2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Inter-thread Communication in Multithreaded, Reconfigurable Coarse-grain Arrays

Dani Voitsechov[†] Oron Port[†] Yoav Etsion[†,§]
[†]Electrical Engineering [§]Computer Science
Technion–Israel Institute of Technology
{dimavoi,soronpo,yetsion}@technion.ac.il

***Abstract*—Traditional von Neumann GPGPUs only allow threads to communicate through memory on a group-to-group basis. In this model, a group of producer threads writes intermediate values to memory, which are read by a group of consumer threads after a barrier synchronization. To alleviate the memory bandwidth imposed by this method of communication, GPGPUs provide a small scratchpad memory that prevents intermediate values from overloading DRAM bandwidth.**

In this paper we introduce direct inter-thread communications for massively multithreaded CGRAs, where intermediate values are communicated directly through the compute fabric on a point-to-point basis. This method avoids the need to write values to memory, eliminates the need for a dedicated scratchpad, and avoids workgroup global barriers. We introduce our proposed extensions to the programming model (CUDA) and execution model, as well as the hardware primitives that facilitate the communication. Our simulations of Rodinia benchmarks running on the new system show that direct inter-thread communication provides an average speedup of 2.8× (10.3× max) and reduces system power by an average of 5× (22× max), when compared to an equivalent Nvidia GPGPU.

***Index Terms*—CGRA, dataflow, GPGPU, SIMD, MPI, reconfigurable -architectures,non-von Neumann-architectures,inter-thread communication**

I. Introduction

Conventional von Neumann GPGPUs employ the data-parallel *single-instruction multiple threads* (SIMT) model. But pure data parallelism can only go so far, and the majority of data parallel workloads require some form inter-thread communication. Common GPGPU programming models (e.g., CUDA, OpenCL) group threads into cooperative thread arrays (CTAs, or workgroups), which enable threads in a CTA to communicate using shared memory.

This model has two major limitations. First, communication is mediated by a shared memory region, typically implemented using a hardware scratchpad. It thus requires support for high communication bandwidth and as such is energy costly. The second limitation is the synchronization model. Since the order of scheduling of the threads within a CTA is unknown, a synchronization barrier must be invoked before consumer threads can read the values written to the shared memory by their respective producer threads.

As an alternative to von Neumann GPGPUs, Voitsechov and Etsion [1], [2] recently introduced a coarse-grained reconfigurable GPGPU architecture (CGRA) coupled with a dataflow execution model [3]–[5]. The proposed architecture, which we refer to as a *massively multithreaded CGRA* (MT-CGRA), maps the compute graph of CUDA kernels to a CGRA and uses the dynamic dataflow execution model to run multiple CUDA threads. The direct connectivity provided by the CGRA fabric is leveraged to eliminate multiple von Neumann bottlenecks, including the register file and instruction control. Nevertheless, because the MT-CGRA model is still bound by a shared memory and synchronization barriers for inter-thread communication, it incurs their power and performance overheads.

In this paper we present *dMT-CGRA*, an extension to MT-CGRA that supports direct inter-thread communication through the CGRA fabric. By extending the programming model, the execution model, and the underlying hardware, the new architecture forgoes the shared memory/scratchpad and global synchronization operations.

dMT-CGRA extends CUDA with two primitives that enable programmers to express direct inter-thread dependencies. The primitives let programmers state that thread N requires a value generated by thread $N-k$, for any arbitrary thread index N and a scalar k. The compiler maps the programming primitives to temporal links in the kernel's dataflow graph. The temporal links express dependencies between concurrently executing instances of the dataflow graph, each representing a different thread. Finally, two new functional units redirect dataflow tokens between graph instances (threads) such that the dataflow firing rule is preserved.

Our new architecture improves performance and energy consumption by leveraging three distinct properties:

- **Reduced memory bandwidth** Replacing the shared memory with direct in-CGRA token routing reduces the shared memory bandwidth and its associated energy consumption.
- **Inter-thread data reuse** Lightweight inter-thread communication allows threads to share data loaded from memory using in-CGRA routing, and thus alleviates shared memory traffic when serving as a software-managed cache.
- **Code compaction** Replacing shared memory with direct inter-thread communication eliminates code associated with address computations and boundary checks, and thus reduces the number of dynamic operations.

This research was supported by the Israel Science Foundation (979/17).

Overall, we show that dMT-CGRA outperforms NVIDIA GPGPUs by $3\times$ on average while consuming $5\times$ less energy. The remainder of this paper is organized as follows. Section II describes the motivation for direct inter-thread communication on an MT-CGRA and explains the rationale for the proposed design. Section III then presents the dMT-CGRA execution model and the proposed programming model extensions, and Section IV presents the dMT-CGRA architecture. We present our evaluation in Section V and discuss related work in Section VI. Section VII concludes the paper.

II. Inter-thread Communication in a Multithreaded CGRA

Modern massively multithreaded processors, namely GPGPUs, employ many von-Neumann processing units to deliver massive concurrency, and use shared memory (a scratchpad) as the primary means for inter-thread communication. This design imposes two major limitations:

1) The frequency of inter-thread communication barrages shared memory with intermediate results, requiring high bandwidth support for the dedicated scratchpad and dramatically increasing its power consumption.
2) The inherently asynchronous memory decouples communication from synchronization and forces programmers to use explicit synchronization primitives such as barriers, which impede concurrency by forcing threads to wait until all others have reached the synchronization point.

More flexible inter-thread communication is offered by the dataflow computing model, which synchronizes computations by coupling direct communication of intermediate values between functional units with the *dataflow firing rule*. Our proposed direct MT-CGRA (dMT-CGRA) architecture extends the *massively multithreaded CGRA* (MT-CGRA) [1], [2] design, which employs the dataflow model, to support inter-thread communication: whenever an instruction in thread A sends a data token to an instruction in thread B, the latter will not execute (i.e., fire) until the data token from thread A has arrived. This simple use of the dataflow firing rule ensures that thread B will wait for thread A. The dMT CGRA model thus avoids costly communication with a memory scratchpad. The existing internal buffers of the CGRA are utilized as a fast, low-power medium for inter-thread communication, thus allowing most communicated values to propagate directly to their destination. Only a small fraction of tokens that cannot be buffered in the fabric are spilled to memory. Moreover, by coupling communication and synchronization using message-passing extensions to SIMT programming models, dMT-CGRA implicitly synchronizes point-to-point data delivery without costly barriers.

The remainder of this section argues for the coupling of communication and synchronization, and discusses why typical programs can be satisfied by the internal CGRA buffering.

A. Dataflow and message passing

We demonstrate our dMT-CGRA message-passing extensions using a *separable convolution* example [6] included in the NVIDIA software development kit (SDK) [7]. This convolution applies a kernel to an image by applying 1D convolutions on each image dimension. For brevity, we focus our discussion on a single 1D convolution with a kernel of size 3. The example is depicted using pseudo-code in Figure 1.

```
thread_code(thread_t tid) {
  // common: not next to margin
  if(!is_margin(tid - 1) && !is_margin(tid + 1) ) {
    result[tid] = globalImage[tid-1] * kernel[0]
                + globalImage[tid]   * kernel[1]
                + globalImage[tid+1] * kernel[2];
  // corner: next to left margin
  } else if(is_margin(tid - 1)) {
    result[tid] = globalImage[tid-1] * kernel[0]
                + globalImage[tid]   * kernel[1];
  // corner: next to right margin
  } else if(is_margin(tid - 1)) {
    result[tid] = globalImage[tid]   * kernel[1];
                + globalImage[tid+1] * kernel[2];
  }
}
```

(a) Spatial convolution using only global memory

```
thread_code() {
  // map the thread to 1D space (CUDA-style)
  tid = threadIdx.x;
  // load image into shared memory
  sharedImage[tid] = globalImage[tid];
  // pad the margins with zeros
  if (is_margin(tid))
    pad_margin(sharedImage, tid);
  // block until all threads finish the load phase
  barrier(): // e.g. CUDA syncthreads
  // execute the convolution; (kernel preloaded in shmem)
  result[tid] = sharedImage[tid-1] * kernel[0]
                  + sharedImage[tid  ] * kernel[1]
                          + sharedImage[tid+1] * kernel[2];
}
```

(b) Spatial convolution on a GPGPU using shared memory

```
thread_code() {
  // map the thread to 1D space (CUDA-style)
  tid = threadIdx.x;
  // load one element from global memory
  mem_elem = globalImage[tid];
  // tag the value of the variable to be sent,
  // in case the variable gets rewritten.
  tagValue<mem_elem>();
  // wait for tokens from threads tid+1 and tid-1
  lt_elem = fromThreadOrConst<mem_elem,/*tid*/-1,0>();
  rt_elem = fromThreadOrConst<mem_elem,/*tid*/+1,0>();
  // execute the convolution
  result[tid] = lt_elem  * kernel[0]
              + mem_elem * kernel[1]
              + rt_em  * kernel[2];
              }              el
```

(c) Spatial convolution on a MT-CGRA using thread cooperation

Fig. 1: Implementation of a *separable convolution* [6]) using various inter-thread data sharing models. For brevity, we focus on 1D convolutions, which are the main and iterative component in the algorithm.

Separable convolution can be implemented using global memory, shared memory, or a message-passing programing model. The trivial parallel implementation, presented in Figure 1a, uses global memory. If the entire kernel falls within the image margins, the matrix elements should be simply multiplied with the corresponding elements of the convolution kernel. If either thread-id (TID) *TID - 1* or *TID + 1* is outside the margins, its matching element should be zero. Although this naive implementation is very easy to code, it results in multiple memory accesses to the image, which are translated to high power consumption and low performance.

GPGPUs use shared memory to overcome this problem, as shown in Figure 1b. Each element of the matrix is loaded once and stored in the shared memory (sharedImage array in the code). The image margins are then padded with zeros. A barrier synchronization must then be used to ensure that all threads finished loading their values. Only after the barrier can the actual convolution be computed. Nevertheless, although the computation phase does not require access to the global memory, the lack of direct inter-thread communication forces redundant accesses the the pre-loaded shared memory, as each image and kernel element is loaded by multiple threads.

A dataflow architecture, on the other hand, can seamlessly incorporate a message-passing framework for inter-thread communication. Figure 1c demonstrates how separable convolution can be implemented in dMT-CGRA. The message-passing primitive allows threads to request the values of other threads' variables. Given the underlying single-instruction multiple-threads (SIMT) model, threads are homogeneous and execute the same code (with diverging control paths). Each thread first loads one matrix element to a register (as opposed to the shared memory write in Figure 1b).

Once the element is loaded, the thread goes on to wait for values read from other threads. The programmer must tag the version of the named variable (in case the variable is rewritten) that should be available to remote threads using the *tagValue* call. Thread(s) can then read the remote value using the *fromThreadOrConst()* call (see Section III-A for the full API), which takes three arguments: the name of the remote variable, the thread ID from which the value should be read, and a default value in case the thread ID is invalid (e.g., a negative thread ID). As in CUDA and OpenCL, thread IDs are mapped to multi-dimensional coordinates (e.g., *threadIdx* in CUDA [8]) and Thread IDs are encoded as constant deltas between the source thread ID and the executing thread ID. Communication calls are therefore translated by the compiler to edges in the code's dataflow graph representing dependencies between instances of the graph (i.e., threads). This process turns the data transfers into the underlying dataflow firing rule (to facilitate compile-time translation, the arguments are passed as C++ template parameters).

The strength of the model lies in this implicit embedding of the communication into the dataflow graph. Values can be forwarded directly between threads by the dMT-CGRA processor, eliminating the need for shared-memory mediation. In addition, the embedding allows threads to move to the computation phase once their respective values are ready, independently of other threads. Since no barriers are required, the implicit dataflow synchronization does not impede parallelism.

```
// IDs in a thread block are mapped to 2D space (e.g., CUDA)
thread_code() {
  // map the thread to 2D space (CUDA-style)
  tx = threadIdx.x;
  ty = threadIdx.y;
  // load A and B into shared memory
  sharedA[tx][ty] = A[tx][ty];
  sharedB[tx][ty] = B[tx][ty];
  // block until all threads finish the load phase
  barrier(): // e.g. CUDA syncthreads
  // compute an element in sharedC (dot product)
  sharedC[tx][ty] = 0;
  for(i=0; i<K; i++)
    sharedC[tx][ty] += sharedA[tx][i]*sharedB[i][ty];
  }
  // write back dot product result to global memory
  C[tx][ty] = sharedC[tx][ty]
}
```

(a) Matrix multiplication on a GPGPU using shared memory.

```
thread_code() {
  // mapping the thread to 2D space (CUDA-style)
  tx = threadIdx.x;
  ty = threadIdx.y;
  //compute memory access predicates
  En_A = (tx == 0);
  En_B = (ty == 0);
  // compute the dot product. the loop is statically
  // unrolled to compute the indices a compile-time
  C[ty][tx] = 0;
#pragma unroll
  for(i=0; i<K; i++) {
    a = fromThreadOrMem<{0, -1}>(A[tx][i], En_A);
    b = fromThreadOrMem<{1,  0}>(B[i][ty], En_B);
    C[ty][tx] += a*b;
  }
}
```

(b) Dense matrix multiplication on the dMT-CGRA architecture using direct inter-thread communication.

Fig. 2: Multiplications of dense matrices $C = A \times B$ using shared memory on a GPGPU and direct inter-thread communication on an MT-CGRA. Matrix dimensions are $(N \times M) = (N \times K) \times (K \times M)$.

B. Forwarding memory values between threads

Multiple concurrent threads often load the same multiple addresses, stressing the memory system with redundant loads. The synergy between a CGRA compute fabric and direct inter-thread communication eliminates this problem by enabling dMT-CGRA to forward values loaded from memory through the CGRA fabric. Figure 2 illustrates this property using matrix multiplication as an example. The figure depicts the implementation of a dense matrix multiplication $C = A \times B$ on a GPGPU and on dMT-CGRA. In both implementations each thread computes one element of the result matrix C.

Figure 2a demonstrates how the classic GPGPU implementation stresses memory. The implementation concurrently copies the data from global memory to shared memory and executes a synchronization barrier (which impedes parallelism), after which each thread computes one element in the result matrix C. Consequently, each element in the source matrices A and B, whose dimensions are $N \times K$ and $K \times M$, respectively, is accessed by all threads that compute a target element in C whose coordinates correspond to either its row or column. As

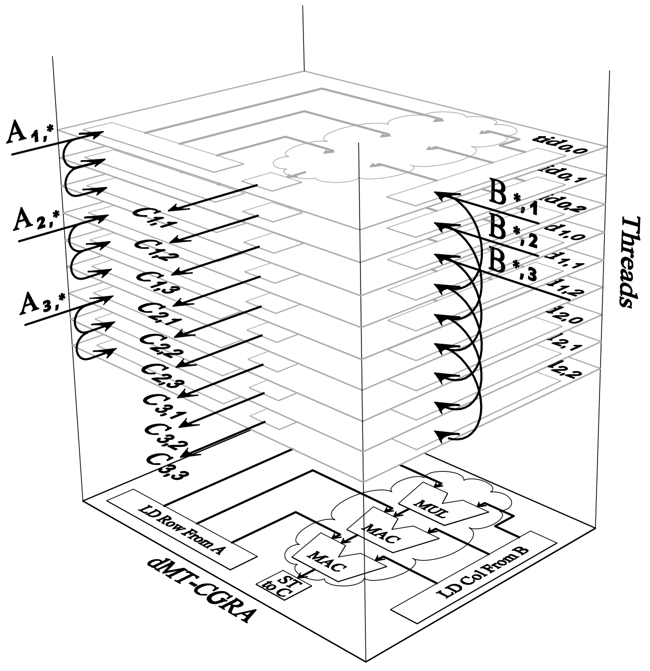


Fig. 3: The flow of data in dMT-CGRA for a 3x3 marix multiplication. The physical CGRA is configured with the dataflow graph (bottom layer), and each functional unit in the CGRA multiplexes operations from different instances (i.e., threads) of the same graph.

a result, each element is loaded by $N \times M$ threads.

We introduce a new memory-or-thread communication primitive to eliminate these redundant memory accesses. The new primitive uses a compile-time predicate that determines whether to load the value from memory or to forward the loaded value from another thread. The dMT-CGRA toolchain maps the operation to special units in the CGRA (described in Section IV) and, using the predicate, configures the CGRA to route the correct value.

Figure 2b depicts an implementation of a dense matrix multiplication using the proposed primitive. Each thread in the example computes one element in the destination matrix C, and the programming model maps each thread to a spatial coordinate (as in CUDA/OpenCL). Rather than a regular memory access, the code uses the *fromThreadOrMem* primitive, which takes two arguments: a predicate, which determines where to get the value from, and a memory address, from which the required value should be loaded. The primitive also uses one parameter, a two-dimensional coordinate that indicates the thread from which the data may be obtained (the coordinates are encoded as the multi-dimensional difference between the source thread and the executing thread's coordinates).

Finally, Figure 3 illustrates the flow of data between threads for a 3×3 matrix multiplication. While the figure shows a copy of the dataflow graph for each thread, we remind the reader that the underlying dMT-CGRA is configured with a single dataflow graph and executes multiple threads by moving their tokens through the graph out-of-order, using dynamic dataflow token-matching. As each thread computes one element in target matrix C, threads that compute the first column load the elements of matrix A from memory, and the threads that compute the first row load the elements of matrix B. As the figure shows, threads that load values from memory forward them to other threads. For example, thread $(0, 2)$ loads the bottom row of matrix A and forwards its values to thread $(1, 2)$, which in turn sends them to thread $(2, 2)$.

The combination of a multithreaded CGRA and direct inter-thread communication thus greatly alleviates the load on the memory system, which plagues massively parallel processors. The following sections elaborate on the design of the programming model, the dMT-CGRA execution model, and the underlying architecture.

III. Execution and programming model

This section describes the dMT-CGRA execution model and the programming model extensions that support direct data movement between threads.

The MT-CGRA execution model: The MT-CGRA execution model combines the *static* and *dynamic* dataflow models to execute single-instruction multiple-thread (SIMT) programs with better performance and power characteristics than von Neumann GPGPUs [1]. The model converts SIMT kernels into dataflow graphs and maps them to the CGRA fabric, where each functional unit multiplexes its operation on tokens from different instances of a dataflow graph (i.e., threads).

An MT-CGRA 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 IV. The interconnect is configured using the program's dataflow graph to statically move tokens between the functional units. Execution of instructions from each graph instance (thread) thus follows the *static dataflow model*. In addition, each functional unit in the CGRA employs *dynamic, tagged-token dataflow* [5], [9] to dynamically schedule different threads' instructions. This prevents memory stalled threads from blocking other threads, thereby maximizing the utilization of the functional units.

Prior to executing a kernel, the functional units and interconnect are configured to execute a dataflow graph that consists of one or more replicas of the kernel's dataflow graph. Replicating the kernel's dataflow graph allows for better utilization of the MT-CGRA grid. The configuration process itself 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/OpenCL coordinates (e.g., threadIdx in CUDA) into the array. 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.

Inter-thread communication on an MT-CGRA: As described above, the MT-CGRA execution model is based on

```
// return the tagged-token for a given tid
<token, tag> = elevator_node(tid) {
  // does the source tid falls within the thread block?
  if(in_block_boundaries(tid - Δ)) {
    // valid source tid? wait for the token.
    token = wait_for_token(tid - Δ);
    return <token, tid>;
  } else {
    // invalid source tid? push the constant value.
    return <C, tid>;
  }
}
```

Fig. 4: The functionality of an elevator node (with a Δ TID shift and a fallback constant C).

dynamic, tagged-token dataflow, where each token is coupled with a tag. The multithreaded model uses TIDs as token tags, which allows each functional unit to match each thread's input tokens. The crux of inter-thread communication is thus reduced to changing a token's tag to a different TID.

We implement the token re-tagging by adding special *elevator* nodes to the CGRA. Like an elevator, which shifts people between floors, the elevator node shifts tokens between TIDs. An elevator node is a single-input, single-output node and is configured with two parameters — a ΔTID and a constant C. The functionality of the node is described as pseudo-code in Figure 4 (and is effectively the implementation of the *fromThreadOrConst* function first described in Figure 1c). For each downstream TID, the node generates a tagged token consisting of the value obtained from the input token for $TID - \Delta$. If $TID - \Delta$ is not a valid TID in the thread block, the downstream token consists of a preconfigured constant C. The elevator node thus communicates tokens between threads whose TIDs differ by Δ, which is extracted at compile-time from either the *fromThreadOrConst* or *fromThreadOrMem* family of functions (Section III-A). These inter-thread communication functions are mapped by the compiler to elevator nodes in the dataflow graph and to their matching counterparts in the CGRA.

Each elevator node includes a small token buffer. This buffer serves as a single-entry output queue for each target TID. The ΔTID that a single elevator node can support is thus limited by the token buffer size. To support $\Delta TIDs$ that are larger than a single node's token buffer, we design the elevator nodes so that they can be cascaded, or chained. Whenever the compiler identifies a ΔTID that is larger than a single elevator node's token buffer, it maps the inter-thread communication operation to a sequence of cascading elevator nodes. In extreme cases where ΔTID is too large even for multiple cascaded nodes, dMT-CGRA falls back to spilling the communicated values to the shared memory. Cascading of elevator nodes is further discussed in Section IV.

Nonetheless, our experimental results show that inter-thread communication patterns exhibit locality across the TID space, and that values are typically communicated between threads with adjacent TID (a Euclidean distance was used for 2D and 3D TID spaces). Figure 5 shows the cumulative distribution

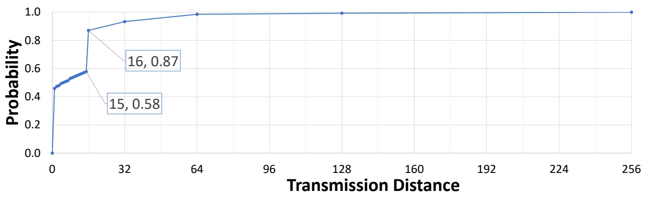


Fig. 5: Cumulative distribution function (CDF) of delta lengths across various benchmarks. 87% of the code we evaluated communicates across ΔTID of 16, indicating strong communication locality.

function (CDF) of the $\Delta TIDs$ exhibited by the benchmarks used in this paper (the benchmarks and methodology are described in Section V-A). The figure shows that the commonly used delta values are small and a token buffer of 16 is enough to support 87% of the benchmarks with no need to cascade *elevator* nodes. However, 42% of the transmission distances are greater than 15. Thus, a GPU *shuffle/permute* operation, which moves data between execution lanes, is not enough. This is because, in a 32-lane GPU SM it will leave 50% of the threads without a producer.

The second functional unit required for inter-thread communication is the enhanced load/store unit (eLSU). The eLSU extends a regular LSU with a predicated bypass, allowing it to return values coming either from memory or from another thread (through an elevator unit). An eLSU coupled with an elevator unit (or multiple thereof) thus implements the *fromThreadOrMem* primitive.

A. Programming model extensions

We enable direct inter-thread communication by extending the CUDA/OpenCL API. The API, listed in Table I, allows threads to communicate with any other thread in a thread block. In this section we describe the three components of the API.

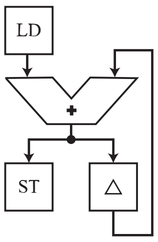
B. Communicating intermediate values

The *fromThreadOrConst* and *tagValue* functions enable threads to communicate intermediate values in a producer-consumer manner. The function is mapped to one or more elevator nodes, which send a tagged token downstream once the sender thread's token is received. This behavior blocks the consumer thread until the producer thread sends the token. The *fromThreadOrConst* function has two variants. The variants share three template parameters: the name of the *variable* to be read from the sending thread, the ΔTID between the communicating threads (which may be multi-dimensional), and a *constant* to be used if the sending TID is invalid or outside the *transmission window*.

The *transmission window* is defined as the span of TIDs that share the communication pattern. The second variant of the *fromThreadOrConst* function allows the programmer to bound the window using the *win* template parameter. We define the *transmission window* as follows: the *fromThreadOrConst* function encodes a monotonic communication pattern between threads, e.g., thread *TID* produces a value to thread TID+Δ,

Function	Description
`token fromThreadOrConst<variable, TIDΔ, constant>()`	Read `variable` from another thread, or `constant` if the thread does not exist.
`token fromThreadOrConst<variable, TIDΔ, constant, win>()`	Same as above, but limit the communication to a window of `win` threads.
`void tagValue<variable>()`	Tag a variable value that will be sent to another thread.
`token fromThreadOrMem<TIDΔ>(address, predicate)`	Load `address` if `predicate` is true, or get the value from another thread.
`token fromThreadOrMem<TIDΔ, win>(address, predicate)`	Same as above, but limit the communication to a window of `win` threads.

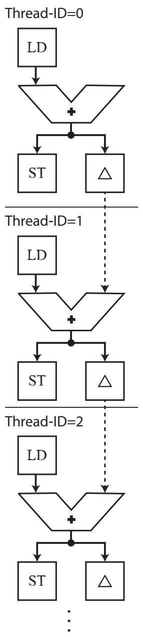
TABLE I: API for inter-thread communications. Static/constant values are passed as template parameters (functions that require ΔTID have versions for 1D, 2D, and 3D TID spaces).



(a) The static dMT-CGRA mapping when executing prefix-sum (scan).

```
thread_code() {
  // mapping the thread to
  // ID space (CUDA-style)
  tid = threadIdx.x;
  //load one value (LD)
  //from global memory
  mem_val = inArray[tid];
  //add the loaded value to
  //the sum so far
  sum =
   fromThreadOrConst<sum,-1,0>()
   + mem_val;
  tagValue<sum>();
  //store partial sum to global memory
  prefixSum[tid] = sum;
}
```

(b) Prefix sum implementation using inter thread communication.



(c) The dynamic execution of prefix sum.

Fig. 6: Example use of the *tagValue* function.

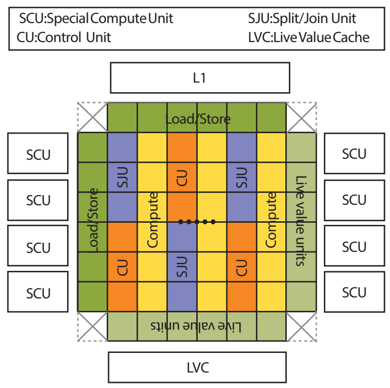
which produces a value to thread *TID*+2×Δ, and so forth. The transmission window is defined as the maximum difference between TIDs that participate in the communication pattern. For a window of size *win*, the thread block will be partitioned into consecutive thread groups of size *win*, e.g., threads $[TID_0 \ldots TID_{win-1}]$, $[TID_{win} \ldots TID_{2\times win-1}]$, and so on. The communication pattern $TID \rightarrow TID + \Delta$ will be confined to each group, such that (for each n) thread $TID_{n\times win-1}$ will not produce a value, and thread $TID_{n\times win}$ will receive the default constant value rather than wait for thread $TID_{n\times win-\Delta}$.
Bounding the transmission window is useful to group threads at the sub-block level. In our benchmarks (Section V-A), for example, we found grouping useful for computing reduction trees. A bounded transmission window enables mapping distinct groups of communicating threads to separate segments at each level of the tree.
The *tagValue* function is used to tag a specific value (or version) of the variable passed to *fromThreadOrConst*. The call to *tagValue* may be placed before or after the call to *fromThreadOrConst*, as shown in the *prefix sum* example depicted in Figure 6 (the example is based on the NVIDIA CUDA SDK [7]). The prefix sum problem takes an array a of values and, for each element i in the array, sums the array values $a[0] \ldots a[i]$. The code in Figure 6b uses the *tagValue* to first compute an element's prefix sum, which depends on the value received from the previous thread, and only then sends the result to the subsequent thread. Figure 6a illustrates the resulting per-thread dataflow graph, and Figure 6c illustrates the inter-thread communication pattern across multiple threads (i.e., graph instances). The resulting pattern demonstrates how decoupling the *tagValue* call from the *fromThreadOrConst* call allows the compiler to schedule the store instruction in parallel with the inter-thread communication, thereby exposing more instruction-level parallelism (ILP).

C. Forwarding memory values

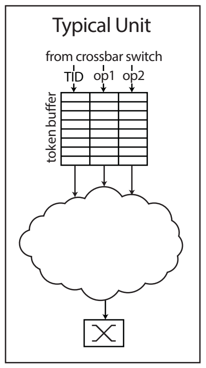
The *fromThreadOrMem* function allows threads that load the same memory address to share a single load operation. The function takes ΔTID as a template parameter, and an *address* and *predicate* as run time evaluated parameters (the function also has a variant that allows the programmer to bound the transmission window). Using the predicate, the function can dynamically determine which threads will issue the actual load instruction, and which threads will *piggyback* on the single load and get the resulting value forwarded to them. A typical use of the *fromThreadOrMem* function is shown in the matrix multiplication example in Figure 2b. In this example, the function allows for only a single thread to load each row and each column in the matrices, and for the remaining threads to receive the loaded value from that thread. In this case, the memory forwarding functionality reduces the number of memory accesses from $N \times K \times M$ to $N \times M$.

D. Programming complexity

Parallel programming is not trivial, and the proposed model is no exception. Nevertheless, the model is unencumbered by microarchitectural constraints and only requires the programmer to understand the parallel algorithm at hand. This is in contrast to inter-thread communication in contemporary GPGPUs, which requires programmers to consider the memory system microarchitecture alongside the parallel algorithm. A full quantitative comparison of the two models is an interesting research trajectory. However, since this paper focuses on the model's performance and energy benefits, a qualitative comparison of the two programming models is out of scope.



(a) A VGIW MT-CGRA core



(b) A typical MT-CGRA unit

Fig. 7: MT-CGRA core overview

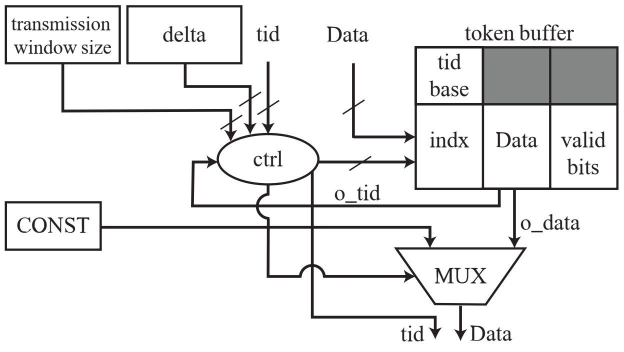


Fig. 8: An elevator node stores the in-flight tokens in the unit's token buffer. A controller manipulates TIDs and controls the value of the output tokens.

IV. The dMT-CGRA Architecture

This section describes the dMT-CGRA architecture, focusing on the extensions to the baseline MT-CGRA [1] required to facilitate inter-thread communication. Figure 7 illustrates the high-level structure of the MT-CGRA architecture.

The MT-CGRA core itself, presented in Figure 7a, is a grid of functional units interconnected by a statically routed network on chip (NoC). The core configuration, the mapping of instructions to functional units, and NoC routing are determined at compile-time and written to the MT-CGRA when the kernel is loaded. During execution tokens are passed between the various functional units according to the static mapping of the NoC. The grid is composed of heterogeneous functional units, and different instructions are mapped to different unit types in the following manner: memory operations are mapped to the load/store units, computational operations are mapped to the floating point units and ALUs (compute units), control operations such as select, bitwise operations and comparisons are mapped to control units (CU), and split and join operations (used to preserve the original intra-thread memory order) are mapped to Split/Join units (SJU).

During the execution of parallel tasks on an MT-CGRA core, many different flows representing different threads reside in the grid simultaneously. Thus, the information is passed as tagged tokens composed from the data itself and the associated TID, which serves as a tag. The tag is used by the grid's nodes to determine which operands belong to which threads.

Figure 7b illustrates the shared structure of the different units. While the functionality of the units differ, they all include tagged-token matching logic to support thread interleaving through dynamic dataflow. Specifically, tagged tokens arrive from the NoC and are inserted into the token buffer. Once all operands for specific TIDs are available, they are passed to the unit's logic (e.g., access memory in LSUs, compute in ALU/FPU). When the unit's logic completes its operation, the result is passed as a tagged token back to the grid through the unit's crossbar switch.

In this paper we introduce two new units to the grid — the *elevator node* and the *enhanced load/store unit* (eLSU). While existing units may manipulate the token itself, they do not modify the tag itself because they must preserve the association between tokens and threads. The two new units facilitate inter-thread communication by modifying the tags of existing tokens.

Figure 8 and Figure 9 depict the elevator node and eLSU nodes, respectively. We introduce these new nodes to the grid by converting the existing control nodes to elevator nodes and LSUs to eLSUs. The conversion only includes adding combinatorial logic to the existing units, since all units in the grid already have an internal opcode register and token buffers. The logic added to the control units to support elevation operations comprises a few registers that store delta constants and a multiplexer. LSUs also require simple control logic. The overall area overhead for these units is below 3%, and the total overhead is below 0.1%. The power consumption is dramatically reduced with comparison to the basic MT-CGRA design, since tokens do not travel via the NoC but are instead rerouted back into the token buffer (see Figure 15 for the fraction of energy spent in the NoC).

A. Elevator node

The elevator node, depicted in Figure 8, implements the *fromThreadOrConst* function, which communicates intermediate values between threads. When mapping *fromThreadOrConst* call, the node is configured with the call's ΔTID and default constant value. An *elevator* node receives tokens tagged with a TID and changes the tag to $TID+\Delta$ according to its preconfigured Δ. It then sends the resulting tagged token downstream.

In the most common case, the node receives an input token from one thread and sends the retagged token to another thread. In this case, threads serve as both data producers, sending a token to a consumer thread, and as consumers, waiting for a token from another producer thread. Alternatively, a thread TID may not serve as a producer if its target thread's ID, $TID + \Delta$, is invalid or outside the current transmission

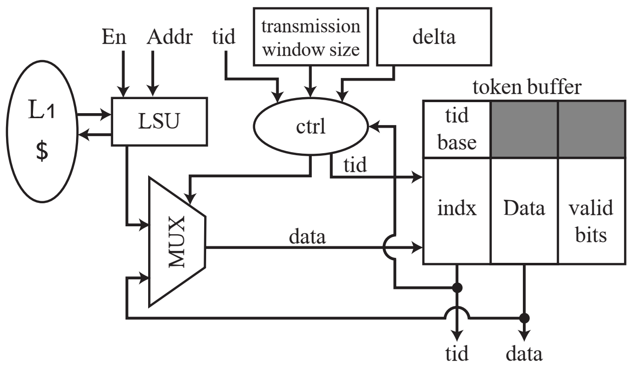
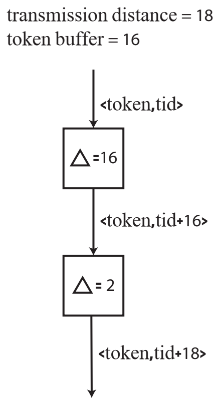
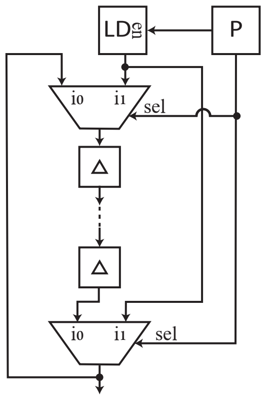


Fig. 9: An eLSU node, consisting of an LSU with an additional adder to manipulate the TID, and a comparator to test whether the result is outside the margins. The *En* input (predicate) determines whether a new value should be introduced. The output is looped back in to create new tokens with higher TIDs with the same data loaded by a previous thread.



(a) Cascading elevator nodes to manage a ΔTID that is larger than the token buffer.



(b) When a *fromThreadOrMem* procedure needs to deal with Δ larger than the token buffer size, the function will be mapped to a cascade of predicated *elevator* nodes in a closed cycle.

Fig. 10: Cascading *elevator* nodes

window. Correspondingly, when the sending thread's ID (e.g., $TID - \Delta$) is outside the transmission window, the elevator node injects the preconfigured constant to the tagged token sent downstream. For threads that both produce and consume tokens, the controller passes the input token to its receiver by modifying the tag from TID to $TID + \Delta$ and pushing the resulting tagged token to the $TID + \Delta$ entry in the token buffer. In addition, the original input TID should be acknowledged by marking the thread as ready in the token buffer. Alternatively, if a thread TID simply needs to receive the predefined constant value as a token, the controller pushes a tagged token comprising the constant and TID to the token buffer. In this case, setting the acknowledged bit does not require an extra write port to the token buffer but only the ability to turn two bits at once.

B. Enhanced load/store unit (eLSU)

The eLSU is used to implement the *fromThreadOrMem* function, which enables threads to reuse memory values loaded by another thread without issuing redundant memory accesses. Figure 9 presents the eLSU, which is a LSU enhanced with control logic that determines whether the token should be brought in from memory or from another thread's slot in the token buffer. The eLSU operates as follows: if the *Enable* (En) input is set, the receiving thread will access the memory system and load the data. Otherwise, if the En is not set, the thread's TID will either be added to the token buffer, where it will wait for another thread to write the token, or the controller will find the token holding the data fetched from memory waiting in the token buffer. In the latter scenario, the thread may continue its flow through the dataflow graph. When the eLSU produces an output token, the token is duplicated and one copy is internally parsed by the node's logic. While the original token is passed on downstream in the MT-CGRA, Δ is added to the TID of the duplicated token. If the resulting TID is equal to or smaller than the *transmission window*, the tagged token will be pushed to the token buffer. Otherwise, the duplicated token will be discarded since its consumer is outside the *transmission window*. Using this scheme, each value is loaded once from memory and reused $\frac{windowsize}{\Delta}$ times, significantly reducing the memory bandwidth.

C. Supporting large transmission distances

The dMT-CGRA architecture uses the token buffers in *elevator* and *eLSU* nodes to implement inter-thread communication. During compilation, the compiler examines the distance between the sending thread and the receiving thread represented as the ΔTID passed to the *fromThreadOrConst* or *fromThreadOrMem* functions. If the distance is smaller or equal to the size of the token buffer, the *fromThreadOrConst* or *fromThreadOrMem* calls will be mapped to a single *elevator* node or *eLSDT* unit, respectively. But if ΔTID is larger than the token buffer, the compiler must cascade multiple nodes to support the large transmission distance.

Long distances in *fromThreadOrConst* calls: In the rare instance that a fromThreadOrConst function needs to communicate values. a *fromThreadOrConst* function needs to communicate values over a transmission distance that is larger than the size of the token buffer, the compiler cascades multiple *elevator* nodes (effectively chaining their token buffers) in order to support the required communication distance.

Figure 10a shows such a scenario. The required transmission distance in the figure is 18, but the token buffer can only hold 16 entries. The compiler handles the longer distance by mapping the operation to two cascaded *elevator* nodes. The compiler further configures the ΔTID of the first node to 16 (the token buffer size) and that of the second one to 2, resulting in the desired cumulative transmission distance of 18.

Parameter	Value
dMT-CGRA Core	140 interconnected compute/LDST/control units
Computational units	32 ALUs, 32 FPUs, 12 Special Compute units
Load/Store units	32 eLDST Units
Control units	16 Split/Join units, 16 Control/Elevator 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 II: dMT-CGRA system configuration.

In the general case of a *transmission window* that is larger than the token buffer size, the number of cascaded units will be $\left\lceil \frac{TID\Delta}{Token\ Buffer\ Size} \right\rceil$. In extreme cases, where the ΔTID is so large that it requires more elevator nodes that are available in the CGRA, the communicated values will be spilled to the *Live Value Cache*, a compiler managed cache used in the MT-CGRA architecture [2]. This approach is similar to the spill fill technique used in GPGPUs.

Long distances in *fromThreadOrMem* procedures: By default, *fromThreadOrMem* calls are mapped to eLSU nodes. Unlike the *elevator* node the eLSU cannot simply be cascaded because it acts as a local buffer for its in-flight memory accesses. For example, in Figure 3 the columns of matrix **B** are loaded by the first three threads and transmitted over a distance of 3 threads ($\Delta TID = 3$). In this case, while the third thread loads its data, the eLSU must be able to hold on the first two loaded values in order to transmit them later. This requires a token buffer of at least 3 entries. A system with a smaller token buffer would require external buffering.

The additional external buffer is constructed by mapping the operation to a loop of cascaded *elevator* nodes. As depicted in Figure 10b, the loop is enclosed by *control* nodes serving as MUXs. To reuse memory values of distant threads the output of the terminating MUX is connected to the input of the first MUX. In this scenario the compiler will map the load instruction to a predicated load-store unit. The predicate passed to the *fromThreadOrMem* will serve as the selector for the MUXs. When the predicate evaluates to false, the original memory value is looped back through the second MUX back to the *elevator* node cascade. A value originating from the TID entering the cascade will be retagged with the target thread ID $TID + \sum_i \Delta_i$. The sum of the *elevator* node ΔTID therefore accounts for the required communication distance. Nevertheless, as shown in Figure 5, the typical ΔTID fits inside the eLSU's token buffer.

V. Evaluation

This section presents our evaluation of the dMT-CGRA architecture. We first discuss the impact of dMT-CGRA on memory bandwidth and code complexity, and then present their implications on overall performance and energy efficiency.

A. Methodology

Simulation framework: We used the GPGPU-Sim simulator [10] and GPUWattch [11] power model (which uses performance monitors to estimate the total execution energy) to evaluate the performance and power of the dMT-CGRA, the MT-CGRA and the GPU architecture. These tools model the NVIDIA GTX480 card, which is based on NVIDIA Fermi. We extended GPGPU-Sim to simulate a MT-CGRA core and a dMT-CGRA core, and we used per-operation energy estimates obtained from RTL place&route results for the new components to extend the power model of GPUWattch to support the MT-CGRA and dMT-CGRA designs.

As a baseline, we compared dMT-CGRA with NVIDIA Fermi and the VGIW architecture [2], an MT-CGRA architecture without direct inter-thread communication. Although Fermi is not the newest NVIDIA architecture, it is the only one with an open, validated power model [11].

The system configuration is shown in Table II. By replacing the Fermi SM with a dMT-CGRA core, we retain the non-core components. For consistency, the amount of logic comprising a dMT-CGRA core is similar to the amount found in an NVIDIA SM and in an SGMF/VGIW MT-CGRA core. The amount of SRAM in both the MT-CGRA and dMT-CGRA is smaller than that used in an NVIDIA SM, since the GPU's RF is replaced with memory structures that consume less than 50% of the SRAM (50KB of token buffers, 64 KB LVC, 8KB CVT).

Compiler: We compiled CUDA kernels using LLVM [12] and extracted their SSA [13] code. This was then used to configure the dMT-CGRA grid and interconnect.

Benchmarks: Of the 21 benchmarks in Rodinia 3.1 [14], two thirds (14 benchmarks) use shared memory, and only two of them require dynamic deltas (<10%). Thus, the scheme proposed in this paper would be beneficial for approximately half of the benchmarks. We evaluated dMT-CGRA using a diverse set of kernels with different characteristics taken from both the Rodinia benchmark suite and the NVIDIA SDK [7].The kernels are listed in Table III. The table also highlights how the different benchmarks benefit from dMT-CGRA, whether due to *memory reuse*, *lower memory bandwidth*, or executing

Application	Description	Memory reuse	Lower mem. BW	Less code
scan	Prefix sum		✓	✓
matrixMul	Matrix multiplication	✓		
conv	Convolution filter		✓	✓
reduce	Parallel Reduction		✓	✓
lud	Matrix decomposition	✓		
srad	Speckle Reducing Anisotropic Diffusion			✓
BPNN	Neural network training	✓	✓	✓
hotspot	Thermal simulation tool		✓	
pathfinder	Find the shortest path on a 2-D grid		✓	

TABLE III: The benchmarks used in this study and how they benefit from the dMT-CGRA architecture. *Memory reuse* marks kernels in which inter-thread communication was used to eliminate redundant memory accesses; *Lower Mem. BW* marks kernels that benefit from direct inter-thread communication rather than indirect, shared-memory based communication; and *Fewer insts.* marks kernels whose code was simplified (i.e., executed fewer instructions) by using direct inter-thread communication primitives.

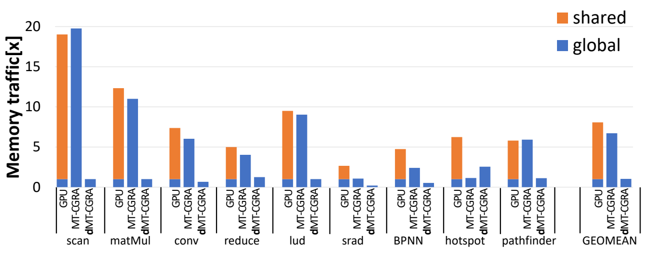


Fig. 11: Number of words read/written from/to memory (shared and global) normalized to the number of words communicated to/from the **global memory** in the Fermi baseline

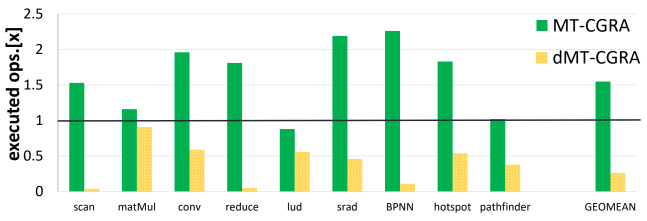


Fig. 12: A comparison of the number of dynamic operations executed on dMT-CGRA and the baseline architectures.

less code. All evaluated kernels use shared memory and can be expressed with static deltas.

B. Reduced memory bandwidth

Figure 11 depicts the amount of data read/written from the global and shared memory. As expected, the shared memory serves most of the bandwidth on NVIDIA GPGPUs. Furthermore, in VGIW, the shared memory address space is mapped to global memory, to which all the data bandwidth is directed. In contrast, dMT-CGRA and its programming model extensions confine almost all data transfers inside the execution fabric and dramatically reduce the costly transfers of data to shared and global memory.
The figure thus demonstrates how dMT-CGRA can dramatically reduce memory bandwidth by eliminating the shared memory as an intermediary for inter-thread communication (e.g., scan, convolution). In addition, direct inter-thread communication allows threads to reuse memory values already loaded by others (e.g., matrixMul, lud). We will discuss the impact of the reduced memory bandwidth on overall performance and energy consumption later in this section.

C. Reduced operations count

Direct inter-thread communication eliminates redundant address calculation needed for shared-memory based communications, redundant boundary checks, and shared-memory based reduction operations. Figure 12 compares the number of dynamic operations executed on each architecture (with Fermi as a baseline) and shows that for most of the benchmarks this number is dramatically reduced. This is most pronounced in *scan*, where direct inter-thread communication eliminates a substantial fraction of the code that performs data reduction. In contrast, for matrixMul, which only benefits from memory reuse, the reduction is negligible since it simply replaces memory load instructions with direct communication primitives. Overall, the figure shows that dMT-CGRA reduces the number of executed operations by $\sim$75% on average compared to a Fermi GPGPU.

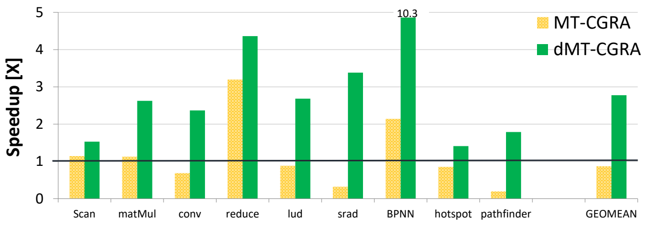


Fig. 13: Speedups obtained with dMT-CGRA compared to the baseline architectures.

D. Performance analysis

The performance of any processor depends on the number of instructions it executes and on the utilization of its functional units. As shown above, dMT-CGRA reduces number of dynamic operations executed by each benchmark. Furthermore, memory reuse across threads (e.g., *matrixMul*) reduces long memory latencies that may affect utilization. Finally, spatial architectures are not bound by instruction fetch width and register file bandwidth and can thereby operate all the functional units on the grid. For example, a spatial architecture composed of 140 functional units can theoretically deliver $\frac{140}{32} \approx 4.4\times$ higher IPC than an equivalent 32-wide GPGPU .
Figure 13 demonstrates the performance speedups obtained by dMT-CGRA over the baseline architectures. For the most part, performance correlates with thereduction in the number of operations shown in Figure 12. In addition, benchmarks that exhibit memory reuse (e.g., *matrixMul*) benefit from increased performance because of the reduction in global memory bandwidth. Conversely, benchmarks that exhibit low ILP/TLP (e.g., *scan*) have little performance benefit despite the reduction in the number of operations. Overall, Figure 13 shows that dMT-CGRA outperforms NVIDIA Fermi by $2.8\times$ on average (up to $10.3\times$).
The heterogeneous composition of the MT-CGRA functional units makes it difficult to achive full utilization, since different workloads have different instruction mixes. For example, while the *pathfinder* kernel utilizes 100% of the available ALU units, it does not perform any floating point computations leaving the FP unit idle. Although we fixed the composition of functional units to match that of NVIDIA Fermi for a fair comparison, customizing the functional unit composition in dMT-CGRA can potentially deliver better performance gains.

E. Energy efficiency analysis

We now compare the energy efficiency of the evaluated architectures. Since the different architectures use different *instruction set architectures* (ISAs), we define energy efficiency as the total energy required to execute the benchmark.

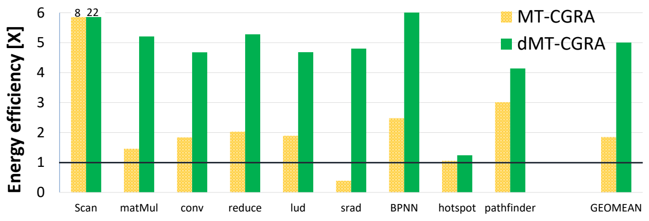


Fig. 14: Energy efficiency of a dMT-CGRA core over a VGIW MT-CGRA core and Fermi SM.

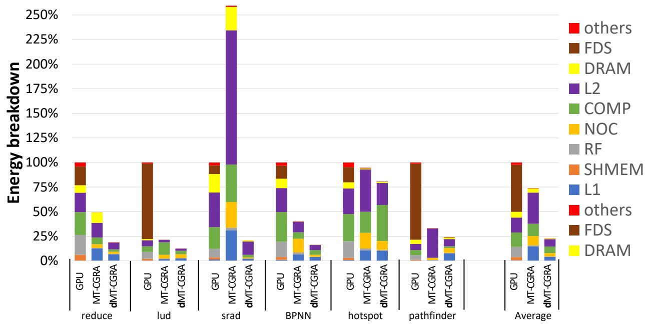
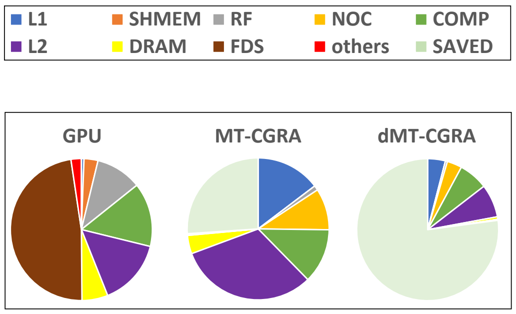


Fig. 15: Energy consumption across core components.

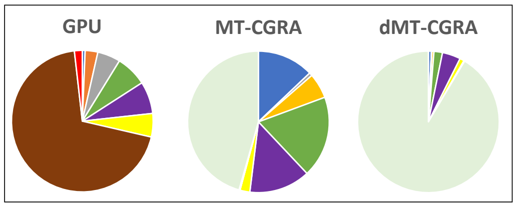
Figure 14 shows the overall energy efficiency of dMT-CGRA and MT-CGRA compared to Fermi. The figure demonstrates the efficiency of dMT-CGRA, which is on average 5× more energy efficient than Fermi (2.8× for MT-CGRA). The best energy reduction is obtained for the *scan* kernel implementation using inter-thread communication. Even though this benchmark does not benefit from a major performance improvement due to low ILP/TLP, its energy efficiency is improved by almost 22× thanks to the reduction in memory access overhead and in the number of operations (attributed to the elimination of the complex tree reduction algorithm shown in Figure 6).

We now examine the average energy consumption of the core components. The energy breakdown for all benchmarks on the three architectures is shown in Figure 15, but for brevity we focus on three representative benchmarks in Figure 16. Specifically, Figure 16a shows the energy breakdown for *matrixMul* (Figure 2), which benefits from data reuse; Figure 16b shows the breakdown for *convolution* (Figure 1), which benefits from a reduction in shared memory bandwidth and in the number of operations; and Figure 16c shows the breakdown for *scan* (Figure 6), which primarily benefits from the reduction in the number of operations. The pie charts are normalized to the energy consumed by Fermi for each task. The reduced energy is marked as the *SAVED* portion in each pie chart.

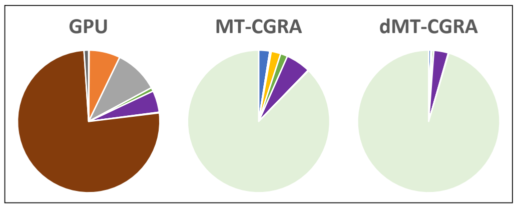
When executed on Fermi all benchmarks waste much of the energy on von Neumann artifacts, namely the pipeline (fetch-decode-schedule, or FDS), the shared memory (SHMEM), the register file (RF), and the data movement among them (included in FDS). As a result, the inefficient Fermi spends only about 14% of its energy on the functional units themselves (COMP). The MT-CGRA architecture variants, on the



(a) Matrix multiplication energy breakdown



(b) Convolution energy breakdown



(c) Scan (prefix-sum) energy breakdown

Fig. 16: The energy breakdown of three representative benchmarks

other hand, eliminate most of the these von-Neumann artifacts. However, the regular MT-CGRA's reliance on memory as a communication medium forces it to spend a lot of energy (about 80% of the total for *matrixMul*, *convolution* and *Prefix-sum*) on its energy on the memory system (L1,L2 and DRAM). Furthermore, the excess memory traffic causes additional traffic on the NoC and increases the NoC's energy consumption.

Finally, dMT-CGRA is shown to be much more energy efficient than its competitors. The direct inter-thread communication primitives eliminate most of the energy consumed by excessive data transfers to memory (L1, L2, DRAM, and NOC) and also eliminate the von Neumann artifacts. Furthermore, we see that the computational units also consume less energy thanks to the reduction in the number of operations (Section V-C). Ultimately, dMT-CGRA incurs the lower overhead for management and control of the computation than do Fermi and MT-CGRA, as its functional units consume an average of 32% of its total energy.

To conclude, our evaluation demonstrates the performance and power benefits of the dMT-CGRA architecture over a von Neumann GPGPU (NVIDIA Fermi) and an MT-CGRA without the support of inter-thread communication.

VI. Related Work

Dataflow architectures and CGRAs:

There is a rich body of work on the potential of dataflow based engines in general, and CGRAs in particular. DySER [15], SEED [16], and MAD [17] extend von-Neumann based processors with dataflow engines that efficiently execute code blocks in a dataflow manner. Garp [18] adds a CGRA component to a simple core in order to accelerate loops. TRIPS [19], WaveScalar [20] and Tartan [21] portion the code into hyperblocks, which are scheduled according to the dependencies between them. Stream-dataflow [22] and R-GPU [23] are static dataflow architectures which pipeline instances of simple tasks through a reconfigurable execution engine. These architectures mainly leverage their execution model to accelerate single-threaded performance. However, TRIPS and WaveCache [21] enable multi-threading. TRIPS by scheduling different threads to different tiles on the grid and WaveCache by pipelining instances of hyperblocks originating from different threads. Nevertheless, none of the mentioned architectures supports simultaneous dynamic dataflow execution of threads on the same grid. While SGMF [1] and VGIW [2] do support simultaneous dynamic multithreaded execution on the same grid, they do not support inter-thread communication.

Message passing and inter-core communication:

Support of inter-thread communication is vital when implementing efficient parallel software and algorithms. The MPI [24] programing model is perhaps the most scalable and popular message passing programing model. Many studies implemented hardware support for fine-grain communication across cores. The MIT Alewife machine [25], MIT Raw [26], ADM [27], CAF [28] and the HELIX-RC architecture [29] add an integrated hardware to multi-core systems, in order to provide fast communication and synchronization between the cores, whereas XLOOPS [30] provides hardware mechanisms to transfer loop-carried dependencies across cores. These prior works have explored hardware assisted techniques to support communication between cores. In this paper we applied the same principles in a massively multithreaded environment and implemented communication between threads.

Inter-thread communication:

To enable decoupled software pipelining in sequential algorithms, DWSP [31] adds a synchronization buffer to support value communication between threads. The NVIDIA, AMD and HSA ISAs offer support for inter-thread communication within a wavefront (warp) using *shuffle/permute* instructions, as described in the relevant programing guides [8], [32], [33]. However, this form of communication is limited to data transfers within a wavefront and cannot be used to synchronize between threads since all threads within a wavefront execute in lockstep. Nevertheless, the addition of such instructions in the SIMT programing models, even if limited in scope, demonstrates the need for inter-thread communication in GPGPUs.

VII. Conclusions

Redundant memory accesses are a major bane for throughput processors. Such accesses can be attributed to two major causes: using the memory for inter-thread communication, and having multiple threads access the same memory address.

In this paper we introduce direct inter-thread communication to the previously proposed *multithreaded coarse-grain reconfigurable array* (MT-CGRA) [1], [2]. The proposed *dMT-CGRA* architecture eliminates redundant memory accesses by allowing threads to directly communicate through the CGRA fabric. The direct inter-thread communication eliminates the use of memory as a communication medium and allows threads to directly forward shared memory values rather than invoke redundant memory loads.

dMT-CGRA obtains average speedups of 3.2× and 2.8× over MT-CGRA and NVIDIA GPGPUs, respectively. At the same time, dMT-CGRA reduces energy consumption by an average of 63% compared to MT-CGRA and 80% compared to NVIDIA GPGPUs.

Acknowledgment

We thank the anonymous reviewers for their comments and suggestions. We thank Reut Zigdon Voitsechov for helping with the graphics. We also thank Oren Kalinsky, Lluis Vilanova, Nimrod Wald and Liraz Blumenkrantz Mencell for their valuable feedback and help throughout this research.

References

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

[2] D. Voitsechov and Y. Etsion, "Control flow coalescing on a hybrid dataflow/von Neumann GPGPU," in *Intl. Symp. on Microarchitecture (MICRO)*, 2015.

[3] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: determinacy, termination, queueing," *SIAM J. Applied Mathematics*, vol. 14, Nov 1966.

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

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

[6] V. Podlozhnyuk, "Image convolution with CUDA," NVIDIA, Tech. Rep., Jun 2007.

[7] NVIDIA, "CUDA SDK code samples." [Online]. Available: http://developer.download.nvidia.com/compute/cuda/ sdk/website/samples.html

[8] *CUDA Programming Guide v7.0*, NVIDIA, Mar 2015.

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

[10] 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.

[11] 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.

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

[13] 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, 1991.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, “Stream-dataflow acceleration,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[23] G.-J. V. D. Braak and H. Corporaal, “R-gpu: A reconfigurable gpu architecture,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, 2016.

[24] Message Passing Interface Forum, “MPI: A message-passing interface standard,” Jun 2015, version 3.1.

[25] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, “The MIT alewife machine: Architecture and performance,” in *Intl. Symp. on Computer Architecture (ISCA)*, 1995.

[26] 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, 2002.

[27] D. Sanchez, R. M. Yoo, and C. Kozyrakis, “Flexible architectural support for fine-grain scheduling,” in *Intl. Conf. on Arch. Support for Prog. Lang. & Operating Systems (ASPLOS)*, 2010.

[28] Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin, “CAF: Core to core communication acceleration framework,” in *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, 2016.

[29] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, “HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs,” in *Intl. Symp. on Computer Architecture (ISCA)*, 2014.

[30] 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.

[31] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, “Decoupled software pipelining with the synchronization array,” in *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, 2004.

[32] AMD, “Vega instruction set architecture, reference guide,” 2017. [Online]. Available: https://gpuopen.com/amd-vega-instruction-set-architecture-documentation/

[33] HSA, “HSA programmer’s reference manual: HSAIL virtual ISA and programming model, compiler writer, and object format (BRIG),” 2017. [Online]. Available: http://www.hsafoundation.com/standards/