Decoupled Affine Computation for SIMT GPUs

Kai Wang Calvin Lin
Department of Computer Science
The University of Texas at Austin
Austin, Texas 78712, USA
{kaiwang,lin}@cs.utexas.edu

ABSTRACT

This paper introduces a method of decoupling *affine computations*—a class of expressions that produces extremely regular values across SIMT threads—from the main execution stream, so that the affine computations can be performed with greater efficiency and with greater independence from the main execution stream. This decoupling has two benefits: (1) For compute-bound programs, it significantly reduces the dynamic warp instruction count; (2) for memory-bound workloads, it significantly reduces memory latency, since it acts as a non-speculative prefetcher for the data specified by the many memory address calculations that are affine computations.

We evaluate our solution, known as Decoupled Affine Computation (DAC), using GPGPU-sim and a set of 29 GPGPU programs. We find that on average, DAC improves performance by 40% and reduces energy consumption by 20%. For the 11 compute-bound benchmarks, DAC improves performance by 34%, compared with 11% for the previous state-of-the-art. For the 18 memory-bound programs, DAC improves performance by an average of 44%, compared with 16% for state-of-the-art GPU prefetcher.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**;

ACM Reference format:
Kai Wang Calvin Lin Department of Computer Science The University of Texas at Austin Austin, Texas 78712, USA . 2017. Decoupled Affine Computation for SIMT GPUs. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/http://dx.doi.org/10.1145/3079856.3080205

Keywords: GPU, decoupling, affine computation

1 INTRODUCTION

GPUs are optimized for regular data parallel computations, for which they provide significant power and performance benefits over CPUs. Much of their benefit comes from their vector model, which allows GPUs to coalesce control flow and memory accesses to amortize their overhead across multiple data elements. To provide programming convenience, the SIMT model is used by many GPUs to transform vector computation into data parallel threads (SIMT threads). However, the SIMT model introduces inefficiencies for scalar computations, which must be redundantly computed on every thread [17], so previous work [9, 25, 26] proposes specialized hardware support for scalar computations.

This specialized support for scalar computations can be generalized to the notion of *affine computations* [6], which are linear combinations of scalars and thread IDs, and which can be executed efficiently by exploiting their high degree of regularity across threads. Affine computations are common because GPU workloads use thread IDs to map work to SIMT lanes, so many memory address calculations and many predicate computations are expressed in terms of these thread IDs.

	Baseline GPU					Affine Computation
Operands	SIMT Lanes					Affine Tuple
	0	1	2	...	31	(base, offset)
A	0x100	0x104	0x108	...	0x17C	(0x100, 4)
B	0x200	0x200	0x200	...	0x200	(0x200, 0)
C=A+B	0x300	0x304	0x308	...	0x37C	(0x300, 4)

Figure 1: Operand Values–Baseline GPU and Affine Computation

Figure 1 shows how affine computations can be computed much more efficiently than their direct SIMT counterparts. First, we see that affine computations can be compactly represented as **affine tuples:** The value of `A` starts at `0x100` in thread 0 and then increases by 4 with each successive thread, so the entire `A` vector can be represented as a tuple `(0x100, 4)`, where `0x100` is a base and `4` is the implied offset per thread. Similarly, the scalar `B` can be represented as the tuple `(0x200,0)`. Next, we see that to compute the value of `C`, we need just two additions—one to add `A`'s base to `B`'s base and another to add `A`'s offset to `B`'s offset—producing `(0x300, 4)`, whereas a standard computation would require one addition for each SIMT lane. `C`'s affine tuple can then be used as a source operand for subsequent affine computations. Of course, at some point, such as when accessing memory, the affine tuple must be **expanded** to the different concrete values, such as those that represent cache-line addresses (see Section 4.2).

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06. . . $15.00
https://doi.org/http://dx.doi.org/10.1145/3079856.3080205

Previous support for affine computation [13] adds an affine functional unit to each Streaming Multiprocessor (SM) of a GPU, but such an approach only removes redundancy within a single warp and does not reduce the dynamic warp instruction count, so its performance and energy efficiency benefits are limited.

In this paper, we show that there are two significant advantages to decoupling the execution of **affine instructions** (i.e., instructions that are eligible for affine computation) from the execution of non-affine instructions. First, unlike previous work [13], the decoupling of affine instructions allows a single affine tuple to eliminate redundancy across multiple warps (see Figure 2), so, for example, if an SM executes 48 warps concurrently, there is an additional 48× redundancy to remove for each affine computation. As a result, our solution decreases the dynamic warp instruction count, which improves both program performance and energy efficiency (see Section 4). Second, for memory-bound workloads, this decoupling significantly reduces memory latency, allowing the address calculations to bypass stalled instructions on a GPU's in-order cores, thereby providing a form of non-speculative data prefetching.

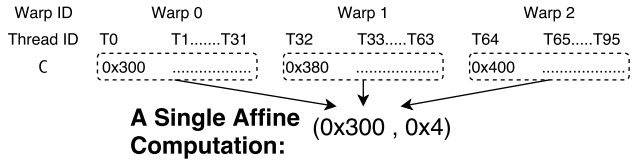


Figure 2: A Single Affine Tuple Applies to Multiple Warps—the value of operand C from Figure 1

.

As a redundancy reduction technique, our solution, which we refer to as Decoupled Affine Computation (DAC), uses decoupling to enable affine instructions to be executed just once on an **affine warp**, while non-affine instructions execute as usual on separate non-affine warps. Figure 3 illustrates the advantages using an example with one affine instruction and one non-affine instruction executed on four warps. The figure shows that on a baseline GPU (left), the affine instruction is computed using standard SIMT lanes. With previous affine computation techniques [13] (center), the affine instruction is executed more efficiently on scalar functional units, but the affine instruction is still executed redundantly across multiple warps. With DAC (right), a compiler separates the code into two streams, with the affine instruction executing on a separate affine warp that is executed just once.

As a memory latency hiding technique, our solution is similar in spirit to the idea of Decoupled Access/Execute (DAE) architectures [23], which decouple a program into a memory access stream and an execution stream, but there are significant differences. First, a direct adaptation of DAE would be quite expensive on SIMT GPUs, since it would double the number of warps in a program execution. DAC instead allows one affine warp to service a large number of non-affine warps. Second, rather than decouple all memory

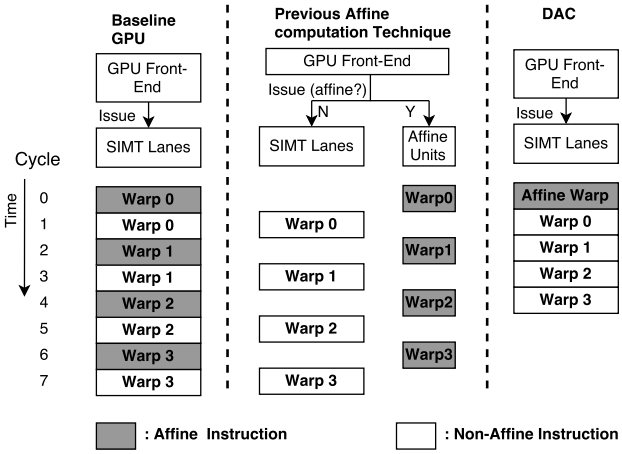


Figure 3: Instruction Issue Trace of a Baseline GPU, Previous Affine Computation Technique, and DAC.

access instructions, DAC decouples affine computations for memory address computations and predicate computations, which exhibit greater independence from the execution stream.

This paper makes the following contributions:

- We introduce the notion of Decoupled Affine Computation, which decouples GPU kernel execution into affine warps and non-affine warps to reduce the dynamic warp instruction count and to hide memory latency.
- We introduce a mechanism for decoupling affine computations in the face of control flow divergence, which further increases DAC's coverage of affine instructions in SIMT workloads.
- We implement our solution in a version of GPGPU-sim 3.2.2 that has been modified to better model the memory system. For a set of 29 GPGPU benchmarks, DAC achieves a 40.7% geometric mean speedup and a 20.2% reduction in total energy consumption (18.4% reduction in dynamic energy) when compared to a baseline GTX 480 GPU. Those improvements are achieved by reducing the instruction count by 26.0% and by decoupling 79.8% of global and local load requests.

  When compared against a generously provisioned state-of-the-art GPU prefetcher (MTA) [15] on the 18 memory-bound programs, DAC achieves a 44.7% mean speedup compared to MTA's 16.7% speedup.

  When compared against previous support for affine computation (CAE) [13] (again generously provisioned) on the 11 compute-bound benchmarks, DAC achieves a 34.0% mean speedup, compared to CAE's 11.0%.

The remainder of this paper follows a standard organization. Section 2 describes Related Work, and Section 3 provides background material that makes the paper more accessible. We then describe our solution in Section 4 and our empirical evaluation in Section 5, before concluding in Section 6.

2 RELATED WORK

We now describe relevant prior work in the areas of affine computation, GPU prefetching, and Decoupled Access Execution.

Scalar and Affine Computation. Previous work [9, 25, 26] proposes a dedicated data path for scalar computation to eliminate redundancy and to improve performance and energy efficiency. Some GPUs [1] also include a scalar data path alongside the vector data path.

Our solution extends the special support for affine computation by decoupling its execution onto a separate warp, which (1) reduces the dynamic warp instruction count and (2) reduces memory latency.

Lee, et al [3] present a compiler-based technique to identify opportunities for scalar code to execute under divergent constraints in GPU workloads. Collange, et al [6] present a scalarizing compiler technique for mapping CUDA kernel to SIMD architectures. We build on their insights and present a compiler technique for identifying control-flow divergent conditions.

Memory Latency Hiding in GPUs. Another line of work [11, 12, 15, 22, 27] builds on the regularity of memory accesses across different GPU threads to infer prefetches based on the observed behavior of a few threads. Unfortunately, GPU prefetchers can sometimes be vexed by useless prefetches for inactive threads, which can cause cache pollution and other contention [15]. By contrast, our solution issues early memory requests non-speculatively as a part of the program execution, and it does not suffer from mispredictions or early evictions.

Kim et al. [14] present a technique that allows warps to continue issuing non-dependent instructions without waiting for long-latency instructions to complete.

Decoupled Access Execution. Decoupled Access Execution (DAE) [7, 8, 16, 23] is a lightweight memory latency hiding mechanism for in-order processors. The main idea is to decouple memory instructions (the access stream) from other instructions (the execute stream) so that the access stream can bypass memory stalls and issue memory requests early. Arnau et al. [2] decouple memory accesses from a fragment processor's tile queue, allowing a tile's memory requests to be issued before dispatch. DAC employs decoupling to affine computations both to reduce memory latency and to improve computational efficiency.

3 BACKGROUND AND MOTIVATION

This section provides more details about affine computation and quantifies the potential number of affine instructions in SIMT workloads.

SIMT kernels often use scalar data, such as kernel parameters (e.g. `num`, `A[]`) and the thread ID to map memory accesses and control flow to threads. For example, Figure 4 shows a sample CUDA kernel, and Figure 5 shows the affine tuples that can be used by this code.

```
void example_kernel(int
     A[],int B[],int
    dim,int num)
{
  int tid=blockIdx.x*
      blockDim.x+
      threadIdx.x;
  for(int i=0;i<dim;i
      ++)
  {
    int tmp=A[i*num+tid
        ];
    B[i*num+tid]=tmp+1;
  }
}
```

(a) CUDA Code

```
 1 mul r0, blockIdx.x,
       blockDim.x;
 2 add tid, threadIdx.x,
       r0;
 3 mul r1, tid, 4;
 4 add addrA, A[], r1;
 5 add addrB, B[], r1;
 6 mov i, 0;
 7 LOOP:
 8 ld.global tmp, [addrA];
 9 add r2, tmp, 1;
10 st.global [addrB], r2;
11 add i, i, 1;
12 mul r3, num, 4;
13 add addrA, r3, addrA;
14 add addrB, r3, addrB;
15 setp.ne p0, dim, i;
16 @p0 bra LOOP;
```

(b) Pseudo Assembly Code

Figure 4: Example Kernel

	Operand Value			Affine Tuple
operand name	Thread 0	Thread 1	Thread 2	(Base, Offset)
A[]	0x80000	0x80000	0x80000	(0x80000,0x0)
#3, r1	0x0	0x4	0x8	(0x0,0x4)
#4, addrA	0x80000	0x80004	0x80008	(0x80000,0x4)
#12, r3	0x1000	0x1000	0x1000	(0x1000,0x0)
#13, addrA	0x81000	0x81004	0x81008	(0x81000,0x4)

Figure 5: Affine Values and Affine Tuples for Three Threads

An affine tuple represents values as a function of the thread ID:

$$operand\_value \;\; base\; thread\_ID \times offset \tag{1}$$

Here, base corresponds to scalar data and offset is the constant difference between adjacent threads. Since base and offset have the same value for all threads, the affine tuple, (base, offset), represents all of the thread's values with just two registers.

Affine computation is performed directly on affine tuples. Affine addition adds a base to a base and an offset to an offset, e.g. `add addrA, A[], r1;`

$$b1, o2 \; b2, o2 \;\; b1 \; b2, o1 \; o2 \tag{2}$$

Multiplication of two affine operands is not allowed, but the multiplication of a scalar and an affine operand can be performed by multiplying the base and offset by the scalar value, e.g. `mul r1, tid, 4;`.

$$b1, 0 \times b2, o2 \;\; b1 \times b2, b1 \times o2 \tag{3}$$

Other similar ALU operations (e.g. sub, shl, mad, etc.) are supported, and these simple operations constitute a large portion of computations on scalar data and Thread IDs, as they are frequently used for address and predicate bit vector computations.

A sequence of affine computations can continue as long as both source and destination operands can be represented as affine tuples. Otherwise, affine tuples must be **expanded** into concrete values. For memory instructions with affine addresses (e.g. `addrA`) and for predicate computation instructions with

affine operands (e.g. #15), expansion can be handled efficiently in most cases. For example, `addrA`, has an offset of 4, and 32 consecutive threads of a warp can be serviced by a single cache line address; thus a warp can be expanded by a single ALU operation. We describe efficient address and predicate bit vector expansion mechanisms in Sections 4.2 and 4.3. If an affine tuple cannot be expanded into predicate bit vectors or addresses, then it must be expanded into concrete vector values by evaluating function (1) explicitly for each thread.

In Figure 4, `addrA`, `addrB`, and `p0` are computed entirely from scalar data and thread IDs. Although the example is trivial, such program patterns are common in SIMT workloads.

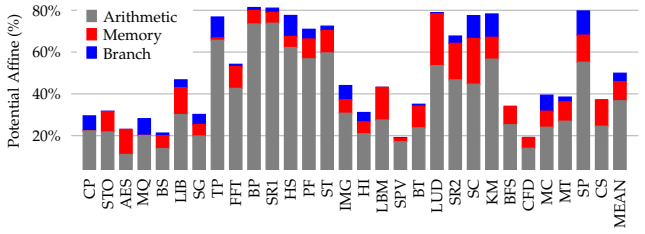


Figure 6: Percentage of Instructions Computing on Scalar Data and Thread IDs

Figure 6 shows that for our 29 benchmarks, about half of the static instructions are potentially affine instructions. These are "potentially affine instructions" because two factors—control flow divergence and instruction type—can force them to execute in non-affine warps. Previous affine computation techniques [6, 13] cannot execute affine computation after control flow divergence, but our solution uses compile-time analysis and runtime mechanisms to execute affine instructions after limited forms of divergence. In addition, as described in Sections 4.4 and 4.6, our solution provides support for additional instruction types (e.g. mod, min, max, etc), which cannot be handled by previous affine computation solutions.

4 OUR SOLUTION

Our solution targets regular SIMT workloads, where scalar data and thread IDs are commonly used for address and predicate computation instructions.

We now present our solution, first describing the basic idea and then walking through a code example to show how the original code is decoupled, is executed, and allows memory latency to be hidden. We then describe the overall hardware design before describing each component in more detail.

The Basic Idea. For affine computations, the fundamental source of redundancy is the fact that each warp executes the same kernel code. For vector computations, this replication is not an issue, because the same instruction operates on different data on the different warps. For affine instructions, this replication translates to redundancy.

To solve this problem, DAC decouples affine and non-affine instructions into separate instruction streams and executes them on different warps. For concurrent warps of an SM, a single affine warp fetches and executes only affine instructions, while non-affine warps fetch and execute only non-affine instructions. DAC is thus able to use a single affine warp for the affine instructions, while still launching as many warps as needed for the non-affine instructions.

The kernel code is decoupled by a static compiler, while specialized hardware is added to support for the affine stream at run time.

```
1 LOOP:
2 mul r0, blockIdx.x,
      blockDim.x;
3 add tid, threadIdx.x,
      r0;
4 mul r1, tid, 4;
5 add addrA, A[], r1;
6 add addrB, B[], r1;
7 mov i, 0;
8 LOOP:
9 enq.data addrA;
10 enq.addr addrB;
11 add i, i, 1;
12 mul r3, num, 4;
13 add addrA, r3, addrA;
14 add addrB, r3, addrB;
15 setp.ne p0, dim, i;
16 enq.pred p0
17 @pred bra LOOP;
```

(a) The Affine Instruction Stream

```
1 LOOP:
2 ld.global tmp, deq.data
      ;
3 add r2, tmp, 1;
4 st.global [deq.addr],
      r2;
5 @ deq.pred bra LOOP;
```

(b) The Non-Affine Instruction Stream

Figure 7: Decoupling the Kernel in Figure 4b

Code Example. Figure 7 shows that the original code from Figure 4 is compiled into two instruction streams.

We see that memory accesses are decoupled into two parts: The affine warp uses affine tuples to compute the memory addresses and then sends the affine tuples to the non-affine warps by Enqueueing them to the address queue. The non-affine warps then Dequeues the concrete values. For example, the Store instruction on line 10 of the original code (`st.global[addrB], r2;`) is translated into line 10 in the affine instruction stream (`enq.addr addrB;`) and line 4 in the non-affine stream (`st.global [deq.addr], r2;`). Predicate computation instructions are handled in a similar manner.

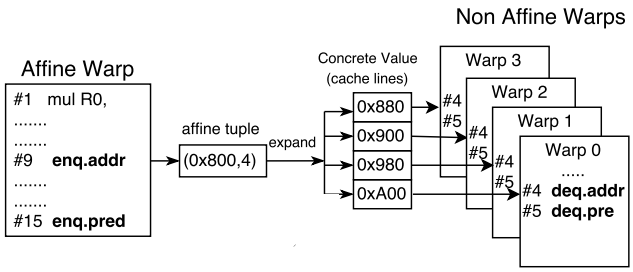


Figure 8: Interaction Between the Affine Warp and the Non-Affine Warps

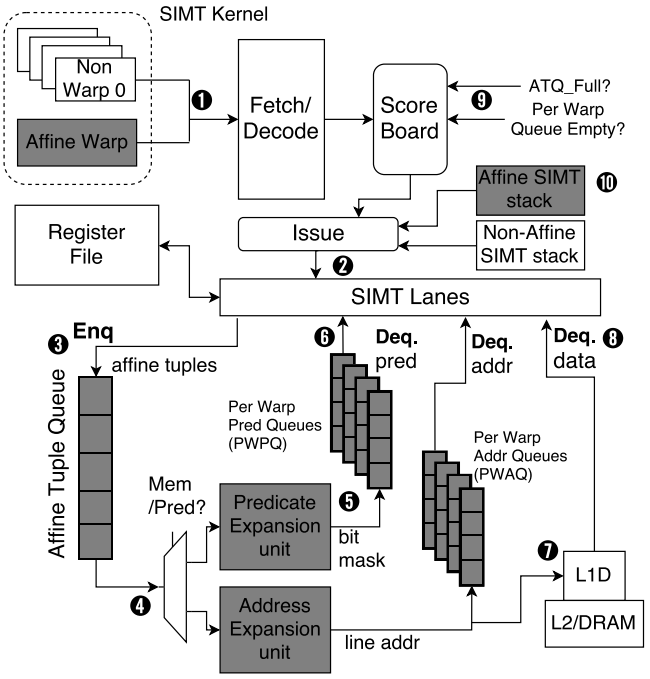


Figure 9: DAC Hardware Organization

The Enqueue and Dequeue instructions trigger hardware mechanisms that (1) expand affine tuples into concrete values and (2) coordinate the the two streams at run time. Figure 8 shows the interaction between the two instruction streams in hardware. The single affine warp sends a tuple for expansion when executing an Enqueue instruction. An affine tuple is expanded into concrete values (cache line addresses or predicate bit vectors) and buffered for each non-affine warp. Non-affine warps then retrieve the concrete values from buffer when executing Dequeue instructions.

DAC only decouples instructions that compute memory addresses and predicate bit-vectors, since their end products (i.e. addresses and bit-vectors) can be efficiently expanded in most cases (Section 3).

To understand why the affine warp can run ahead of the non-affine warps to hide memory latency, observe that the affine warp operates on read-only data, such as thread IDs and kernel parameters, and it does not modify memory, so the affine warp can execute independently from the non-affine stream. More importantly, the affine warp fetches memory (but does not use it) on behalf of the non-affine warps, so the affine warp can issue memory requests while bypassing stalls. For example, in Figure 7a, line 9 of the decoupled kernel loads data pointed to by `addrA` in a loop. The affine warp can request `[addr]` for the next iteration without waiting for the requests of the previous iteration to finish, since only non-affine warps operate on data `[addr]` (`tmp`+1). In other words, the original program's data dependence on `[addr]` is broken by executing the use of the data on the non-affine warps.

4.1 Design Overview

Our overall design is shown in Figure 9 with the baseline GPU components appearing in white and the added components in gray. Most of the added hardware is used to handle Enqueue and Dequeue instructions, including the expansion of affine tuples to concrete values, and to support execution of the affine warp.

Because DAC executes only a small number of affine instructions, DAC does not use a dedicated functional unit for the affine warp. Instead, the affine warp executes on SIMT lanes (Section 4.4). Thus, both affine and non-affine warps are fetched, decoded ①, and issued ② to SIMT hardware in the same way.

DAC adds a dedicated warp context for the affine stream and launches one affine warp per SM. DAC launches as many warps as the baseline GPU for the non-affine stream. Due to on-chip resource constraints, the GPU may not be able to concurrently execute all threads of the non-affine stream, so DAC executes the affine warp once for each batch of concurrent non-affine warps.

The affine and non-affine warps are executed concurrently via fine-grain multi-threading. Affine tuple expansion is performed by dedicated hardware in parallel with non-affine warp executions, so the latency of expansion is typically hidden.

When the affine warp executes an `enq` instruction, the associated affine tuple is enqueued ③ to the tail of the **Affine Tuple Queue** (**ATQ**). The **Predicate Expansion Unit** or the **Address Expansion Unit** then fetches the affine tuple from the head of the ATQ ④. Using the affine tuple, the expansion units generate predicate bit masks or coarse-grain addresses for each non-affine warp. A predicate bit mask, for example, is then enqueued ⑤ to the tail of the **Per Warp Predicate Queue** (**PWPQ**). As the name suggests, there is one PWPQ for each concurrent non-affine warp. Finally, when a non-affine warp executes a `deq.pred` instruction, the bit mask is dequeued ⑥ from its PWPQ, and the bit mask is used to set the predicate register. The process is similar for address expansion (`enq.addr`). The expansion unit designs are described in Section 4.2 and 4.3.

For the `enq.data` instruction, which is used for global and local load requests, DAC generates addresses and requests data from memory as soon as the addresses are generated. The requests ⑦ are sent to the L1 data cache and then to the lower levels on cache misses. The requested data is locked in L1 upon retrieval from L2 or DRAM. Later, when a non-affine warp executes the `deq.data` instruction, the data is retrieved from L1 ⑧.

At the scoreboard stage ⑨, DAC checks whether `enq` or `deq` warp instructions are eligible to be issued. For the `enq` instruction, if the ATQ has no available space, or if one of the PWPQs is full, then the affine warp is not allowed to issue. For `deq`, if an non-affine warp's PWPQ or PWAQ is empty, or the data prefetched from main memory is not yet available, then the corresponding non-affine warp is not allowed to issue, so ready non-affine warps are issued instead.

Finally, a dedicated **Affine SIMT Stack** is used to handle the affine warp's control flow ⑩, while the non-affine warps

still use the baseline GPU's SIMT stack. The Affine SIMT Stack allows the affine warp to execute largely independently of the non-affine warp when performing early memory accesses.

We now describe each components in more detail.

4.2 Address Expansion Unit

The Address Expansion Unit (**AEU**) takes affine tuples as input and generates concrete addresses for each non-affine warp.

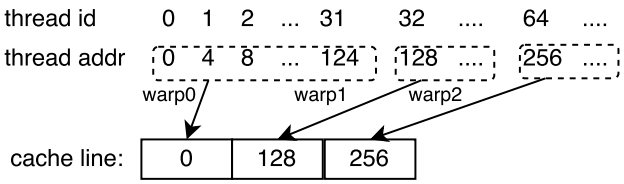


Figure 10: Cache Line Access Regularity: Cache Line References by Warps with an Offset of 4

The AEU generates cache line addresses directly from the affine tuples without generating addresses for individual threads. For example, Figure 10 shows that with an offset of 4, warps access consecutive 128-byte cache lines, so the AEU will generate a sequence of consecutive cache line addresses from the starting address.

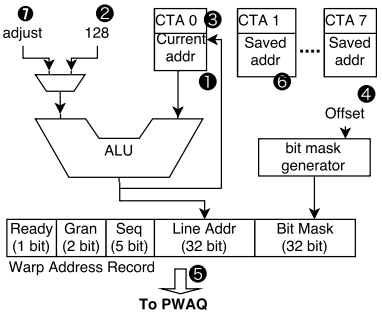


Figure 11: Address Expansion Unit

Figure 11 shows the design of the AEU, which is equipped with a single integer ALU. For each CTA (block), the starting address is computed once per CTA as *base+block_offset×block_index* ①, and the overhead is amortized across threads of the CTA. Thereafter, the address is incremented ② and accumulated ③ by 128 at a time to generate cache-line addresses for consecutive threads and warps.

To indicate which word (of the 128 byte data) a thread should access, the AEU generates a bit mask that accompanies the address ④. For instance, an offset of 4 will generate a bit mask `111111...` to indicate that all 32 words are accessed; similarly, an offset of 8 generates `101010...` to indicate the access of every other word in the region. To reduce the storage and computation overhead of address generation, the address and bit mask are then pushed to the PWAQ as a warp address record ⑤, which is a compact encoding of each individual thread's addresses. The non-affine warps later dequeue the records to perform memory accesses, and data are mapped to threads by the bit masks and the line addresses. The `granularity` bits indicate whether each thread access a word, a half word, or a byte, so the bit mask and line address are interpreted differently.

To avoid stalls (e.g. when a non-affine warp's PWAQ is full), the AEU uses one accumulated address register for each concurrent CTA ⑥, allowing it to switch among CTAs to generate addresses. For multi-dimensional thread indices, it is possible for the consecutive increments to be disrupted when threadIdx.y is incremented by 1. In this case, an adjustment ⑦ is added to the current accumulated value. The adjustment has the same value for all threads so is computed only once.

For the `enq.data` instruction (global and local loads), the AEU also sends requests to the L1 cache or the lower levels of the memory hierarchy on a miss. To avoid the eviction of requests that arrive before their demand accesses, DAC adds lock counters to the tag array, which temporarily disable replacement for a cache line. The AEU locks cache lines upon issuing memory requests, and the the non-affine warp unlocks cache lines upon access. Unlike speculative prefetching, the early requests are guaranteed to be accessed by the non-affine warp and eventually unlocked, so this locking is safe. Memory accesses that are not affine must be issued by the non-affine warps, but deadlock is avoided because the AEU can lock at most (N−1) sets of an N-way cache. It is possible to create contention between locked cache lines and non-affine cache lines, but we do not observe this to be a problem because usually only a small portion of the cache is locked at any given time.

Early memory accesses can cause conflicts with barrier operations (syncthreads). To avoid conflicts, barrier instructions are replicated to both the affine and the non-affine warps. The AEU handles barrier operations on behalf of the affine warp. When the affine warp executes a barrier instruction, the AEU disables expansion for the target non-affine blocks; the AEU only issues memory requests for non-affine blocks that pass the barrier. Affine warps themselves do not access memory (the only access read-only data such kernel parameters), so they are not affected by these barriers.

4.3 Predicate Expansion Unit

The Predicate Expansion Unit (**PEU**) generates predicate bit vectors for the non-affine warps.

Predicate bit vectors are generated by comparisons (e.g. greater-than) between two operations. For a predicate computation to be decoupled, DAC requires that one operand (the *scalar operand*) be a scalar, where all threads in the same block have the same value. If the other operand is also a scalar, then only a single comparison is needed for all threads in the block. For our 29 benchmarks, this case constitutes 64% of the decoupled predicate computations.

In general, the decoupled affine instructions correspond to the regular portion of the original kernel, so the control flows are more likely to be convergent for threads in a warp or in a block/CTA.

If the other operand is not a scalar, then as with the AEU, an accumulation is performed. The idea is that if a warp's first and last thread's values are larger or smaller than the scalar operand, then due to the constant offset of the affine operand [13], all threads in between must have the same result. Thus, a convergent bit mask is generated for a warp with only 2 comparisons. This case constitutes of 93% of the decoupled predicate computations, including the scalar case. For the remaining 7%, the SIMT lanes are used to compare all 32 threads of a warp. Therefore, the PEU optimizes the common cases for bit vector generation.

4.4 Affine Tuple Computation

Instead of using dedicated scalar functional units, DAC performs affine computations on the SIMT lanes, which means that additional ALU operations are available to support more sophisticated affine computations, thereby increasing coverage.

Since CUDA supports up to 3 dimensions of block indices (x,y,z) and thread indices, DAC allows each dimension to have its own offset. Thus, DAC maps one base and up to 6 offsets onto SIMT lanes. Each base or offset of an affine tuple is mapped to a SIMT lane for computation, as shown in Figure 12. For a scalar tuple, the base is mapped to all used lanes to facilitate multiplication between affine and scalar values.

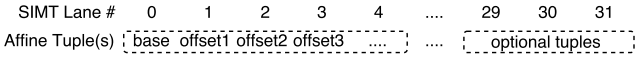


Figure 12: Mapping Affine Tuple(s) to SIMT Lanes

Additional lanes also allow DAC to support modulo operations with a scalar divisor, which are used by some workloads to map addresses. The idea is that threads' values still exhibit regularity after the mod operation, but they must be adjusted by the divisor. The destination operand of a mod operation becomes a special **mod-type** affine tuple: (base, offset, **mod_base**, **divisor**). The **mod_base** is the old base mod divisor, the **divisor** is value of the scalar divisor, and the new base is set to 0. In subsequent computations, the mod-type tuple works with addition (i.e. add to base) with a scalar value, and multiplication with a scalar value (i.e. multiply all fields including the divisor).

The remaining lanes are used to support affine computation after control flow divergence (Section 4.6), where multiple sets of affine tuples are potentially used.

4.5 Affine SIMT Stack

The affine and non-affine instruction streams work as a single kernel, so the control flow that affects affine instructions is replicated to both types of warps. For example, consider a pair of corresponding statements in the affine and the non-affine streams—"`if(tid<bound)` **`enq`**" and "`if(tid<bound)` **`deq`**". The if-statement means that the affine warp should only enqueue and expand the tuple for non-affine threads that require the data (i.e. the non-affine warps with `tid` less than `bound`). Therefore, for these decoupled instructions, the control flow of the affine warp corresponds to that of the non-affine threads, and the affine warp "executes" those threads in lock-step, except it replaces vector computation with affine computation.

The decoupled affine stream can have control flow divergence, which potentially reduces efficiency. However, in general, there are two reasons why the affine warp can still be executed efficiently. First, DAC decouples the regular portions of workloads, which tend to exhibit less divergence. Second, scalar loops, where all threads execute the same number of iterations, are common for decoupled instructions. In our experiments, the affine warp instructions only constitute 4% of total warp instructions on average.

In most cases, the affine warp and the Predicate Expansion Unit already produce the non-affine warps' bit vectors, which are also used by the affine warp for control flow. Otherwise, such as when data dependent control flow occurs, non-affine warps must provide bit-vectors for the affine warp.

To enable affine warps to run ahead of the non-affine warps, we equip the affine warp with its own SIMT stack (the Affine SIMT Stack) for handling control flow. DAC use a two-level Affine SIMT Stack, which exploits convergence at the warp level to reduce the need to check and update control flow on a thread-by-thread basis.

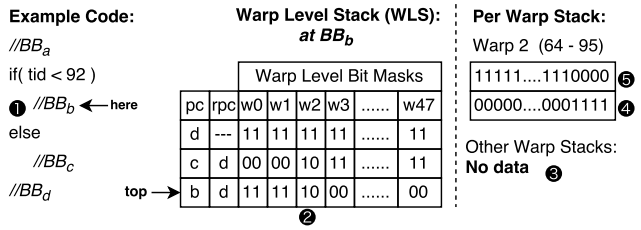


Figure 13: Re-Convergence Stack for the Affine Warp

Figure 13 depicts the two-level stack. The functionality is similar to that of a baseline GPU (and the non-affine SIMT stack). For the code example on the left, threads re-converge at Basic Block D (BB_d), and the affine warp is currently at BB_b ①. The **Warp Level Stack** (**WLS**) encodes each non-affine warps' bit vector with only 2 bits. "11" indicates that all threads in the warp are 1s; "00" denotes all 0s; "10" denotes otherwise. The PC and RPC (re-convergence PC) fields are shared by all warps. The "11" and "00" cases only require checking the WLS without inspecting each threads' bits. The WLS reduces the number of bits that are checked and updated. For the "10" case, **Per Warp Stacks** (**PWSs**) are used for threads within a warp. On a Fermi GPU, 48 PWSs are used for concurrent warps on an SM. In the example, only warp 2 (`w2`) must update its PWS ②, and ④ and ⑤ show the content. All other warps use only WLS, and their PWSs have no data ③. The PWSs do not have PC and RPC fields, which reduces storage.

4.6 Divergent Affine Tuples

When control-flow divergence occurs, a single affine tuple may not be enough to sustain affine computations. To increase

the coverage of affine instructions, DAC uses a combination of compiler-based static analysis and a runtime hardware mechanism to exploit affine computations after limited divergence. Here, we discuss the impact of control-flow divergence and the hardware mechanism.

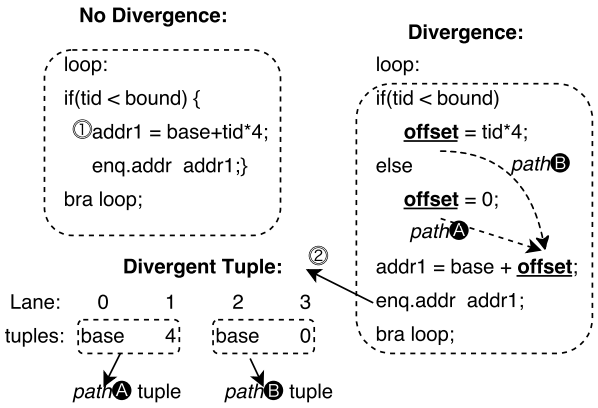


Figure 14: Divergent Base-Offset Pairs on SIMT Lanes

The code on the left of Figure 14 represents a case where a single tuple suffices even after control flow divergence. In this case, the affine warp will not Enqueue `addr1` for inactive threads. All active threads' `addr1`, however, are still computed by the same base and offset ①, so active threads still use the same affine tuple. In this case, it is suffice to mask off the inactive threads when expanding the affine tuple; this is handled by the Affine SIMT Stack (Section 4.5).

The code on the right represents a case where affine tuples become divergent. The common case is that threads compute addresses or predicates differently depending on whether boundary conditions are met. In the example, a thread's value for **offset** can be either `0` or `tid*4`, so `addr1` has two affine tuples for all threads: (base,4) and (base,0).

Since DAC computes affine tuples on SIMT lanes, both `Path` Ⓐ and `Path` Ⓑ tuples can be mapped to SIMT lanes to be computed simultaneously ②. The basic idea is to perform two affine tuples computations for two sets of threads in lock-step, just as in vector computation. In general, at most 2 divergent conditions (or 4 tuples) can affect an affine operand; otherwise, the related affine instructions will not be decoupled. In addition, divergent affine tuples with loop carried dependences will also not be decoupled.

The AEU selects whether to expand `Tuple` Ⓐ or `Tuple` Ⓑ based on the control flow of each thread. The compiler technique for detecting divergent tuples and the hardware mechanism for expansion selection are described in Section 4.7.

We also extend the concept of divergence to instructions, such as max, min, and abs, which incorporate both value assignment and predication. For instance, "`max dst, src1, src2;`" is equivalent to "`dst = (src1 > src2) ? src1 : src2`".

4.7 Compilation

The compiler is responsible for taking unmodified kernel code and decoupling it into affine and non-affine instruction streams. This decoupling involves two main tasks: (1) identifying affine operands and instructions, and (2) identifying divergent affine tuples and conditions.

Identifying Affine Operands. Our technique for identifying affine operands is derived from previous solutions [6, 13]. Each operand is classified as one of three possible types: *scalar* (e.g. kernel parameters), *affine* (e.g. threadIdx), or *non-affine* (e.g. memory), which are listed in order from most specific to most general. The compiler initially assigns types to non-register operands. By creating a control-flow graph (CFG) and performing reaching definition analysis on the CFG, the initial types are iteratively propagated through register operands and instructions.

At each instruction, if more than one definition reaches a source operand, the most general type among the definitions is assigned to the source operand. The destination operand of an instruction is assigned the most general type among the source operands. Instructions with operations not supported by affine computation produce *non-affine* destination operands directly.

After the classification process, memory access and predicate computation instructions with *scalar* and *affine* type source operands are candidates for decoupling into the affine stream. We refer to instructions that define another instruction's source operands as **predecessors**. From each candidate memory and predicate instruction, the compiler recursively traverses the CFG backwards to checks its predecessor instructions for divergence by performing Divergent Affine Analysis, which we explain in the next section.

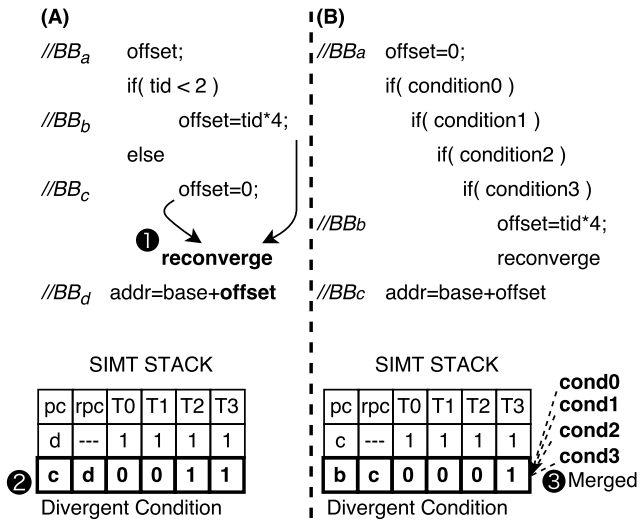


Figure 15: Using SIMT Entry as Divergent Condition

Divergent Affine Analysis. The goal of Divergent Affine Analysis is to identify divergent affine tuples (described in Section 4.6) along a recursive predecessor chain. At each predecessor, the compiler recognizes divergent affine tuples when a source operand has two or more reaching definitions. For example, in Figure 15 (a), "`offset`" is defined at BB_b and BB_c before reaching BB_d, so "`addr`" has two affine tuples to expand. At run time, for each thread, expansion units choose one of the affine tuples according to the thread's control flow. We call the conditions for making the choice **divergent affine conditions**.

Since the affine warp uses the Affine SIMT Stack for handling control flow, we use SIMT stack entries as divergent affine conditions. Using Figure 15(a) as an example, the compiler identifies the re-convergence point of two reaching definitions' basic blocks ①. The last SIMT stack entry before re-convergence ② is the divergent affine condition, since it distinguishes threads that use BB_b's definition from those that use BB_c's.

The use of SIMT stack entries as divergent conditions potentially reduces the amount of condition checking. The example in Figure 15(b) shows that multiple branches may cause a single divergent reaching definition. However, only one SIMT stack entry is needed for divergent affine condition ③. There is no need to check multiple branch conditions, since the SIMT stack already checks and merges those conditions for handling control-flow.

Divergent reaching definitions can occur at an arbitrary predecessor of an affine instruction, where expansions are required. Thus, DAC saves to a dedicated Divergent Condition Register File (DCRF) the required SIMT stack entries (bit-vectors) at re-convergence points, so that they can be checked by expansion units later. As with the Affine SIMT Stack (Section 4.5), the DCRF has a two-level structure but used as a register file rather than a stack.

After detecting a divergent affine tuple, the re-convergence points are marked by the compiler, and a DCRF entries is allocated by the compiler.

Decoupling. After divergent analysis, if a candidate affine memory or predicate computation instruction has two or fewer divergent conditions for all its recursive predecessor instructions, then it is eligible for decoupling.

Eligible affine instructions become Enqueue instructions in the affine stream and Dequeue instructions in the non-affine stream. Their predecessor instructions are placed in the affine stream. The predecessor instructions are removed from the non-affine stream provided that no non-affine instruction depends on them.

4.8 Area Estimation

Most of DAC's hardware budget goes to expansion units, which add 2 ALUs per SM, and to the various SRAM components, which add 6 KB per SM. The Affine Tuple Queue (ATQ) has 24 entries, with a total size of 393 bytes. The Per Warp Address Queue (PWPQ) has 192 entries partitioned among warps, with a total size of 1560 bytes. Similarly, the Per Warp Address Queue (PWPQ) has 192 entries and a total size of 768 bytes.

The Affine SIMT Stack has a depth of 8. It has (1) a Warp Level Stack (WLS) with bit masks, PC, and RPC, which require 224 bytes, and (2) per warp stacks with bit masks only, which require 1536 bytes. The Divergent Condition Register File (DCRF) has the same amount of storage as the Affine SIMT Stack.

We model the SRAM components using CACTI [19], which yields 0.21 mm^2 of estimated area per SM. We estimate the area of 2 ALUs with the model used in GPUWattch [18], which yields 0.16 mm^2 per SM. On a GTX 480, with a die size of 520 mm^2 [10], the area overhead is 1.06%.

5 EVALUATION

We now evaluate DAC by comparing it against both affine computation and GPU prefetching solutions.

5.1 Methodology

To evaluate performance, we use GPGPU-sim 3.2.2 [4], and to evaluate energy, we use GPUWattch [18]. The baseline GPU is modeled after a Fermi GTX 480 with simulation parameters shown in Table 1. We use CACTI 5.3 [19] to model the energy overhead of DAC's added SRAM components.

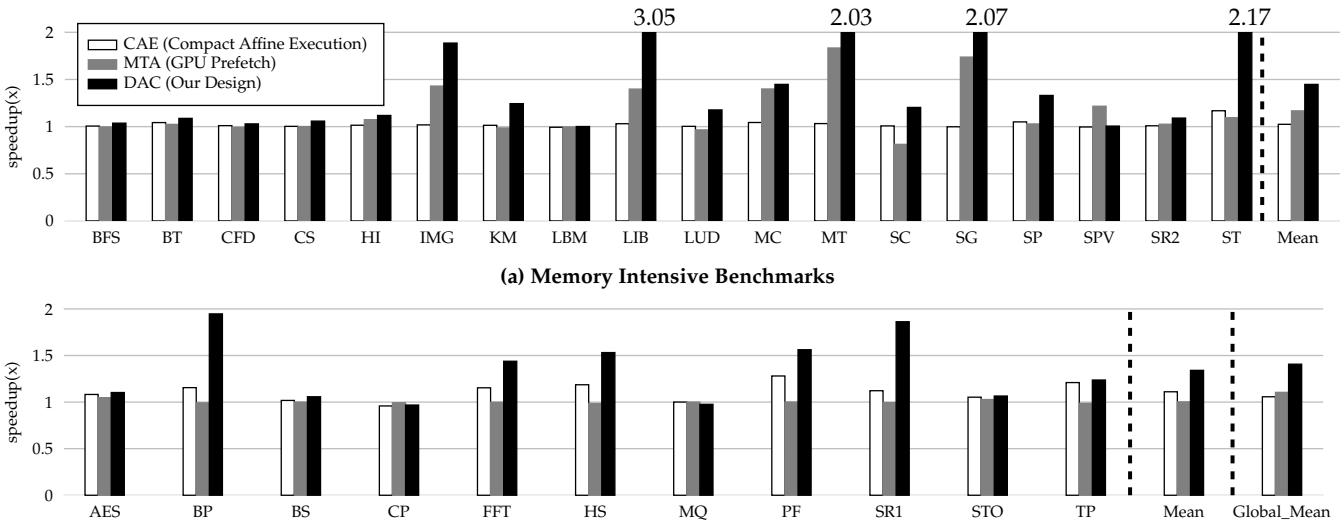
Baseline GPU	
GPU	Fermi (GTX480), 15 SMs, 48 warps/SM
SM	32 SIMT lanes, 128KB register file
Scheduler	2 Schedulers/SM, Two Level Active [20]
L1	48 KB/SM, 4 Ways, 32 MSHRs
L2	769 KB, 6 Partitions, 8 Ways
GPU Prefetcher (MTA)	
Prefetch Buffer	16KB/SM (in addition to the 48KB L1)
Compact Affine Execution (CAE)	
Affine Units	2 Affine Units per SM (one per 16 lanes)
Decoupled Affine Computation (DAC)	
ATQ (per SM)	24 Entries, 392 bytes, 5.3 pJ/Access
PWAQ (per SM)	192 Entries, 1560 bytes, 3.4 pJ/Access
PWPQ (per SM)	192 Entries, 768 bytes, 1.5 pJ/Access
PWS (per SM)	8×48 Entries, 1536 bytes, 2.7 pJ/Access
PWS (per SM)	8×48 Entries, 1536 bytes, 2.7 pJ/Access

Table 1: Simulation Parameters

We simulate all benchmarks with SASS, which is the native instruction set executed directly on GPU hardware. GPGPU-sim parses the SASS assembly code produced by the CUDA tool-chain and generates PTXPLUS, which is the instruction set used by the simulator. PTXPLUS corresponds almost exactly with SASS; the conversion is merely syntactic.

For DAC, the compiler's decoupling of kernels (Section 4.7) is performed on the PTXPLUS instructions in GPGPU-sim's front-end before simulation starts.

5.1.1 Baseline Techniques. To evaluate both the computational and memory latency-hiding aspects of DAC, we also implement two other state-of-the-art designs based on previously proposed techniques, which we now describe. In each case, we provision these techniques with extra hardware that we do not give to DAC.



(a) Memory Intensive Benchmarks

(b) Compute Intensive Benchmarks

Figure 16: Speedup of CAE, MTA, and DAC over the Baseline GTX 480 GPU

Compute Intensive			**Memory Intensive (cont)**		
Name	Abbr.	Suite	Name	Abbr.	Suite
CP	CP	G	imghisto	IMG	G
STO	STO	G	histogram	HI	R
AES	AES	G	LBM	LBM	R
mri_q	MQ	G	SPMV	SPV	R
tpacf	TP	G	b+tree	BT	C
FFT	FFT	G	LUD	LUD	C
backprop	BP	C	sradv2	SR2	C
sradv1	SR1	C	stream cluster	SC	C
hotspot	HS	C	KMEANS	KM	C
pathfinder	PF	C	BFS	BFS	C
blackscholes	BS	P	CFD	CFD	C
Memory Intensive			monte carlo	MC	P
LIB	LIB	G	mersenne twister	MT	P
sgemm	SG	R	Scalar Product	SP	P
stencil	ST	R	Convolution Sep.	CS	P

Table 2: List of Benchmarks – G: GPGPU-sim distribution [4], R: Rodinia benchmark suite [5], C: CUDA SDK, P: Parboil benchmark suite [24]

Compact Affine Execution (CAE). To evaluate DAC's efficiency in handling affine computations, we compare against CAE, which augments the baseline GPU with an affine data path based on Kim et al's design [13].

CAE tracks affine operands at run time to determine which warp instructions are eligible for affine computation. After fetch-decode, eligible warp instructions are sent not to the SIMT lanes but to the affine function units for execution. CAE improves efficiency by replacing vector computations with affine computations for threads within a warp.

The original work adds a single affine unit to each SM. As mentioned by Kim, et al [13], for a 32 lane GPU, the affine unit yields little performance benefit, since a warp with 32 threads will take the same number of cycles to execute whether it is using the SIMT lanes or the affine unit. Still, the affine unit has the benefit of reducing SIMT lane occupancy.

The original CAE work [13] uses one affine functional unit for each SM, but to illustrate the benefit of reduced lane occupancy, our implementation of CAE uses two affine units for each SM. The baseline GTX 480 has two schedulers, where each scheduler individually issues instructions to 16 SIMT lanes [21]. For the baseline GPU, each scheduler takes **two** cycles to issue one warp instruction, since a 32 thread warp is issued to 16 lanes. For our CAE with two affine units (one for each scheduler), each scheduler takes only **one** cycle to issue an affine instruction, since affine instructions do not occupy SIMT lanes. Ideally, the computation throughput is doubled for affine instructions on CAE. However, CAE still executes the same number of instructions as the baseline GPU, and the instruction issue-scheduling rate is doubled to fully utilize the affine units.

GPU Prefetcher (MTA). To evaluate DAC's ability to hide memory latency, we compare it to a system that augments the baseline GPU with a data prefetcher based on Many-Thread Aware prefetching (MTA) [15].

MTA detects both intra-warp memory access offsets (e.g. load instructions in loops within a warp) and inter-thread offsets (e.g. load instructions issued by adjacent warps) for a few SIMT threads. The regularity is then speculatively generalized to all warps to achieve scalable prefetching. In addition, a throttling mechanism is used to control the aggressiveness of prefetching based on the number of evicted cache lines that are prefetched but not used by the GPU [15].

To alleviate cache pollution [12, 15], we give MTA a dedicated 16KB prefetch buffer on each SM, in addition to the 48KB L1 cache.

5.1.2 Benchmarks. We evaluate 29 benchmarks from 4 suites as shown in Table 2. We divide them into two categories: memory intensive and compute intensive. We consider a benchmark to be *memory intensive* if the baseline GPU can achieve a speedup of at least 1.5 when using a perfect memory system (i.e. no latency and unlimited bandwidth). The remaining benchmarks are considered to be *compute intensive*.

5.2 Speedup

Figure 16 shows the speedup of DAC, CAE, and MTA over the baseline GTX 480 GPU for our 29 benchmarks. DAC's geometric mean speedup of 1.40 is significantly better than either CAE's or MTA's. As expected, CAE provides benefits for just the compute-intensive benchmarks, while MTA provides benefits for just the memory-intensive benchmarks. Not only does DAC improve the performance of both classes of programs, but it offers the best performance within each class of programs. For the compute-intensive benchmarks, DAC has a speedup of 1.34, while CAE achieves a speedup of 1.15. For the memory-intensive benchmarks, DAC produces a speedup of 1.44, while MTA achieves a speedup of 1.16.

5.3 Instruction Execution Reduction

Figure 17 shows that for the 29 benchmarks, DAC executes on average 0.74× as many warp instructions as the baseline GPU. So DAC reduces the dynamic instruction count by 26%, which in turn reduces execution time and improves energy efficiency; the effect is particularly evident for compute-intensive benchmarks. Only 4.6% of the instructions executed on DAC are affine instructions (See Figure 17), showing that DAC does not require a dedicated affine functional unit.

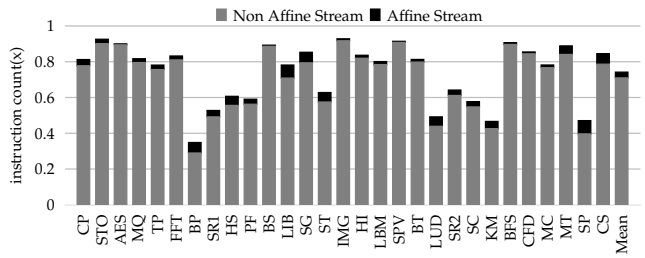


Figure 17: Number of Warp Instructions Executed by DAC Normalized to the Baseline GPU

With two affine units per SM (and two warp schedulers), our implementation of CAE doubles the affine instruction throughput compared to the baseline. By contrast, DAC executes a single affine instruction to replace 9 instructions on the baseline GPU on average, so it increases execution throughput for affine instructions by 9× over the baseline GPU.

5.4 Affine Instruction Coverage

The coverage of affine instructions is the percentage of warp instructions executed by the baseline GPU that could be handled as affine instructions by CAE or DAC. For the the 11 compute intensive benchmarks, Figure 18 shows that DAC

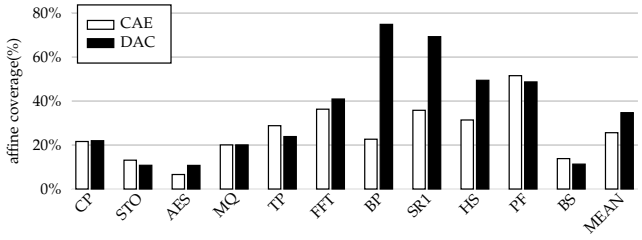


Figure 18: Affine Instruction Coverage of DAC and CAE

achieves a geometric mean coverage of 34%, compared to 25% for CAE.

Because DAC identifies affine computations statically and because DAC uses SIMT lanes to execute affine instructions, DAC supports affine computations after limited control flow divergence and uses offloaded affine SIMT stack entries to reduce overhead. By contrast, CAE has no facilities for performing affine computations after divergence. While CAE's scheme for identifying affine instructions is more flexible, CAE must use the SIMT lanes to expand any affine tuples involved in divergence back to vector values [13]. Moreover, CAE's affine functional unit uses a single ALU for offset computations, which requires all 32 threads of a warp to have the same offset pattern. For benchmarks, such as HT and BP, whose last-level dimension is smaller than 32, CAE only handles scalar computations (i.e. an offset of 0), since the threads in a warp do not follow a single offset pattern.

5.5 Memory Latency Hiding

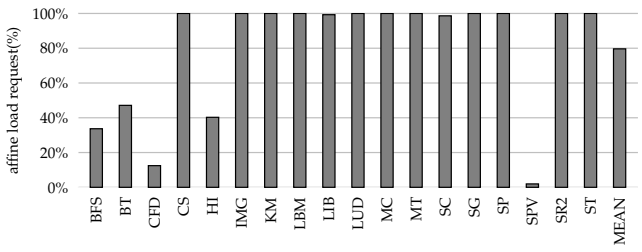


Figure 19: Percentage of Affine Global and Local Load Requests on DAC

DAC can hide memory latency because the affine warp can run ahead of non-affine warps and issue load requests without waiting for previous requests to finish. An indicator for DAC's latency hiding ability is the percentage of global and local load addresses that are produced by affine instructions, which can be issued by the affine warp. Figure 19 shows that for our memory-intensive benchmarks, an average of 79.8% of the global and local load requests are issued by the affine warp. Many benchmarks have close to 100% coverage, because regular SIMT workloads often use scalar data and thread IDs to map memory addresses for coalesced memory accesses.

For benchmarks such as BFS and BT that make heavy use of indirect memory addresses to access complex data structures,

DAC sees little performance improvement. In addition, benchmarks may also be constrained by bandwidth, row buffer locality, or bank conflicts; in such cases, the affine warp might not run ahead sufficiently. Therefore, some benchmarks (e.g. LBM) show little performance improvement despite the high percentage of affine memory requests.

MTA and DAC use different mechanisms to hide memory latency. In DAC, affine memory requests are non-speculative and are generated by instruction executions of the affine warp. By contrast, MTA hides latency by speculatively issuing prefetch requests when triggered by on-demand memory accesses.

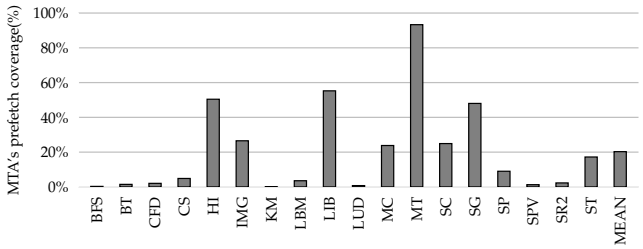


Figure 20: MTA Prefetcher Coverage

Many SIMT workloads have highly regular memory accesses, so the MTA prefetcher has high prediction accuracy. MTA's latency hiding ability is correlated with prefetcher coverage (see Figure 20), which is defined to be the number of L2 and DRAM accesses that are covered by the prefetcher. We see that MTA's throttling mechanism reduces harmful prefetches, but it also reduces coverage when additional bandwidth are available in some cases. In some other cases (e.g. SC), the throttling mechanism does not prevent cache pollution.

5.6 Energy Efficiency

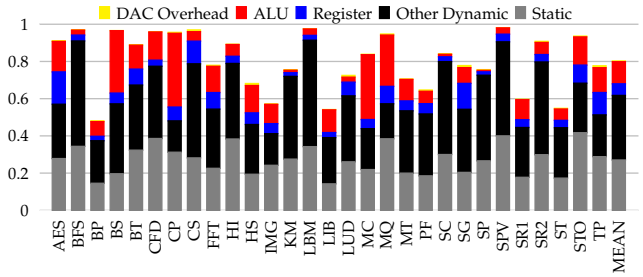


Figure 21: Energy Consumption of DAC Normalized to the Baseline GPU

Figure 21 shows the total energy consumption (dynamic and static) of DAC normalized to the baseline GPU. For our 29 benchmarks, the geometric mean is 0.798. Thus, DAC reduces total energy by 20.2%, and it reduces dynamic energy alone by 18.4%. The major sources of savings are reduced ALU operations and reduced register accesses due to reduced dynamic instruction executions. DAC reduces the number of ALU operations by 44% and the energy consumption of ALUs by 34%. DAC also reduces the number of register accesses by 17% and the energy consumption of the register file by 32%. By reducing execution time, DAC reduces static energy consumption by 29%.

The overhead of DAC is only 0.96% of the dynamic energy consumption. Most of the overhead comes from the expansion of affine tuples. The expansion units are efficient since they typically use only one or two ALU operations to expand an affine tuple for a given warp.

6 CONCLUSIONS

In this paper, we have shown how two distinct ideas—affine computations and Decoupled Access Execution (DAE)—can be synergistically combined to greatly improve the performance and energy efficiency of SIMT GPUs.

First, specialized support for affine computations on SIMT GPUs has until now preserved the model in which a single instruction stream executes on all warps, which limits the redundancy reduction to within a single warp. By decoupling the affine computations to a separate affine instruction stream, DAC overcomes this limitation, allowing a single affine warp to produce values for many non-affine warps and to reduce warp instruction count.

Second, a naive implementation of DAE on GPUs would imply a doubling of the number of threads, but because affine computations represent such a large reduction in computation, DAC focuses on affine memory accesses and adds one warp per SM to significantly hide memory latency.

The result is a system that improves performance and energy efficiency for both memory-intensive and compute-intensive workloads, reducing total energy consumption by 20.2%, and achieving a speedup of 40.7%.

Acknowledgments. We thank Don Fussell, Akanksha Jain, and the anonymous referees for their valuable feedback on early drafts of this paper. This work was funded in part by NSF grants CNS-1543014 and DRL-1441009 and by a gift from the Qualcomm Foundation.

REFERENCES

[1] AMD. 2012. AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE. (2012). https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf

[2] José-María Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2012. Boosting Mobile GPU Performance with a Decoupled Access/Execute Fragment Processor. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 84–93. http://dl.acm.org/citation.cfm?id=2337159.2337169

[3] Krste Asanovic, Stephen W. Keckler, Yunsup Lee, Ronny Krashinsky, and Vinod Grover. 2013. Convergence and Scalarization for Data-parallel Architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/CGO.2013.6494995

[4] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International*

Symposium on Workload Characterization (IISWC) (IISWC '09). IEEE Computer Society, Washington, DC, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[6] Sylvain Collange, David Defour, and Yao Zhang. 2010. Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations. In *Proceedings of the 2009 International Conference on Parallel Processing (Euro-Par'09)*. Springer-Verlag, Berlin, Heidelberg, 46–55. http://dl.acm.org/citation.cfm?id=1884795.1884804

[7] Neal Clayton Crago and Sanjay Jeram Patel. 2011. OUTRIDER: Efficient Memory Latency Tolerance with Decoupled Strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/2000064.2000079

[8] Roger Espasa and Mateo Valero. 1996. Decoupled vector architectures. In *Proceedings. Second International Symposium on High-Performance Computer Architecture*. 281–290. https://doi.org/10.1109/HPCA.1996.501193

[9] Syed Zohaib Gilani, Nam Sung Kim, and Michael J. Schulte. 2012. Power-efficient Computing for Compute-intensive GPGPU Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 445–446. https://doi.org/10.1145/2370816.2370888

[10] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[11] Hyeran Jeon, Gunjae Koo, and Murali Annavaram. 2014. *CTA-aware Prefetching for GPGPU*. Technical Report CENG-2014-08. Dept. of Electrical Engineering, University of Southern California. http://ceng.usc.edu/techreports/2014/Annavaram%20CENG-2014-08.pdf

[12] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 332–343. https://doi.org/10.1145/2485922.2485951

[13] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. 2013. Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 130–141. https://doi.org/10.1145/2485922.2485934

[14] Keunsoo Kim, Sangpil Lee, Myung Kuk Yoon, Gunjae Koo, Won Woo Ro, and Murali Annavaram. 2016. Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 163–175.

[15] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. 2010. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 213–224. https://doi.org/10.1109/MICRO.2010.44

[16] Yunsup Lee. 2016. *Decoupled Vector-Fetch Architecture with a Scalarizing Compiler*. Technical Report UCB/EECS-2016-117. EECS Department, University of California, Berkeley.

[17] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2011. Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/2000064.2000080

[18] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 487–498. https://doi.org/10.1145/2485922.2485964

[19] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to understand large caches. *University of Utah and Hewlett Packard Laboratories, Tech. Rep* (2009).

[20] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 308–317. https://doi.org/10.1145/2155620.2155656

[21] NVIDIA. 2009. Fermi Architecture Whitepaper. (2009). http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

[22] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. 2013. APOGEE: Adaptive Prefetching on GPUs for Energy Efficiency. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 73–82. http://dl.acm.org/citation.cfm?id=2523721.2523735

[23] James E. Smith. 1982. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th International Symposium on Computer Architecture (ISCA)*. 112–119.

[24] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. 2012. *Parboil: A revised benchmark suite for scientific and commercial throughput computing*. Technical Report IMPACT 12-01. Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaigne.

[25] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, Lisa R. Hsu, and Huiyang Zhou. 2013. Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 433–442. https://doi.org/10.1145/2464996.2465022

[26] Yi Yang, Ping Xiang, Michael Mantor, Norman Rubin, Lisa Hsu, Qunfeng Dong, and Huiyang Zhou. 2014. A Case for a Flexible Scalar Unit in SIMT Architecture. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE Computer Society, Washington, DC, USA, 93–102. https://doi.org/10.1109/IPDPS.2014.21

[27] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. 2012. CPU-assisted GPGPU on Fused CPU-GPU Architectures. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/HPCA.2012.6168948