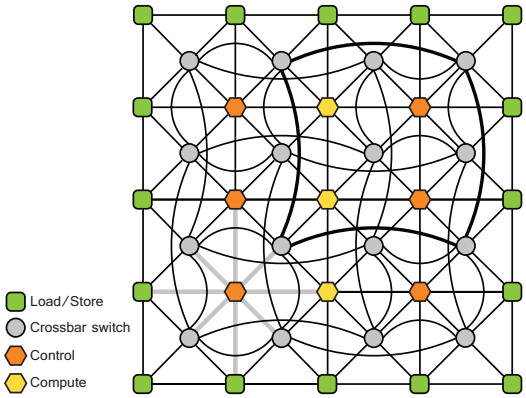


Figure 9: Topology of 5×5 grid. *1)* Each unit is connected to the 4 nearest switches and the 4 nearest units (gray lines in bottom left quadrant); *2)* Each switch is connected to the 4 nearest units; *3)* Each switch is also connected to the 4 switches with the distance of 2 hops (bold dark arches).

in each Fermi SM into a set of 108 functional units. These include 32 compute units, 32 control units, 32 LDST units and 12 special compute units ($4\times$ for integer division, $4\times$ for single-precision floating-point division and $4\times$ for square root computation). In comparison, a Fermi SM organizes the same units into 32 CUDA cores, each composed of a compute block (ALU and floating-point, including division blocks) and a special function units that implements square root operations. Crucially, this fine-grain organization enables SGMF cores to operate more functional units at any given time.

5.3. Interconnect

The interconnect is implemented as reconfigurable crossbar switches that are configured once for each kernel. Since the configuration of the switches is static, new tokens can be sent through the switch on every cycle.

An example 5*x*5 grid is shown in Figure 9. The network topology is a folded hypercube [8], which equalizes the connectivity of the switches on the perimeter. Switches are interconnected at distances of two (e.g., a switch at location $[n,n]$ is connected to switches at $[n\pm 2, n\pm 2]$), which decreases the number of hops needed for connecting distant units on the fabric. Furthermore, adjacent functional units are directly connected to each other.

6. Methodology

RTL Implementation We have implemented the SGMF core in Verilog (including all unit types and interconnect) to evaluate its components' power, area and timing. The design was synthesized using the Synopsys toolchain and a commercial 65*nm* cell library, and the results were then extrapolated for a 40*nm* process. Based on this process, we determine that the SGMF core consumes 54.30mm^2 at 65*nm* and $\sim$21.18mm^2 at 40*nm*. For comparison, the 40*nm* die size used in the Nvidia Fermi GTX480 card is over 500mm^2.

Parameter	Value
SGMF Core	32 func. /32 control/32 LDST
SGMF System	15 SGMF cores
Fermi/GTX480 System	15 streaming multiprocessors
Frequency [GHz]	core 1.4, Interconnect 1.4 L2 0.7, DRAM 0.924
L1	64KB, 32 banks, 128B/line, 4-way
L2	786KB, 6 banks, 128B/line, 16-way
GDDR5 DRAM	16 banks, 6 channels

Table 1: SGMF cache configuration.

Simulation framework The comparative performance and power evaluation was done using the GPGPU-Sim cycle-accurate simulator [2] and GPUWattch [25] power model. The baseline for both performance and power models is the Nvidia GTX480 card that is based on the Nvidia Fermi processor. We have extended GPGPU-Sim to simulate SGMF cores. These replaced the original Fermi SMs in the GTX480 model.

GPUWattch uses performance monitors to estimate the total execution energy. Using the per-operation energy estimates, obtained from the synthesis results, we have extended GPUWattch's power model to support SGMF cores.

The system configuration is shown in Table 1. By replacing the Fermi SMs with SGMF cores, we have fully retained the memory system design. The only differences between the Fermi and SGMF memory systems are that SGMF uses write-back and write-allocate policies in the L1 caches, as opposed to Fermi's write-through and write-no-allocate.

Compiler We compile CUDA code using LLVM [23] and pass the resulting SSA [6] code to an SGMF backend. The backend maps the SSA code to the SGMF grid and configures its interconnect. Kernels that could not be directly mapped (Section 4.6) was partitioned manually.

Benchmarks We evaluate the SGMF architecture using kernels from the Rodinia benchmark suite [4] and are listed in Table 2. Importantly, the benchmarks are used as-is and are optimized for SIMT processors.

7. Evaluation

We evaluate the SGMF architecture both at the single and multi-core level and compare its performance and energy to that of a single SM in the Nvidia Fermi and the entire GTX480 system, respectively. We also explore the impact of varying token buffer sizes on overall performance and power.

7.1. SGMF core

To isolate the performance and energy benefits of the SGMF architecture, we first evaluate a single core (which uses a full GTX480 memory system). This isolation eliminates memory effects induced by inter-core cache contention.

Performance Figure 10 shows performance speedups of an SGMF core over a Fermi SM. The figure shows that the single-core SGMF processor enjoys consistent performance gains over Fermi. For example, using buffers of 16 tokens yields

Application	Application Domain	Kernels	Description
CFD	Fluid Dynamics	*compute_step_factor* , *cuda_time_step*	Computational fluid dynamics solver
NN	Data Mining	*euclid*	K nearest neighbor distance calculator
PF	Medical Imaging	*normalize_weights_kernel*	Particle filter - target location estimator
BFS	Graph Algorithms	*kernel2*	Breadth-first search
GE	Linear Algebra	*Fan1* , *Fan2*	Gaussian elimination - linear system solver
BPNN	Pattern Recognition	*bpnn_adjust_weights_cuda*	Back Propagation - training of a neural network

Table 2: A short description of the benchmarks that was used to evaluate the system

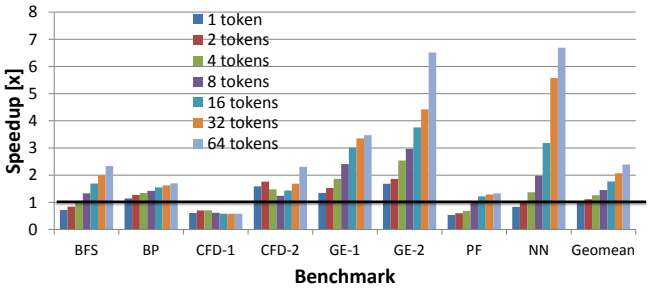


Figure 10: Speedup of an SGMF core, with varying token buffer sizes, over a Fermi SM.

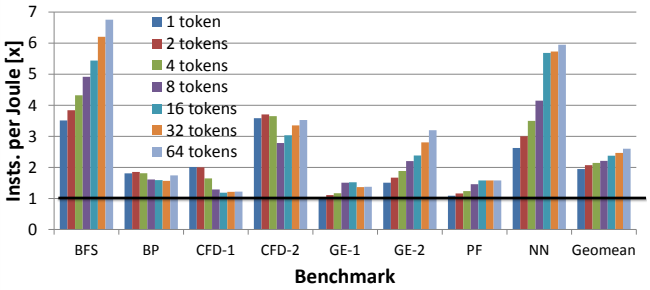


Figure 11: Energy efficiency of an SGMF core with varying token buffer sizes. Improvement is shown as *insts.-per-Joule(SGMF)/insts.-per-Joule(Fermi)*.

speedups ranging from 1.2× for PF to 3.75× for GE-2, with an average speedup of 1.77×. The only exception is the memory bound CFD-1, whose threads stall on memory accesses.

The SGMF core combines both instruction-level parallelism (ILP) and thread-level parallelism (TLP) to maximize the utilization of the grid, as their culmination delivers more instructions that are ready to execute at any given time.

ILP is attributed to the amount of parallelism available in the SGMF core, compared with that available in the SM. In particular, the SM executes a 32 instructions (a thread warp) every cycle. This limits its parallelism to at most 32 operations per cycle. In contrast, the SGMF core decomposes the same type and amount of logic available on the SM into 108 units that can execute in parallel. Effectively, the logic of a single CUDA core, which can execute one instruction per cycle, is thus distributed in the SGMF core into distinct units that execute concurrently: a compute unit; a control unit; and a special compute unit. The SGMF core thus delivers more parallelism than Fermi's SM.

Distinctively, TLP is determined by the size of the token and operand buffers, that enable each functional unit to manage multiple instructions in-flight. The effect of TLP is highly visible in the memory intensive CFD-2. Increasing the number of threads initially degrades performance due to cache contention. Nevertheless, once the size of the buffers exceed 8 tokens, the performance improves as the amount of TLP begins to hide the memory latencies [14].

Energy efficiency Figure 11 compares the energy efficiency of an SGMF core (with varying token buffer sizes) with that of a Fermi SM, by depicting the ratio between the *insts./Joule* they yield (reflecting both static and dynamic energy and that of the memory system). The figure shows that an 16-token SGMF processor delivers 1.2–5.7× more *insts./Joule* for the different benchmarks, with an average of 2.37×.

The energy efficiency of the SGMF is attributed to two factors: its increased parallelism over the SM core and its ability to eliminate most of the von-Neumann energy inefficiencies.

As noted above, the SGMF architecture uses the same amount and types of logic elements available in the Fermi SMs. It does, however, allow more elements to provide useful work in parallel. For example, SM's CUDA cores includes logic for both arithmetic and branch operations. Yet, even though both are active on every cycle only one provides useful work. In contrast, the SGMF core splits the arithmetic and branch logic into two distinct unit types — the compute unit and the control unit, respectively. The SGMF core thus spends the same amount of energy executing both logic elements on any given cycle, but can extract useful work from both.

Finally, we find that the SGMF core spends less energy on artifacts of the execution model. For example, an energy breakdown of the two cores (not shown) uncovers that SGMF spends 67% less energy on management structures (interconnect, buffers, functional units) than the Fermi SM (register file, global pipeline, instruction scheduler).

In summary, we show that an SGMF core yields both better performance and higher energy efficiency that a Fermi SM.

7.2. Multicore SGMF processor

We now turn to evaluate a 15-core SGMF multicore processor and compare it to the Nvidia GTX480 (15 Fermi SMs).

Performance Figure 12 shows the speedups obtained by the multicore SGMF processor over the 15-SM Fermi. Interestingly, the depicted results are not as consistent as those obtained for the single SGMF core (note that both measurements employ the same memory system configuration). The multicore SGMF processor with 16-slots token buffers yields

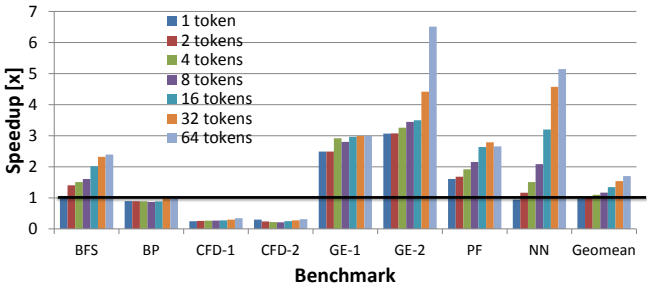


Figure 12: Speedup of a 15-core SGMF processor, with varying token buffer sizes, over a Fermi with 15 SMs.

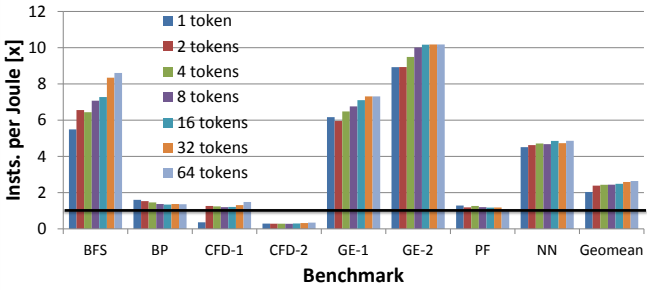


Figure 13: Energy efficiency of a 15-core SGMF processor with varying token buffer sizes. Improvement is shown as *insts.-per-Joule(SGMF)/insts.-per-Joule(Fermi)*.

speedups that vastly range between 0.25–3.5×. Nevertheless, on average, the speedup is 1.35×. The inconsistent speedups suggest that the SGMF memory behavior is quite different than that of the Fermi multi-processor and may, at times, suffer from memory contention in a multicore configuration. Conversely, benchmarks that do not contend for shared resources greatly benefit from the SGMF architecture. In particular, GE-1, GE-2 and NN achieve speedup of 2.6–3.5×.

The distinct access patterns are explained by the nature of the two execution models. When encountering memory instructions, SIMT threads all execute the same static instruction. In contrast, SGMF threads all execute distinct static memory instructions at any given time. SIMT threads are thus typically optimized to coalesce their memory accesses, such that multiple concurrent accesses are directed to different cache banks. Indeed, the Fermi memory system itself is designed for coalesced accesses, and the Rodinia kernels are optimized for SIMT processors rather than for SGMF. Under these settings, increasing the size of the token buffer, and thereby the number of active threads, may not beneficial since the L2 cache and memory system get overloaded. For example, the BP, CFD-1 and CFD-2 kernels, which suffer from non-coalesced L2 accesses, are agnostic to the token buffer size.

In summary, the multicore performance comparison attest to the performance benefits of the SGMF architecture and, at the same time, motivate further research to the design of an SGMF-tuned memory system.

Energy Figure 13 depicts the ratio between the $instructions/Joule$ achieved by a 15-core SGMF processor (with varying token buffers sizes) over a Fermi GTX480. As expected, the results coinside with those shown in the multicore performance comparison.

For example, the CFD-2 kernel, which exhibits intense cache contention and achieve very low performance on the SGMF multicore, suffers degraded energy efficiency. Specifically, they achieve only ∼0.3× the efficiency they enjoy on the Fermi regardless of the token buffer size. This caused by the increased energy consumption of the memory system that is triggered by the contention. For CFD-2, the cache system consumes 7.5× more energy when serving the SGMF cores.

In contrast, GE-1 and GE-2, which do not suffer from cache contention, greatly benefit from the improved energy efficiency of the SGMF cores. They enjoy 7.1× and 10.1× better energy efficiency, respectively, for 16-slot token buffers.

In conclusion, our evaluation highlights the high performance and energy efficiency of the SGMF architecture. Despite its susceptibility to cache contention on existing GPGPU memory systems, the multicore SGMF processor achieves an average speedup of ∼2.1× and improved energy efficiency of ∼3.2× when compared to an Nvidia Fermi multicore.

8. Conclusions

We have presented the *single-graph multiple-flows* (SGMF) execution model and architecture. The model targets coarse-grain reconfigurable fabrics (CGRFs) and is compatible with Nvidia's *single-instruction multiple-threads* (SIMT) model. The SGMF architecture is presented as an energy efficient design alternative to contemporary GPGPUs. The architecture employs thread pipelining and dynamic instruction scheduling to concurrently execute multiple data-parallel threads on a CGRF. Specifically, dataflow graphs are extracted from CUDA kernels and are mapped to the modified CGRF.

The paper describes the design challenges of a CGRF-based multi-threaded architecture and how they can be resolved. Specifically, these include preventing deadlocks among unordered threads, hiding latencies of non-pipelined arithmetic operations, enabling inter-thread memory parallelism while preserving intra-thread memory ordering, and avoiding underutilization of functional units by equalizing paths in the dataflow graph.

We evaluate the performance and energy efficiency of the SGMF architecture using kernels from the Rodinia benchmark suite. We show that, on average, an SGMF core delivers better performance than a streaming multi-processor of an Nvidia Fermi GPGPU, while consuming ∼57% less energy.

Finally, we motivate further research into the design of the SGMF architecture, and specifically its memory system. We show how the raw energy efficiency of the SGMF architecture is hindered by the use of existing GPGPUs' memory systems, which are tuned to the existing SIMT execution model rather than the SGMF model.

Acknowledgment

We thank the anonymous referees for their valuable comments and suggestions. This research was funded by the Israel Science Foundation (ISF grant 769/12; equipment grant 1719/12) and an FP7 CIG grant (no. 334258). Y. Etsion was supported by the Center for Computer Engineering at Technion; D. Voitsechov was supported by the Hasso Plattner Institute (HPI).

References

[1] Arvind and R. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Trans. on Computers*, vol. 39, no. 3, pp. 300–318, Mar 1990.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator." in *ISPASS*. IEEE, 2009, pp. 163–174.

[3] T. J. Callahan and J. Wawrzynek, "Adapting software pipelining for reconfigurable computing," in *Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000, pp. 57–64.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE Intl. Symp. on Workload Characterization (IISWC)*, ser. IISWC '09, 2009, pp. 44–54.

[5] D. E. Culler, K. E. Schauser, and T. von Eicken, "Two fundamental limits on dataflow multiprocessing," in *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, 1993, pp. 153–164.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[7] J. B. Dennis and D. Misunas, "A preliminary architecture for a basic data flow processor," in *Intl. Symp. on Computer Architecture (ISCA)*, 1975, pp. 126–132.

[8] A. El-Amawy and S. Latifi, "Properties and performance of folded hypercubes," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 1, pp. 31–42, Jan. 1991.

[9] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Trans. on Computers*, vol. 45, no. 5, pp. 552–571, 1996.

[10] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A reconfigurable architecture and compiler," *IEEE Computer*, vol. 33, no. 4, pp. 70–77, Apr. 2000.

[11] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2011, pp. 503–514.

[12] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Intl. Symp. on Microarchitecture (MICRO)*, 2011, pp. 12–23.

[13] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Comm. ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[14] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. Weiser, "Many-core vs. many-thread machines: Stay away from the valley," *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 25–28, Jan 2009.

[15] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Intl. Symp. on Computer Architecture (ISCA)*, 2010, pp. 37–47.

[16] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Intl. Symp. on Computer Architecture (ISCA)*, 2010, pp. 280–289.

[17] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: portable stream programming on graphics engines," in *Intl. Conf. on Arch. Support for Prog. Lang. & Operating Systems (ASPLOS)*, 2011, pp. 381–392.

[18] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic CGRAs," in *Intl. Symp. on Field Programmable Gate Arrays*, 2013, pp. 171–180.

[19] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Intl. Symp. on Microarchitecture (MICRO)*, 2003, pp. 93–104.

[20] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, pp. 7–17, 2011.

[21] S. Keckler, "Life after Dennard and how I learned to love the picojoule," Intl. Symp. on Microarchitecture (MICRO), 2012, (keynote).

[22] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: Media processing with streams," *IEEE Micro*, vol. 21, pp. 35–46, 2001.

[23] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Intl. Symp. on Code Generation & Optimization (CGO)*, 2004, pp. 75–.

[24] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine," in *Intl. Conf. on Arch. Support for Prog. Lang. & Operating Systems (ASPLOS)*, 1998, pp. 46–57.

[25] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: enabling energy optimizations in GPGPUs," in *Intl. Symp. on Computer Architecture (ISCA)*, 2013, pp. 487–498.

[26] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[27] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, M. Budiu, and S. C. Goldstein, "Tartan: Evaluating spatial computation for whole program execution," in *Intl. Conf. on Arch. Support for Prog. Lang. & Operating Systems (ASPLOS)*, Oct 2006, pp. 163–174.

[28] J. Nickolls and W. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.

[29] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[30] Nvidia, *Fermi Compute Architecture Whitepaper*.

[31] NVIDIA, "NVIDIA Tegra 4 family CPU architecture: 4-PLUS-1 quad core," 2013. [Online]. Available: http://www.nvidia.com/docs/IO/116757/NVIDIA_Quad_a15_whitepaper_FINALv2.pdf

[32] OpenCL Working Group, "The OpenCL specification," www.khronos.org/opencl, Oct 2009, ver. 1.0.

[33] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *Intl. Conf. on Programming Language Design and Impl. (PLDI)*, 1990, pp. 257–271.

[34] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," in *Intl. Symp. on Computer Architecture (ISCA)*, 1990, pp. 82–91.

[35] R. M. Russell, "The CRAY-1 computer system," *Comm. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.

[36] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *Intl. Symp. on Computer Architecture (ISCA)*, 2003, pp. 422–433.

[37] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley, "Dataflow predication," in *Intl. Symp. on Microarchitecture (MICRO)*, 2006, pp. 89–102.

[38] S. Swanson, A. Schwerin, A. Petersen, M. Oskin, and S. Eggers, "Threads on the cheap: Multithreaded execution in a WaveCache processor," in *Workshop on Complexity-effective Design (WCED)*, 2004.

[39] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec 2003, p. 291.

[40] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[41] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *Intl. Conf. on Compiler Construction*, Apr 2002, pp. 179–196.

[42] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 1995, pp. 392–403.

[43] W. Yamamoto, M. Serrano, A. Talcott, R. Wood, and M. Nemirosky, "Performance estimation of multistreamed, superscalar processors," in *Hawaii Intl. Conf. on System Sciences*, vol. 1, 1994, pp. 195–204.