Control Flow Coalescing on a Hybrid Dataflow/von Neumann GPGPU

Dani Voitsechov
Electrical Engineering
Technion - Israel Institute of Technology
dani@tce.technion.ac.il

Yoav Etsion
Electrical Engineering and Computer Science
Technion - Israel Institute of Technology
yetsion@tce.technion.ac.il

ABSTRACT

We propose the hybrid dataflow/von Neumann *vector graph instruction word* (VGIW) architecture. This data-parallel architecture concurrently executes each basic block's dataflow graph (*graph instruction word*) for a vector of threads, and schedules the different basic blocks based on von Neumann control flow semantics. The VGIW processor dynamically coalesces all threads that need to execute a specific basic block into a thread vector and, when the block is scheduled, executes the entire thread vector concurrently. The proposed *control flow coalescing* model enables the VGIW architecture to overcome the control flow divergence problem, which greatly impedes the performance and power efficiency of data-parallel architectures. Furthermore, using von Neumann control flow semantics enables the VGIW architecture to overcome the limitations of the recently proposed *single-graph multiple-flows* (SGMF) dataflow GPGPU, which is greatly constrained in the size of the kernels it can execute. Our evaluation shows that VGIW can achieve an average speedup of 3× (up to 11×) over an NVIDIA GPGPU, while providing an average 1.75× better energy efficiency (up to 7×).

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles — dataflow architectures

Keywords

SIMD, GPGPU, dataflow, reconfigurable architectures

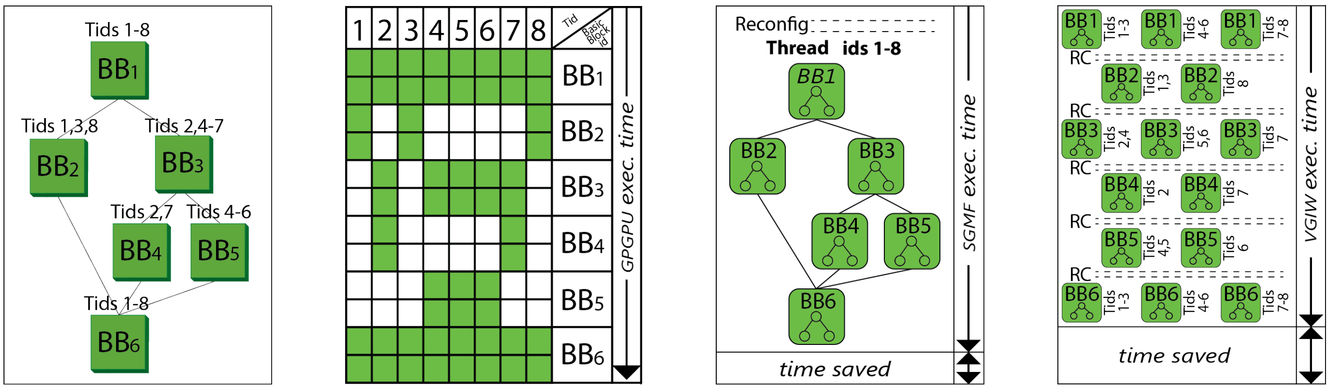
1. INTRODUCTION

The popularity of general purpose graphic processing units (GPGPU) can be attributed to the benefits of their massively parallel programming model, and to their power efficiency in terms of floating point operations (FLOPs) per watt. These benefits are best attested to by the fact that 9 out of the top 10 most power-efficient supercomputers employ NVIDIA or AMD GPGPUs [1, 2]. Yet despite their superior power efficiency, GPGPUs still suffer from inherent inefficiencies of the underlying von Neumann execution model. These include, for example, the energy spent on communicating intermediate values across instructions through a large register file, and the energy spent on managing the instruction pipeline (fetching, decoding, and scheduling instructions). Recent studies [3, 4] estimate that the pipeline and register file overheads alone account for ∼30% of GPGPU power consumption.

As an alternative to von Neumann GPGPUs, Voitsechov and Etsion [5] recently proposed the Single-Graph Multiple-Flows (SGMF) dataflow GPGPU architecture. The SGMF architecture introduces the *multithreaded coarse-grained reconfigurable fabric* (MT-CGRF) core, which can concurrently execute multiple instances of a *control and dataflow graph* (CDFG). The MT-CGRF core eliminates the global pipeline overheads and, by directly communicating intermediate values across functional units, eliminates the need for a large register file. Furthermore, the underlying dataflow model enables the core to extract more instruction-level parallelism (ILP) by allowing different types of functional units to execute in parallel (e.g., all FP and all LD/ST units).

The SGMF architecture, however, is limited by the static mapping of a kernel's CDFG to the MT-CGRF core and cannot execute large CUDA kernels efficiently. First, the limited capacity of the MT-CGRF core cannot host kernels whose CDFG is larger than the reconfigurable fabric. Second, as all control paths through a kernel's CDFG are statically mapped to the MT-CGRF, the core's functional units are underutilized when executing diverging control flows.

In this paper, we propose the hybrid dataflow/von Neumann *vector graph instruction word* (VGIW) architecture for GPGPUs. Like a dataflow machine, VGIW represents basic blocks as dataflow graphs (i.e., *graph instruction words*) and concurrently executes each block for a vector of threads using the MT-CGRF core. Like a von Neumann machine, threads' control flows determine the scheduling of basic blocks. This hybrid model enables the architecture to dynamically coalesce all threads that are about to execute a basic block into a thread



(a) Control flow example (b) GPGPU (c) SGMF (d) VGIW

Figure 1: Illustration of the runtime benefits provided by control flow coalescing. The VGIW model enjoys the performance and power benefits of dataflow execution while eliminating the resource underutilization caused by control divergence.

vector that is executed concurrently. The VGIW architecture thus preserves the performance and power efficiency provided by the dataflow MT-CGRF core, enjoys the generality of the von Neumann model for partitioning and executing large kernels, and eliminates inefficiencies caused by control flow divergence.

Control flow coalescing requires that the architecture communicate live, intermediate values across basic blocks. The VGIW architecture provides this functionality using a *live value cache* (LVC). The LVC functionality resembles that of a register file. But since instruction dependencies are typically confined inside a basic block, most intermediate values are communicated directly through the MT-CGRF, and the LVC only handles a small fraction of inter-instruction communication.

We implemented the architecture using GPGPUSim [6] and GPGPUWattch [4], and evaluated it using kernels from the Rodinia benchmark suite [7]. In order to model the architecture's performance and power properties, we implemented its key components in RTL and synthesized them using a commercial cell library. Our results show that the VGIW architecture outperforms the NVIDIA Fermi architecture by 3× on average (up to 11×) and provides an average 1.75× better energy efficiency (up to 4.3×).

This paper makes the following contributions:

- Introducing the hybrid dataflow/von Neumann *vector-graph instruction word* (VGIW) architecture for GPGPUs.
- Introducing the *control flow coalescing* execution model, which overcomes control flow divergence across data-parallel threads.
- Evaluating a VGIW processor using CUDA kernels from the Rodinia benchmark suite.

The rest of this paper is organized as follows: Section 2 introduces the abstract VGIW machine and execution model, followed by a detailed description of the architecture in Section 3. Section 4 describes our experimental methodology, and Section 5 discusses the evaluation results. Finally, we discuss related work in Section 6 and conclude in Section 7.

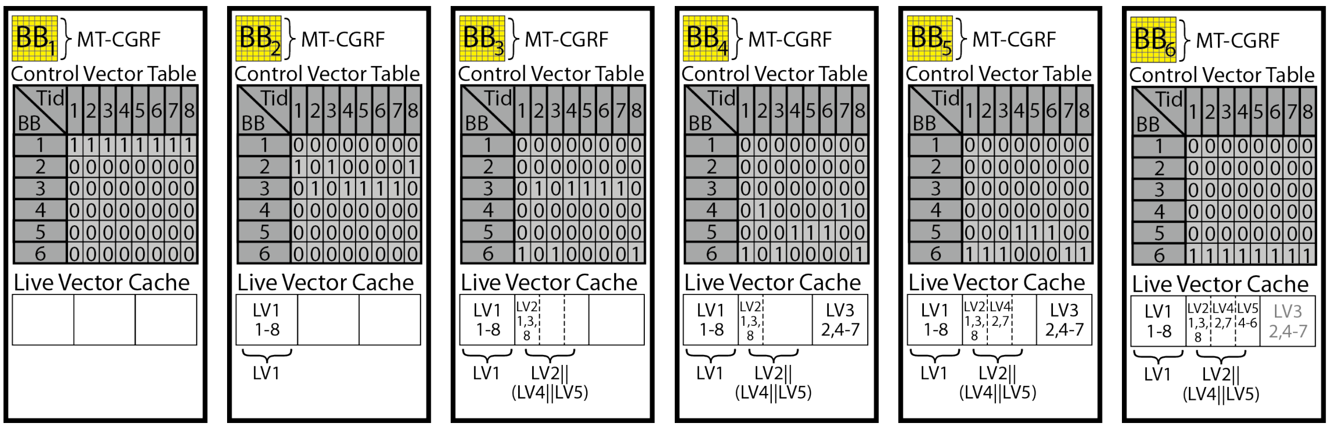
2. CONTROL FLOW DIVERGENCE AND THE VGIW EXECUTION MODEL

The control flow divergence problem, in which data-parallel threads take different paths through the kernel's control flow graph, is known to affect the utilization of data-parallel architectures. Specifically, the execution resources that are assigned to diverging threads whose control flow bypasses the currently executing basic block are wasted.

Figure 1 illustrates how control divergence in a simple nested conditional statement (Figure 1a) impacts contemporary von Neumann GPGPUs, the SGMF data-parallel dataflow architecture [5], and the proposed VGIW design. For brevity, we only show the execution of the kernel on eight threads, whose control flow diverges asymmetrically: threads 1,3,8 execute basic blocks BB1, BB2 and BB6, threads 2,7 execute basic blocks BB1, BB3, BB4 and BB6 while threads 4–6 execute basic blocks BB1, BB3, BB5 and BB6.

Von Neumann GPGPUs execute groups of threads (warps) in lockstep. When the control flow of threads in a group diverges, the GPGPU applies an execution mask to disable lanes (functional units) associated with threads that need not execute the scheduled instruction. As a result, the runtime of threads on a von Neumann GPGPU is composed of the execution time of both taken and non-taken control paths. Figure 1b illustrates this phenomenon. The lanes allocated to threads 2,4–7 are disabled when block BB2 executes. Conversely, when block BB3 executes, all lanes associated with threads 1,3,8 are disabled. The utilization further drops when dealing with nested conditionals and, when executing BB4, all lanes associated with threads 1,3–6,8 are disabled; the same goes for threads 1–3,7–8 when executing BB5. Even though the issue of control divergence has been studied extensively (e.g., [8–12]), it still impacts the performance of contemporary GPGPUs.

Control flow divergence also degrades the power efficiency of the recently proposed SGMF dataflow GPGPU. The SGMF processor maps all the paths through a con-



(a) Before BB1 (b) After BB1 (c) After BB2 (d) After BB3 (e) After BB4 (f) After BB5

Figure 2: The VGIW machine state throughout the execution of the control flow depicted in Figure 1a.

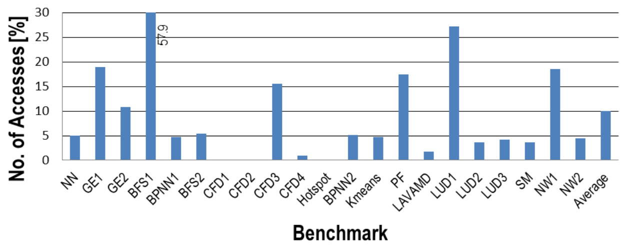


Figure 3: The number of accesses to the LVC as a fraction of the number of accesses to a GPGPU RF for the kernels evaluated in the paper.

trol flow graph onto its MT-CGRF core, as depicted in Figure 1c. The processor thus effectively executes all thread control flows in parallel. While this method prevents branch divergence from increasing thread runtime, it affects the utilization of its computational resources. Ultimately, even though the dataflow SGMF processor delivers substantial performance speedups compared to a von Neumann GPGPU, its efficiency is greatly affected by control flow divergence.

We propose to overcome the control flow divergence problem using *control flow coalescing*, which aggregates all threads waiting to execute a basic block. This execution model, which executes only the blocks needed by each thread, eliminates the loss of both cycles and execution resources due to branch divergence. Figure 1d illustrates the proposed model. For example, it shows how BB3 is executed only by threads 2 and 4–7. It further demonstrates how multiple replicas of BB3 can be mapped to the MT-CGRF core to maximize the utilization of the fabric and accelerate the execution. The rest of this section introduces the VGIW machine model and discusses how its hybrid dataflow/von Neumann execution model facilitates *control flow coalescing.*

The VGIW machine model

The *vector graph instruction word* (VGIW) execution model dynamically coalesces all (possibly divergent) threads that need to execute the same basic block into a thread vector. When a basic block is scheduled to execute on the MT-CGRF core (by loading its dataflow graph), it is applied to the vector of all threads (typically hundreds and thousands) waiting to execute it. The thread vector of each block is constructed during the execution of prior basic blocks, as each thread that executes a basic block adds itself to the thread vector of the next block it needs to execute. The vector also comprises threads that reached it via different control flows.The hybrid VGIW machine thus enjoys the performance and power efficiency of the dataflow model and the generality of the von Neumann model, but without suffering performance degradation due to control divergence.

Figure 2 depicts an abstract VGIW machine (and the execution flow of a simple kernel). The machine consists of a *multithreaded, coarse-grain reconfigurable fabric* (MT-CGRF) execution core, a *control vector cache*, a *live value cache*, and a basic block scheduler.

Multithreaded, Coarse-Grain Reconfigurable Fabric. The MT-CGRF core executes multiple instances of a dataflow graph with high performance and low power characteristics [5]. The core comprises a host of interconnected functional units (e.g., arithmetic logical units, floating point units, load/store units). Its architecture is described in Section 3.5. The core employs dynamic, tagged-token dataflow [13, 14] to prevent memory stalled threads from blocking other threads, thereby maximizing the utilization of the functional units.

Prior to executing a basic block, the functional units and interconnect are configured to execute a dataflow graph that consists of one or more replicas of the basic block's dataflow graph. Replicating the basic block's dataflow graph enables the architecture to better utilize the MT-CGRF grid. The configuration process itself, described in Section 3.2, is lightweight and has negligible impact on system performance. Once configured, threads are streamed through the dataflow core by injecting their thread identifiers and CUDA ThreadIDX coordinates into special *control vector units.* When those values are delivered as operands to successor functional units, they initiate the thread's computation, following the dataflow firing rule. A new thread can thus be injected into the computational fabric on every cycle.

Control Vector Table (CVT). The CVT maintains a list of threads whose control flow has reached a basic block in the kernel. When that basic block is scheduled for execution and the MT-CGRF core has been configured with its dataflow graph, all its pending threads are streamed through the core. The table is updated dynamically. When the core finishes executing a thread's current basic block and its next basic block is determined, the core adds the thread's identifier to the destination block's entry in the table.

Live Value Cache (LVC). The LVC stores intermediate values that are communicated across basic blocks. The compiler statically allocates IDs for live values that are generated by one block and consumed by another. Although the LVC's functionality is analogous to that of a register file, it is accessed much less frequently. This is because most intermediate values are confined to a basic block and are communicated directly through the MT-CGRF core. Only the small fraction of intermediate values that need to be communicated across basic blocks must be stored in the LVC.
Figure 3 depicts this difference in access frequency. It illustrates the number of times the LVC is accessed by the MT-CGRF core (which may host multiple replicas of the dataflow graph) as a fraction of the number of times a GPGPU register file is accessed for the same kernel (counting a single access for an entire warp). The figure shows that the LVC is accessed on average almost 10× less frequently than a GPGPU register file. The difference in access frequency provides the design and execution flexibility that is paramount to the control flow coalescing execution model. In particular, it decouples the grouping of threads from any restrictions that may be imposed by the layout of their intermediate values inside the LVC. For example, it allows the LVC to be accessed at word granularity, in contrast to a GPGPU's vector register file.
This design flexiblity makes it possible to implement the LVC using a cache structure and allows live values to be spilled to memory. This is necessary due to the large storage footprint of all the threads' live values, as each kernel may require dozens of live values, and the core may concurrently execute thousands of threads. All live values are thus mapped to memory as a two-dimensional array that is indexed by the live value ID (row) and thread ID (column). The LVC caches the live value array. It is backed by the L2 cache and accessed using the live value ID and thread ID.

Basic Block Scheduler (BBS). The BBS schedules basic blocks (or dataflow graphs thereof) on the MT-CGRF core. Once the block is selected, the scheduler configures the MT-CGRF core with its dataflow graph and begins streaming the pending threads through the core.

Control flow coalescing on a VGIW machine

Figure 2 illustrates *control flow coalescing* by describing the step-by-step execution of a simple divergent kernel (depicted in Figure 1a). For brevity, we illustrate the execution of the kernel on eight threads only.
As all threads begin with the execution of the kernel's *entry basic block*, BB1, the block's execution is coalesced (Figure 2a). When each thread completes the execution of BB1, it registers the next block on its control path in the control vector table. After block BB1 completes executing all threads (Figure 2b), their control flow diverges. Threads 1,3,8 are registered to execute block BB2, whereas threads 2,4-7 are registered to execute BB3. The scheduler then selects block BB2 for execution. It configures the MT-CGRF core with BB2's dataflow graph and streams threads 1,3,8 through the core. When the threads complete their execution (Figure 2c), they all register to execute block, BB6. Next, block BB3 is selected and configured, and threads 2,4–7 are streamed through the core. When the block completes (Figure 2d), the control flow diverges again as threads 2,7 register to execute BB4 and threads 4-6 register to execute BB5. After executing BB4 (Figure 2e) and BB5 (Figure 2f), all the threads converge back to execute the kernel's exit block BB6.
Control flow coalescing enables the VGIW architecture to better utilize its execution resources. As Figure 1d illustrates, VGIW dynamically forms thread vectors, or warps, that consist of all threads that reached a specific basic block, and only executes the block for its thread vector. Since the execution model targets massive parallelism, the time spent executing each basic block is linear with the size of its thread vector. Importantly, thread grouping does not depend of the control path that each individual thread took to reach a given block, and each thread vector may represent multiple control paths. As a result, the number of reconfiguration depends on the number of basic blocks rather than the number of diverging control paths through the kernel. Given that VGIW is tuned to massive parallelism (as are GPGPUs), thread vectors are typically large.
Reconfigurations of the MT-CGRF core are, therefore, infrequent, and the optimized reconfiguration process only incurs a negligible overhead. In our prototype, for example, reconfiguration only takes 34 cycles, as described in Section 3.2. Consequently, the hybrid design outperforms other GPGPU designs while preserving the generality of the von Neumann model.
Replicating the dataflow graph of individual basic blocks across the MT-CGRF grid is another key contributor to the performance of the VGIW model. The replication multiplies the core's throughput and greatly reduces the execution time of a basic block's thread vector.
Notably, although we only illustrate the execution model using a simple example, the block-by-block execution naturally supports loops. When executing a loop, the threads that need to iterate through the loop are added to the thread vector of the first basic block in the loop body, whereas threads that exit the loop are added to the thread vector of the loop's *epilogue basic block.*

3. THE VGIW ARCHITECTURE

This section presents the hybrid dataflow/von Neumann VGIW architecture and its individual components.

The organization of the VGIW architecture is shown in Figure 4, which also illustrates the high-level interactions between the different components. As described in the abstract machine model (Section 2), the architecture is composed of a basic block scheduler (BBS), a control vector table (CVT), a live value cache (LVC), and an MT-CGRF execution core. A VGIW processor core is connected to a conventional GPGPU memory system through a banked L1 cache.

3.1 Compiling for VGIW

The VGIW compiler partitions a CUDA kernel into basic blocks, generates a control flow graph, and determines the scheduling of basic blocks. The schedule preserves control dependencies, and each block is assigned a unique block ID based on the scheduling order. Each block encodes the block IDs of its successor blocks (up to two), and loops manifest when the block ID of the successor is smaller than that of the original block. This scheduling scheme simplifies the runtime (hardware) scheduler, which simply selects the smallest block ID whose thread vector is not empty. The compiler generates a control flow graph with a single entry point by generating an entry basic block. The entry block uses the reserved block ID 0, which guarantees that all threads execute it first.

The compiler assigns a live value ID for each intermediate value that crosses block boundaries and encodes the IDs in the respective blocks' dataflow graphs. The mapping process is similar to traditional register allocation, and threads access the LVC using the live value ID and the thread ID.

Finally, each basic block is converted into a dataflow graph and undergoes a place and route sequence to generate a static per-block configuration of the MT-CGRF core. For small basic blocks, the compiler includes multiple replicas of a block's graph in the generated configuration. This maximizes the core's utilization and increases thread-level parallelism, as shown in Figure 1d.

3.2 Basic block scheduler (BBS)

The basic block scheduler (BBS), depicted in Figure 5, controls the kernel's execution by selecting the next basic block to execute, configuring the MT-CGRF core to run it, and sending thread IDs to the core. The scheduler also serves as a frontend to the CVT. The BBS reads basic block vectors from the CVT to send them to the core and, conversely, updates the block vectors in the CVT based on the resolved branch information it receives from the MT-CGRF core.

The finite size of the CVT imposes a limit on the number of threads that can be tracked at any given time. The number of live threads is thus limited by tiling their execution, and the BBS sets the tile size based on the number basic blocks in the kernel:

$$tile\,size = \frac{CVT\,size}{\#basic\,blocks \times \#CUDA\,threads} .$$

To execute a kernel, the runtime software loads its basic block sequence to the BBS, along with the per-block configuration of the MT-CGRF core. The runtime then signals the BBS to set all bits in the entry block vector (block ID 0). Finally, the BBS begins sending batches of thread IDs to the core for execution (thread batches are analogue to warps on regular GPGPUs).

Thread batches are sent as $\langle base\,threadID, bitmap\rangle$ tuples, where *base thread ID* represents the first thread in the batch (first bit in the bitmap), and the bitmap indicates which of the consecutively subsequent IDs are scheduled to execute the current basic block. Each packet consists of a 16-bit thread ID and a 64-bit bitmap. When the BBS sends a thread batch packet to the core, it zeros the corresponding bits in the CVT. Conversely, when a batch of threads finish a basic block, the core sends the BBS two batch packets, one for each of the block's successors. The BBS updates the CVT by ORing the bitmaps received from the core with the existing data in the CVT. An OR operation is required since a block may be reached by multiple control flows.

During the execution of a block, the BBS prefetches the configurations of the following blocks into a *configuration FIFO*. Once the execution of all threads in the current block completes, a reset signal is sent to the nodes in the grid. The reset clears the nodes and configures the switches to pass tokens from left to right. The BBS then feeds the configuration tokens to the grid from its left perimeter. This process takes 11 cycles $sqrt(\#nodes)$ and is performed twice to deliver all the configuration data. For the kernels evaluated in this paper, the total configuration overhead averaged at 0.18% of the runtime with a median lower than 0.1%.

3.3 Control vector table (CVT)

The CVT is depicted in Figure 2. It associates each basic block ID with a bit vector that is indexed by thread IDs. A set bit indicates that the corresponding thread ID should execute that basic block next. Naturally, a thread ID's bit can only be set in one of the table entries at any given time. The CVT delivers 64-bit words.

The BBS may access the CVT with both reads and writes in the same cycle. The CVT thus provides both read and write ports. The structure uses a read-and-reset policy that resets words when they are read. This design is used to avoid adding an extra write port. Since a dataflow graph may be replicated in the MT-CGRF core, the BBS may need to perform multiple read/write operations per cycle. The CVT structure is thus partitioned into 8 banks to facilitate ample parallelism.

3.4 Live value cache (LVC)

The LVC stores live values that are transfered across executed basic blocks. The LVC caches a memory resident, two-dimensional matrix that maps all live values and is indexed by $\langle live\,valueID, threadID\rangle$ tuples. *Live value IDs* are set at compile time, while thread IDs are determined at runtime and are sent between MT-CGRF functional units as part of each data token. The LVC is implemented as a banked cache (similar to a GPGPU L1 design [15] and is backed by the memory

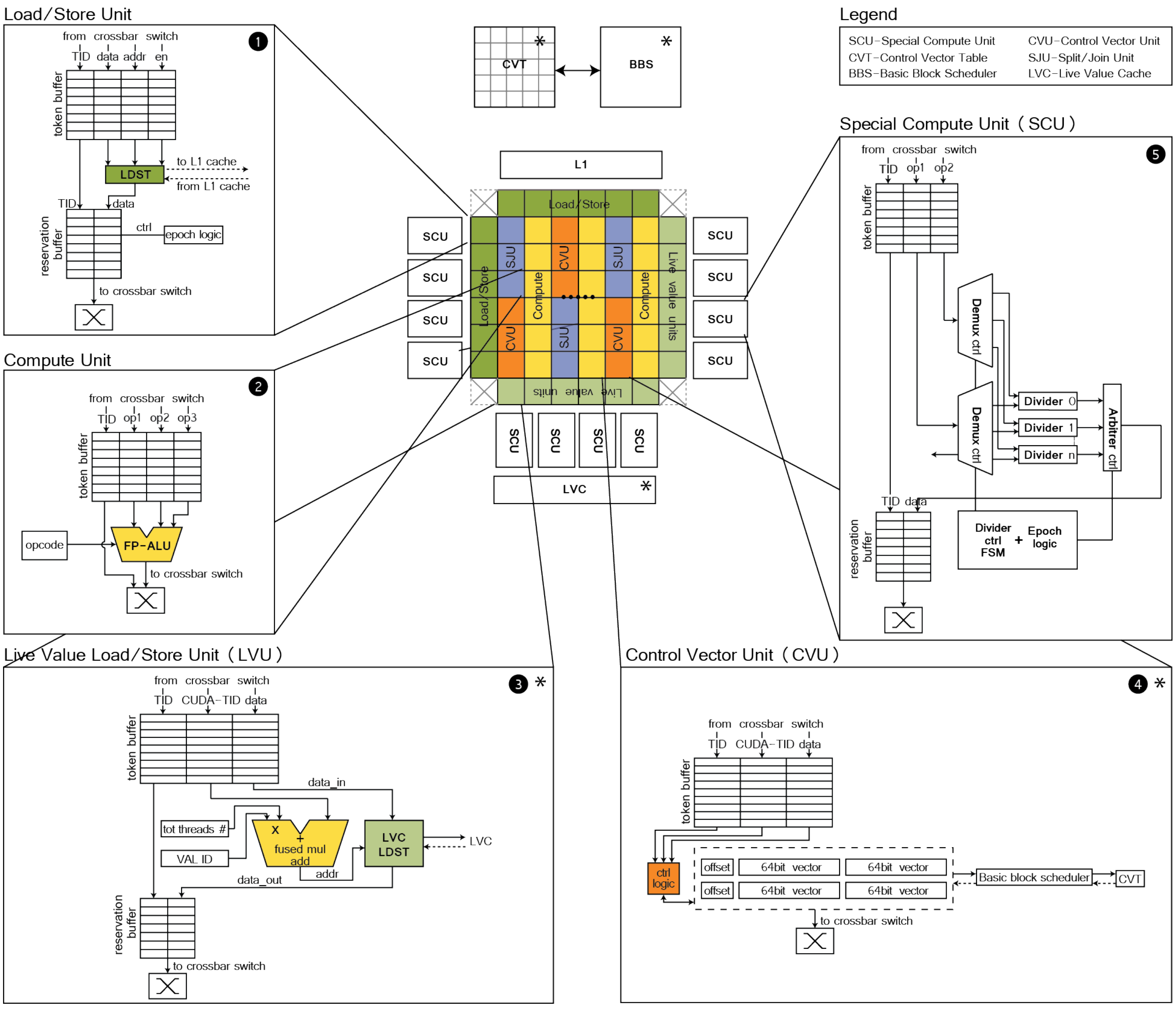


Figure 4: The VGIW core. Components that do not exist in the SGMF design are marked with $*$.

system (L2 cache, main memory). This design simplifies the management of the LVC since it supports spilling intermediate values to memory if the LVC is contended (although this is generally prevented by thread tiling). For brevity, we do not present a full design space exploration of the LVC size and only show results for a 64KB LVC. In comparison, this size is 4× smaller than the register file on the NVIDIA Fermi GPGPU. Together with the total capacity of the MT-CGRF core buffers, which is 70% smaller than the RF on an NVIDIA Fermi [5], the total amount of the VGIW in-core storage is 2× smaller than the NVIDIA Fermi RF.

3.5 The MT-CGRF execution core

The MT-CGRF execution core comprises a grid of interconnected functional units that communicate using data and control tokens. The grid is configured with a basic block's dataflow graph; it can then concurrently stream multiple instances (threads) of the said graph. The core combines pipelining, or static dataflow [16], to extract parallelism inside a thread, and out-of-order execution (or dynamic dataflow [13, 14]) across threads to enable threads that stall on cache misses to be passed over by unblocked threads. The MT-CGRF design extends the original CGRF dataflow core proposed by Voitsechov and Etsion [5]. This section discusses the core's design and inner workings (a more comprehensive discussion can be found in the original SGMF paper [5]).

Each unit in the MT-CGRF fabric includes a token buffer and a statically reconfigurable switch that connects the unit to eight neighboring units. The token buffer enables each unit to be multiplexed among distinct threads. Functional units are multiplexed using virtual execution channels, which operate in a manner similar to virtual network channels. Each entry in the token buffer stores the operands needed for the unit to execute (up to three operands). When all the operands in an entry are available, it is marked as ready and

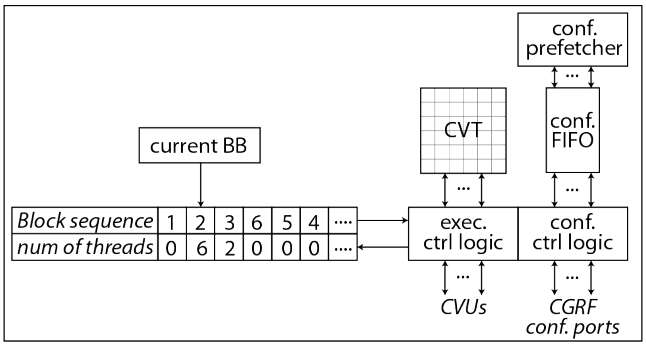


Figure 5: The basic block scheduler (BBS). A kernel is loaded to the BBS as a sequence of basic block IDs. The BBS then schedules the blocks dynamically based on the kernel's control flow.

will be executed. The index of each entry in the token buffer associates it with a virtual execution channel. Each thread that is streamed through the core is assigned a virtual channel ID, and the thread's data and control tokens use that ID to index the token buffer in the destination unit. Consequently, each token buffer entry serves a distinct thread and, when the entry is marked ready, the functional unit executes its preconfigured operation using the thread's operands.

The direct communication of intermediate values across functional units eliminates the need for a central register file, which has been shown to be a major power bottleneck in GPGPUs [3, 4]. Only intermediate values that are communicated across basic blocks (namely those whose lifetime extends that of their originating basic block) are stored in temporary storage — the LVC.

The different functional units contain configuration registers that carry the opcode they need to execute, along with any static parameters. The configuration registers are initialized every time a kernel is loaded to the execution core. To accelerate this process, configuration messages are sent along each row in the grid, reducing the configuration time to $O(\sqrt(N))$, where N is the number of functional units in the grid.

Finally, another common VGIW optimization is basic block replication. As the size of basic blocks' control flow graphs is often smaller than the number of units in the MT-CGRF core, the compiler maximizes the core's utilization and throughput by mapping multiple replicas of a basic block to the core.

The rest of this section discusses the core's six types of functional units and its interconnect: 1. Compute units (FP-ALU). 2. Load/store units (LDSTU). 3. Split/join units (SJU) . 4. Special compute units (SCU). 5. Live value load/store units (LVU). 6. Control vector units (CVU). The circled numbers correspond to the functional units as illustrated in Figure 4.

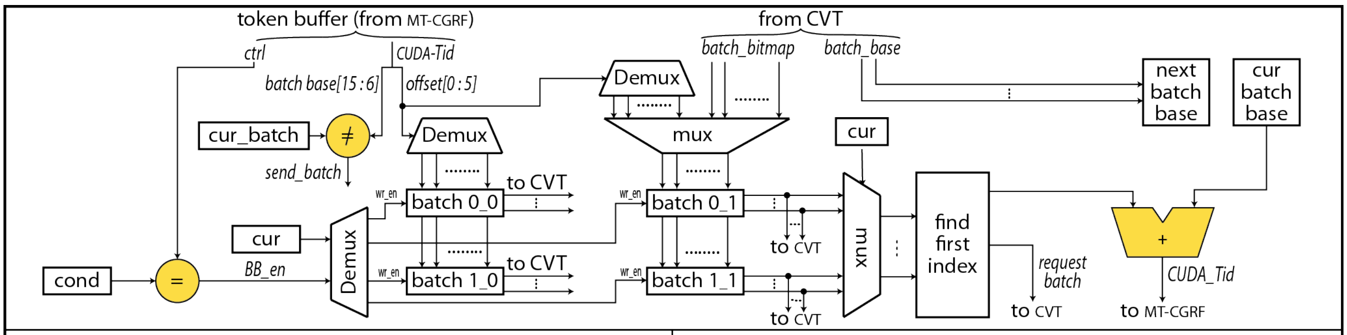
Compute units ❷. Each compute unit merges a floating point unit (FPU) and an arithmetic logic unit (ALU). The unified unit is configured by a single op-code. Merging an ALU and an FPU into a single unit saves 10% in power and 13% in area. All the instructions supported by the unit are pipelined, so it can process a new set of tokens at each cycle. All the non-pipelined components are clustered into the SCUs described below.

Load/store units ❶. Load/store units (LDSTU) connect the MT-CRGF to the memory system through the L1 cache. The units are located on the grid perimeter and are connected to the banked L1 cache through a crossbar switch. Each LDSTU includes a reservation buffer that enables threads to execute out-of-order using dynamic dataflow. The buffer maintains the thread IDs associated with active memory operations. When an operation completes, the unit that maps the successor node in the dataflow graph is signaled (using a data token) that the operand of the unblocked thread is ready. By enabling threads to execute out-of-order and overtake blocked ones, the reservation buffers greatly improve the utilization of the MT-CRGF.

Split/join units. The split/join units (SJUs) provide two, reciprocal functionalities. The *split* functionality sends a single input operand to multiple successor units; it is used to extend nodes with large fanouts that are limited by the interconnect. The *join* functionality, in contrast, provides a primitive that preserves the ordering of memory operations inside each thread (memory operations issued from distinct threads can be safely reordered since threads are data-parallel). A join operation waits for control tokens sent from predecessor nodes and, once those arrive, sends a control token to one or more successor nodes. For example, a store operation must not be issued before all loads that precede it in the original program order have completed (write-after-read). A join operation is thus placed in the dataflow graph between the store and its preceding loads. Only when the join receives control tokens from all preceding loads does it send a token to the successor store operation, and allows it to fire.

Special compute units ❺. The special compute units (SCUs) provide virtual pipelining for non-pipelined arithmetic and floating point operations (e.g., division, square-root). These units are composed of multiple instances of the circuits that implement the non-pipelined operations. When a new operation is issued, the unit selects an available instance to execute the operation. The units thus enable a new non-pipelined operation to begin execution on each cycle. Notably, the SCUs do not consume additional space compared to regular GPGPUs, since the compute units in the MT-CRGF core do not include any non-pipelined elements. SCUs thereby aggregate the non-pipelined elements that are available on each ALU/FPU on regular GPGPUs.

Live value load/store units ❸. The LVUs store and retrieve intermediate values to/from the LVC. A configuration register inside each LVU stores the live value ID that the unit addresses. As discussed above, LVUs access the LVC using $\langle live\,valueID, threadID\rangle$ tuples. The LVUs are located on the perimeter of the MT-CRGF core, alongside the LDSTU.



CVU - as thread terminator

```
At arrival of (CUDA-Tid,ctrl) {
        if (batch_base != cur_batch)
                send_batch <= '1;
        else
                send_batch <= '0;
        When(send_batch == '1){
                Send to CVT (batch_<cur>_0,batch_<cur>_1);
                cur!=cur;
                cur_batch<=batch_base;
        }
        batch_<BB_en>_<cur>|=in;
}
```

CVU - as thread initiator

```
ffs = find_first_indx(batch_<cur>_1);
if(ffs==NULL¹) {
        request_batch<='1;
        cur!=cur;
        cur_batch<=next_batch_base;
}
else {
        CUDA_Tid = cur_batch_base + ffs;
        Send to CGRF CUDA_Tid;
}
At arrival of (batch_base,batch_bitmap){
        next_batch_base<=batch_base;
        batch_<!cur>_1 <=batch_bitmap;
}
```

¹-NULL means no 1's found in the batch.

Figure 6: The control vector unit (CVU) when functioning as a thread initiator/terminator.

Control vector units ❹. Each CVU can function both as a *thread initiator* or as a *thread terminator* (Figure 6). Each replica of a basic block's dataflow graph is assigned an initiator CVU and a terminator CVU. When functioning as a *thread initiator*, the CVU receives thread batches from the BBS. It then computes the CUDA ThreadIDX coordinates for the initiated threads and sends them to its successor nodes. Following the dataflow firing rule, once the successor nodes receive their respective coordinates, they execute. When a basic block is replicated in the MT-CRGF (which is often used by the compiler to maximize the core's utilization), each replica is assigned an initiator CVU.
As described in Section 3.2, thread batches are communicated as $\langle basethreadID, bitmap\rangle$ tuples. When a batch arrives, the CVU begins looping over the bitmap to identify the thread IDs it should initiate (by adding the set bits' indices to the base thread ID). To avoid stalls, CVUs use double buffering of thread batches. Whenever a batch is received, the CVU immediately requests the next batch from the BBS.
Conversely, when a CVU functions as a *thread terminator*, it executes the basic block's terminating branch instruction to determine the next basic block that the thread should execute. The destination block IDs (up to two) are stored in the CVU's configuration register. At runtime, the input token to the CVU determines which of the two targets should be executed next. The CVU maintains two thread batches, one for each possible target block ID and adds each thread to the batch that corresponds with its branch outcome. Notably, since threads are executed out-of-order, thread IDs from different batches may be interleaved. To support threads that complete out-of-order, the CVU maintains a pair of thread batches for each destination block ID. Once the CVU encounters a thread ID from a different batch, it sends the current (possibly partial) batch to the BBS. In total, a CVU maintains storage for four batches (only two are used when functioning as a thread initiator). Each batch is maintained using an 80-bit register (16-bit thread ID + 64-bit bitmap), for a total of 320 bits.

Interconnect. The topology of the interconnect is designed to meet three key requirements: a hop latency of one cycle; reducing the number of hops between non-adjacent units; and equalizing the connectivity of the perimeter units (LDSTU and the LVUs) to that of the units inside the grid. To meet these requirements, the interconnect uses a folded hypercube topology [17]. Each functional unit is connected to its four nearest units and four nearest switches. The switches are also connected to the four switches with a Manhattan distance of two. This topology equalizes the connectivity on the perimeter and reduces the average number of hops between functional units. Finally, the topology is known to scale with the size of the grid.

3.6 The memory system

The memory system of the VGIW is almost identical to that used by the NVIDIA Fermi. A VGIW core has a 64KB, 32-bank L1 cache (with 32 banks), a 768KB, 6-bank L2 cache, and a 16-bank, 6-channel GDDR5 main memory. The only difference between the memory systems is that the VGIW caches use a write-back policy instead of Fermi's write-through policy.

4. METHODOLOGY

The amount of logic that is found in a VGIW core is approximately the same amount that is found in a Nvidia SM and in a SGMF core. In a Nvidia SM, that logic

Parameter	Value
VGIW Core	108 interconnected func./LDST/control units
Functional units	32 combined FPU-ALU units
	12 Special Compute units
Load/Store units	16 Live Value Units
	16 regular LDST units
Control units	16 Split/Join units
	16 Control Vector Units
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: VGIW system configuration.

assembles 32 CUDA cores, while in the VGIW core the CUDA cores are broken down into smaller coarse grained blocks. The breakdown to smaller granularity enables greater parallelism and increases performance.

RTL Implementation. We implemented the major components of the VGIW architecture in Verilog (including the interconnect) to evaluate their 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.

Simulation framework. We used the GPGPU-Sim simulator [6] and GPUWattch [4] power model (which uses performance monitors to estimate the total execution energy) to evaluate the performance and power of the VGIW design. These tools model the Nvidia GTX480 card, which is based on the Nvidia Fermi. We extended GPGPU-Sim to simulate a VGIW core and, using per-operation energy estimates obtained from the RTL place&route results, we extended the power model of GPUWattch to support the VGIW design.
The system configuration is shown in Table 1. By replacing the Fermi SM with a VGIW core, we retain the uncore components. The only difference between the processors' memory systems is that VGIW uses write-back and write-allocate policies in the L1 caches, as opposed to Fermi's write-through and write-no-allocate.

Compiler. We compiled CUDA kernels using LLVM [18] and extracted their SSA [19] code. This was then used to configure the VGIW grid and interconnect.

Benchmarks. We evaluated the VGIW architecture using kernels from the Rodinia benchmark suite [7], listed in Table 2. Importantly, the benchmarks were used as-is and are optimized for SIMT processors.

5. EVALUATION

This section evaluates the performance and power efficiency of the VGIW architecture and compares them to those delivered by NVIDIA Fermi and an SGMF core.

Performance analysis.
Figure 7 shows the speedup achieved by a VGIW core over a Fermi SM. Results range between 0.9× (slowdown) and 11× speedup, with an average of over 3×.

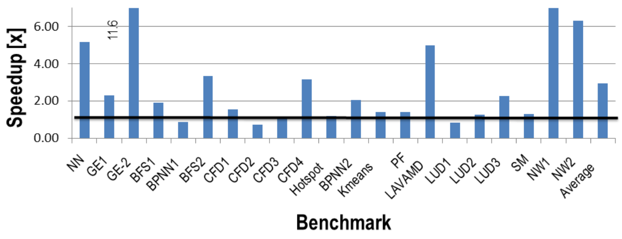


Figure 7: Speedup of VGIW over a Fermi.

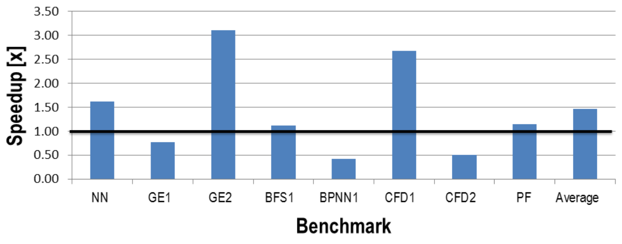


Figure 8: Speedup of VGIW over SGMF.

Although the kernels' performance depends on their particular features, the average 3× speedup corresponds to the increased ILP delivered by the spatial VGIW core. While the Fermi's von Neumann design can only operate 32 of its functional units concurrently (leaving the rest idle), the VGIW spatial design can operate *all* its 108 functional units concurrently.
When examining the results, we divide the kernels into two main categories: computational kernels and memory bound kernels. The VGIW architecture is highly efficient for multithreaded computational kernels, which benefit from the increased ILP delivered by the spatial design. Memory bound kernels, on the other hand, do not fully utilize the grid, and VGIW delivers performance comparable to that of a GPGPU. Occasionally, memory intensive kernels that are optimized for GPGPUs will exhibit slowdown on our system, e.g., the CFD3 kernel that simply moves data from one array to another. Even though VGIW does not perform memory coalescing, its high ILP and inter-thread dynamic dataflow help mask the latencies caused by the cache bank conflicts. We leave the exploration of methods for memory coalescing on MT-CGRFs for future work.
Figure 8 shows the speedup achieved by the VGIW architecture over the SGMF design. Since the SGMF architecture maps the entire kernel into the MT-CGRF grid, it can only execute small to medium kernels with simple control flows. The comparison is thus based on the subset of kernels that can be mapped to the SGMF cores. The figure shows that the performance of VGIW is comparable to that of SGMF. While the average speedup is better than 1.45×, that of the individual kernels varies between 0.4× and 3.1×. SGMF excels with kernels characterized by small basic blocks and a small amount of branch divergence. For these kernels, the overheads of the grid reconfiguration and live value storage thwart the utilization gains of the VGIW.

Application	Application Domain	Kernels (#basic blocks)	Description
BFS	Graph Algorithms	*Kernel(8) ,Kernel2(3)*	Breadth-first search
KMEANS	Data Mining	*invert_mapping(3)*	Clustering algorithm
CFD	Fluid Dynamics	*compute_step_factor(2) ,initialize_variables(1) time_step(1) ,compute_flux(12)*	Computational fluid dynamics solver
LUD	Linear Algebra	*lud_internal(3), lud_diagonal(11), lud_perimiter(22)*	Matrix decomposition
GE	Linear Algebra	*Fan1(2) ,Fan2(5)*	Gaussian elimination
HOTSPOT	Physics Simulation	*hotspot_kernel(27)*	Thermal simulation tool
LAVAMD	Molecular Dynamics	*kernel_gpu_cuda(21)*	Calculation of particle position
NN	Data Mining	*euclid(2)*	K nearest neighbors
PF	Medical Imaging	*normalize_weights_kernel(5)*	Particle filter (target estimator)
BPNN	Pattern Recognition	*adjust_weights(3) ,layerforward(20)*	Training of a neural network
NW	Bioinformatics	*needle_cuda_shared_1(13), needle_cuda_shared_2(13)*	Comparing biological sequences
SM	Data Mining	*streamcluster_kernel_compute_cost(6)*	Clustering algorithm

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

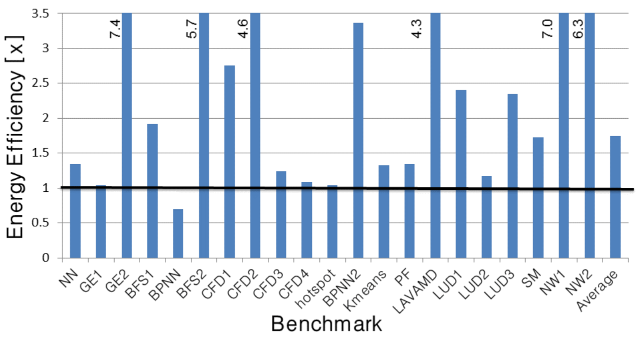


Figure 9: Energy efficiency of a VGIW core over a Fermi SM.

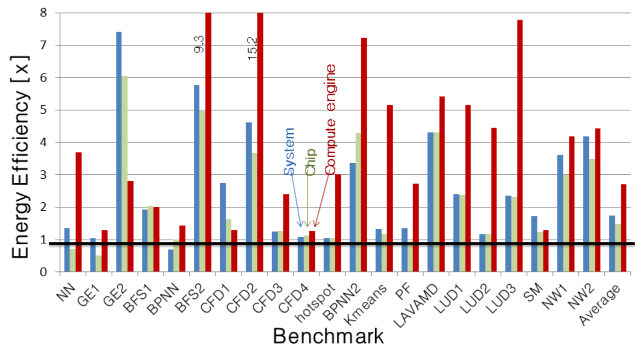


Figure 10: The energy efficiency of VGIW over Fermi at the system, die, and core levels.

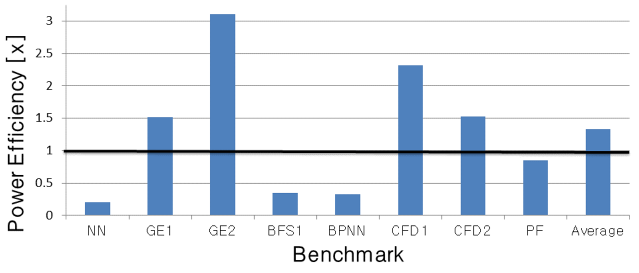


Figure 11: Energy efficiency of a VGIW core over an SGMF core.

Energy efficiency analysis.

In this section we compare the energy efficiency of the evaluated architectures. Our evaluation compares the total energy required to do the work, namely execute the kernel, since the different architectures use different *instruction set architectures* (ISAs) and execute a different number of instructions for the same kernel. We therefore define power efficiency as:

$$\frac{performance}{watt} = \frac{work}{time} \Big/ \frac{energy}{time} = \frac{work}{energy}.$$

Figure 9 shows that the VGIW architecture is 1.75× (∼40%) more energy efficient, on average, than the baseline Fermi architecture. The figure shows that the efficiency of different kernels varies between 0.7× and 7×. The figure also shows a strong correlation between a kernel's characteristics and its energy efficiency benefits, indicating that computational kernels execute much more efficiently on the VGIW architecture.

Figure 10 compares the energy efficiency of the VGIW and Fermi architectures at the level of the entire system (core, L1 and L2 caches, memory controller/interconnect and DRAM), die (core, L1 and L2 caches and memory controller/interconnect), and core (compute engine). The energy of the VGIW core includes the energy spent on the LVC and CVT, and that of Fermi includes the register file. The figure demonstrates that the improved efficiency of VGIW is attributed its efficient compute-engine. It also motivates further research on power efficient memory systems for the VGIW model.

Finally, Figure 11 compares the energy efficiency of VGIW and SGMF for the SGMF-supported kernels. The figure shows that while VGIW is 1.33× (or ∼25%) more efficient than SGMF on average, the results vary between kernels. SGMF excels at small kernels with a little branch divergence, since passing live values through the LVC is more wasteful than passing them directly inside the CGRF. We believe that these results motivate further research on block size optimization for MT-CGRFs cores. Nevertheless, for larger and more complex kernels, especially ones that suffer from branch divergence, the energy saved by VGIW architecture is significant; moreover, VGIW can execute kernels of any size and with any control flow complexity.

To conclude, the evaluation shows the performance and power benefits of the hybrid dataflow/von Neumann VGIW architecture over a von Neumann GPGPU (NVIDIA

Fermi) and a dataflow GPGPU (SGMF).

6. RELATED WORK

Hybrid dataflow/von Neumann architectures. The potential benefits of hybrid dataflow/von Neumann architectures have prompted multiple studies, including TRIPS [20], WaveScalar [21,22], Tartan [23], DySER [24], SEED [25], MAD [26], BERET [27], Garp [28], Dataflow Mini-Graphs [29], Triggered instructions [30], and SGMF [5]. With the exception of SGMF, these designs focus on single-thread workloads rather than data-parallel ones. TRIPS, WaveScalar and Tartan share some common execution characteristics (albeit with very different designs). Programs are partitioned into hyperblocks that are dynamically scheduled to execution nodes, which comprise of simple cores (TRIPS, WaveScalar) or a CGRF element (Tartan [31]). The nodes dynamically schedule hyperblocks according to their data dependencies. TRIPS and WaveScalar also support simultaneous thread execution. TRIPS dynamically schedules threads' hyperblocks on different grid tiles (spatial paritioning), but hyperblock executes individually. Furthermore, by tracking next hyperblock to execute for each thread TRIPS effectively provides a simple form of control flow coalescing. WaveScalar pipelines multiple instances of hyperblocks originating from a single thread, while its WaveCache extension [22] supports pipelining of hyperblocks originating from different threads. The proposed VGIW, on the other hand, simultaneously executes multiple threads on the MT-CGRF core, and time-multiplexes it between the basic blocks.
Other studies employ reconfigurable fabrics as accelerators that execute CDFGs one at a time. Garp [28] adds a CGRF component to a MIPS core in order to accelerate dataflow-friendly loops. DySER [24], SEED [25], and MAD [26] add some CGRF functionality to an out-of-order processor, and the compiler maps code sections to the CGRF. While DySER is designed to be a dataflow functional unit in the out-of-order processor, SEED and MAD extend the core with a dataflow execution engine that can take over when dataflow-friendly code is executed. Finally, the Dataflow Mini-Graphs design [29] improves the performance of out-of-order processors by extracting single-input, single-output dataflow graphs from sequential code and executing them (out-of-order) on a dedicated ALU pipeline.
In contrast, SGMF [5] explicitly targets massively data-parallel workloads as a dataflow GPGPU. SGMF, however, is limited to small kernels that fit in its reconfigurable fabric, and suffers from inefficient use of spatial resources in the presence of control flow divergence.

Branch divergence on GPGPUs. Branch divergence in GPGPUs has been extensively studied, yielding numerous techniques to alleviate the problem. For brevity, we only discuss a few representative ones.
Fung et al. [8] propose to fuse sparse divergent warps into denser warps using a reconvergence stack. In a later work, Fung and Aamodt [9] propose *thread block compaction*, which breaks down fused warps after the control flow converges in order to benefit from the coalesced layout of data in memory. Meng et al. [10] propose to dynamically split divergent warps and interleave their execution, thereby leveraging control flow divergence to hide memory latencies. Narasiman et al. [11] propose two-level warp scheduling, in which warps are dynamically derived from larger, possibly divergent warps. Rhu and Erez [12] propose CAPRI, which improves on thread block compaction techniques [9,11] by predicting which threads are likely to diverge, thereby eliminating a synchronization point at the point of divergence.
The abovementioned studies alleviate the control flow divergence problem on contemporary GPGPUs and show an average performance improvement of $\sim$20%. Importantly, they only aim to improve existing GPGPUs and do not challenge their fundamental execution model.

Miscellaneous.
The Vector-Thread Architecture [32] can operate in either vector mode, in which all compute elements execute the same code for different data streams, or in thread mode, in which the elements execute different basic blocks. When control paths diverge, the architecture switches from vector to thread mode to decouple the execution of diverging threads.
XLOOPS [33] provides hardware mechanisms to transfer loop-carried dependencies across cores. This processor relies on programmer annotation of dependency types to synchronize the data transfers.
CAWA [34] addresses the execution time divergence across GPGPU warps. It proposes a predictor, scheduler, and cache reuse predictor to accelerate lagging warps. VGIW addresses warp execution time divergence by decoupling the execution of threads using inter-thread dynamic dataflow. Nevertheless, most of the optimizations proposed are orthogonal to the execution model and can be applied to VGIW as well.

7. CONCLUSIONS

We presented the hybrid dataflow/von Neumann *vector graph instruction word* (VGIW) architecture. This data-parallel architecture executes basic blocks' dataflow graphs using the *multithreaded, coarse-grain, reconfigurable fabric* (MT-CGRF) dataflow execution engine, and employs von Neumann control flow semantics to schedule basic blocks.
The proposed architecture dynamically coalesces threads that wait to execute each basic block into thread vectors, and executes each block's thread vector using the MT-CGRF compute engine. This *control flow coalescing* enables the VGIW architecture to overcome the control divergence problem, which impedes the performance and power efficiency of data-parallel architectures. Furthermore, maintaining von Neumann semantics enables the VGIW architecture to overcome the limitations of the recently proposed *single-graph multiple-flows* (SGMF) dataflow GPGPU, which is limited in the size of the kernels it can execute.
Our results show that the VGIW architecture outper-

forms the NVIDIA Fermi architecture by 3× on average (up to 11×) and provides an average 1.75× better energy efficiency (up to 7×).

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).

8. REFERENCES

[1] "Top 500 supercomputer sites." www.top500.org, Nov 2014.

[2] "The green 500 list." www.green500.org, Nov 2014.

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

[4] 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.

[5] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for gpgpus," in *Intl. Symp. on Computer Architecture (ISCA)*, 2014.

[6] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator.," in *IEEE Intl. Symp. on Perf. Analysis of Systems and Software (ISPASS)*, 2009.

[7] 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)*, 2009.

[8] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec 2007.

[9] W. W. L. Fung and T. M. Aamodt, "Thread block compaction for efficient simt control flow," in *Symp. on High-Performance Computer Architecture (HPCA)*, 2011.

[10] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Intl. Symp. on Computer Architecture (ISCA)*, 2010.

[11] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Intl. Symp. on Microarchitecture (MICRO)*, 2011.

[12] M. Rhu and M. Erez, "CAPRI: Prediction of compaction-adequacy for handling control-divergence in GPGPU architectures," in *Intl. Symp. on Computer Architecture (ISCA)*, 2012.

[13] Arvind and R. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Trans. on Computers*, vol. 39, pp. 300–318, Mar 1990.

[14] Y. N. Patt, W. M. Hwu, and M. Shebanow, "HPS, a new microarchitecture: rationale and introduction," in *Intl. Symp. on Microarchitecture (MICRO)*, 1985.

[15] 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.

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

[17] A. El-Amawy and S. Latifi, "Properties and performance of folded hypercubes," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, pp. 31–42, Jan. 1991.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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)*, pp. 163–174, Oct 2006.

[24] 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.

[25] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," in *Intl. Symp. on Computer Architecture (ISCA)*, ACM, Jun 2015.

[26] C.-H. Ho, S. J. Kim, and K. Sankaralingam, "Efficient execution of memory access phases using dataflow specialization," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2015.

[27] 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.

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

[29] A. Bracy, P. Prahlad, and A. Roth, "Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec 2004.

[30] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, *et al.*, "Triggered instructions: A control paradigm for spatially-programmed architectures," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2013.

[31] 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, pp. 70–77, Apr. 2000.

[32] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The Vector-Thread architecture," in *Intl. Symp. on Computer Architecture (ISCA)*, 2004.

[33] S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, and C. Batten, "Architectural specialization for inter-iteration loop dependence patterns," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec 2014.

[34] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2015.