HAL
open science

Affine Vector Cache for memory bandwidth savings

Caroline Collange, Alexandre Kouyoumdjian

To cite this version:

Caroline Collange, Alexandre Kouyoumdjian. Affine Vector Cache for memory bandwidth savings. [Research Report] ENS de Lyon. 2011. ensl-00649200

HAL Id: ensl-00649200

https://ens-lyon.hal.science/ensl-00649200v1

Submitted on 7 Dec 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Affine Vector Cache for memory bandwidth savings

Caroline Collange, Alexandre Kouyoumdjian
ENS de Lyon, Université de Lyon, Arénaire,
LIP (UMR 5668 CNRS - ENS Lyon - INRIA - UCBL)

Abstract

Preserving memory locality is a major issue in highly-multithreaded architectures such as GPUs. These architectures hide latency by maintaining a large number of threads in flight. As each thread needs to maintain a private working set, all threads collectively put tremendous pressure on on-chip memory arrays, at significant cost in area and power. We show that thread-private data in GPU-like implicit SIMD architectures can be compressed by a factor up to 16 by taking advantage of correlations between values held by different threads. We propose the Affine Vector Cache, a compressed cache design that complements the first level cache. Evaluation by simulation on the SDK and Rodinia benchmarks shows that a 32KB L1 cache assisted by a 16KB AVC presents a 59% larger usable capacity on average compared to a single 48KB L1 cache. It results in a global performance increase of 5.7% along with an energy reduction of 11% for a negligible hardware cost.

1 Introduction

Modern Graphics Processing Units (GPUs) are increasingly used as general-purpose data-parallel processors. Their architecture obeys the implicit-SIMD execution model, which is also called Single Instruction Multiple Threads (SIMT) by NVIDIA. Implicit-SIMD processors run multiple simultaneous threads in lockstep on replicated ALUs, and let them share cooperatively an instruction fetch unit and a memory access unit. This gives them the efficiency advantage of Single-Instruction, Multiple-Data (SIMD) execution, while retaining a scalable multi-thread programming model [14].

Current high-end GPUs maintain from 25000 to 80000 fine-grained threads in flight. They leverage this massive explicit thread-level parallelism both through concurrent execution to increase throughput and through interleaved execution to hide latencies. However, the sheer number of threads running simultaneously brings heavy contention on the register files and caches [15]. For instance, the NVIDIA Fermi architecture in its default configuration shares a 128KB register file and a 16KB L1 cache among a maximum of 1536 threads. In this configuration, each thread has only 21 registers and 10 bytes of cached private memory at its disposal.

Further increasing the size of the register file and caches appears impractical, as current GPUs already dedicate significant resources to register files and caches. On-chip memory sizes range from 4MB (NVIDIA Fermi [21]) to 12MB (AMD GCN [9]) on current GPUs. When the working set of each thread cannot fit in on-chip memory, parallelism has to be throttled down, hindering the ability to hide memory latency, or external memory has to be used, at a significant cost in power consumption [5, 16].

In order to relieve pressure on first-level caches, we propose to complement the L1 cache with a new hardware structure named the Affine Vector Cache (AVC). The AVC stores blocks of memory that obey specific patterns named *affine vectors*. Affine vectors consist of either a sequence of equal integers, or a sequence of uniformly-increasing integers. As we will see, affine vectors represent an average of 67% of the private memory traffic to the L1 cache when 50% of each thread's working set is allocated to the register file, and can attain 100% on a class of applications in the high-performance computing segment.

The AVC stores affine vectors using a compact encoding scheme, reducing the load on the L1 cache for a small area footprint. Our simulation results show that replacing 16KB of cache space by a 16KB-AVC improves the apparent cache capacity by more than half on a GPU architecture modeled after NVIDIA Fermi [21]. As a consequence, the AVC reduces dynamic energy consumption at the board level by up to 39% (11% on average) in a high register-pressure context. By reusing the existing cache memory arrays, we keep the area overhead negligible.

We first give an overview of implicit-SIMD architectures and introduce inter-thread correlation patterns in section 2. We then conduct a suitability analysis by characterizing dynamic load and store traffic to thread-private memory in section 3. The proposed AVC architecture is

described in section 4. Finally, section 5 experimentally evaluates the AVC and its impact on memory bandwidth usage, power and performance, and quantifies the hardware overhead.

2 Background

We will consider in this paper the case of a GPGPU architecture inspired from NVIDIA GPUs [19, 21], although the AVC can benefit any implicit-SIMD architecture that emphasizes inter-thread locality [13, 20].

2.1 Implicit SIMD

GPUs follow a single program, multiple data (SPMD) programming model. Many parallel instances, or *threads* running a single *kernel* are executed in parallel. The execution order of threads is undefined outside of explicit synchronization barriers.

To amortize the cost of instruction fetch and pipeline control over multiple execution units, GPUs take advantage of implicit SIMD execution. Threads are grouped into so-called *warps*. All threads in a warp run in lockstep, so that one instruction can be fetched and decoded on behalf of multiple threads. Warps are typically composed of 32 to 64 threads. Multiple warps are kept in flight on each *streaming multiprocessor* (SM) to hide memory and pipeline latency. Each SM has a physical register file which is partitioned between all warps running on the SM. The number of registers per thread r is set by the compiler, following optional programmer-provided constraints. A GPU chip has multiple independent SMs for scalability.

From the perspective of the hardware, each instruction can be thought of as operating on warp-sized *vectors* as in an explicit SIMD processor. Each register is formed by a vector of data words $\mathbf{d} = (d_0, d_1 \ldots d_{n-1})$, where n is the warp size. We assume that each word is 32-bit wide. As each thread inside a warp may encounter control flow statements leading to different execution paths, threads of a warp are allowed to diverge, then reconverge at a later time. Threads are individually disabled as the SM sequentially runs through all paths taken by at least one thread. All instructions are implicitly predicated by a hardware-managed activity mask $\mathbf{m}$ that indicates the activity status of each thread of the warp: $\mathbf{m} = (m_0, m_1 \ldots m_{n-1}), m_i \in \{0, 1\}$, where $m_i = 1$ means thread i is active.

2.2 Memory layout

Memory is split into distinct spaces, which include at least a global memory space shared among all running threads, and a thread-private memory space. We will focus on the private memory space for the remainder of this paper, as it constitutes the primary target for affine compression. The private memory space contains thread-private data such as the call stack[1]. The private spaces of threads are interleaved on a word-by-word basis in physical memory. This interleaved layout emphasizes inter-thread spacial locality by assigning consecutive memory locations to threads that run simultaneously on the same SM.

Warp size, interleaving stride and cache block size are adjusted so that memory accesses of all threads in a warp fall within the same cache when they all request the same private address. The operating system or device driver maps private address ranges so that no false sharing of private memory occurs between the caches of different SMs. Private memory does not impose any cache coherence requirement beyond presenting a consistent view of memory to each thread. We assume 128-byte cache blocks, 32-wide warps and 4-byte interleaving as in the NVIDIA Fermi GPU architecture [22].

When threads of a warp request different private addresses, as can happen with indirect array indexing, threads are serialized so the memory access is broken down into multiple transactions [22]. Thus, all memory accesses can be reduced to individual transactions performed on a data vector predicated by a subset of the activity mask.

2.3 Inter-thread data correlation

Implicit SIMD execution and warp width are essentially micro-architectural features that are not exposed in the programming model. The concept of scalar variables in explicit-SIMD architectures has no equivalent in the implicit-SIMD model. Instead, scalar variables such as loop counters and memory addresses are duplicated and computed independently by all threads, leading to uniform and affine vectors.

Following the terminology from [6], we categorize warp-sized vectors of data $\mathbf{d}$ associated with activity masks $\mathbf{m}$ into three classes:

Uniform	$d_i = c$ when m_i is set
Affine	$d_i = b + i \times s$ when m_i is set
Generic	Non-affine (implies non-uniform)

[1] We refer here to the memory space known as *local* in CUDA and as *private* in OpenCL, which is implemented using off-chip *scratch buffers* on AMD GPUs

Uniform and affine vectors have been shown to represent together 44% of register reads and 28% of register write-backs in some GPU computing applications, even when conservatively assuming the activity mask to be all ones [6]. However, this analysis is restricted to the register file, and does not consider memory traffic. In this paper, we assess the presence of affine vectors in memory transfers.

For ease of implementation, we will restrict strides of affine vectors to powers of 2 and zero, and require the base to be a multiple of stride. Restricted affine vectors are either uniform or are such that $d_i = (\beta + i) \times 2^{\sigma}$. Supporting non-power-of-two strides provides little additional benefit, as they account for less than 3% of affine vectors in the GPGPU applications considered in the coming section.

3 Affine vectors in private memory

We conduct an experimental analysis on the characteristics of private memory traffic with respect to affine vectors.

3.1 Methodology

We consider a set of high-register-pressure benchmarks from the NVIDIA CUDA SDK [23] and Rodinia [3]. Register pressure was recognized to be a factor limiting performance in several benchmarks of the Rodinia suite, such as the *Needleman-Wunsch* application [3].

The optimal share of register-file space that should be allocated to each thread depends on many parameters which are both application-specific and architecture-specific [22, 30]. Rather than opt for a single register count for each application, we perform an exhaustive analysis over the whole possible range of register counts. Our intent is to evaluate scenarios where the capacity of the physical register file and the capacity of the L1 cache are re-balanced.

Each benchmark is compiled using NVIDIA's compiler *nvcc*, with incrementally tighter register constraints. Starting from the unconstrained register count r_{default}, we gradually reduce the register count until register allocation fails. The upper bound on register count r_{base} is the minimal per-thread register count below which the compiler starts to spill variables to private memory. In general, r_{base} will be slightly lower than r_{default}. We only consider kernels that exhibit medium-to-high register pressure, by ensuring $r_{\text{base}} \geq 8$. The two kernels of *Needleman-Wunsch* are almost identical and gave similar results. We show results for the first kernel only. The values obtained for r_{default} and r_{base} on the Tesla architecture are shown for each kernel in the first two bars of figure 1. For reference, the default register counts obtained on the Fermi architecture in 64-bit mode are shown on the third bar.

For each valid register count, benchmarks are simulated on the Barra GPU simulator [4] to characterize the memory traffic to private memory. Our baseline architecture is based on the NVIDIA Tesla architecture [19], with a micro-architecture modeled after NVIDIA Fermi GPUs [21], as the latter offer data caches. As the traffic generated by private memory accesses is not affected by the topology of the memory hierarchy, we perform the analysis on a single-SM GPU model. Architectural parameters are summarized in Table 1.

Table 1: Characteristics of the baseline SM

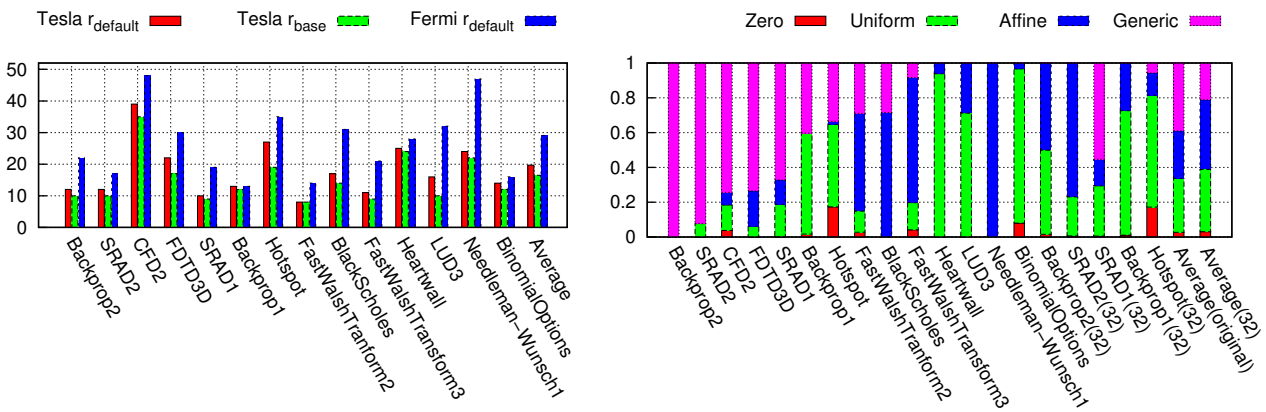
Parameter	Values
Warp size	$n = 32$
Max warps	$w_{\max} = 48$
Total registers	$r_{\text{total}} = 512$
Private L1	48K, 6-way, pseudo-LRU

3.2 Application characterization

For each configuration, we count and classify dynamic load and store instructions on private memory. The ratios of zero, (non-zero) uniform, and (non-uniform) affine transactions over all transactions are summarized in figure 1(b) for a register-file share of 50% of the working set ($r = 0.5\, r_{\text{base}}$). Ratios observed at other register pressure points are similar. For $r = 0.75\, r_{\text{base}}$, the average affine ratio is 66%, while it is 67% at $r = 0.5\, r_{\text{base}}$. We give cumulative values for load and store transaction counts, as their impact on memory-subsystem pressure is comparable. The averages are computed by giving the same weight to all applications, regardless of the number of kernels.

Kernels can be analyzed by realizing that registers are used to retain two different kinds of data: i) *Computation data* are used directly in computations, as part of the program data-flow, and are typically floating-point data. ii) *Control data* are pointers, array indexes and loop counters, which are used primarily in address calculations and comparisons that drive the control flow. Through manual examination of the sources of affine vectors, we make the empirical observation that affine vectors are most commonly found in control data.

Figure 2 plots the private memory traffic profile of selected kernels, with global memory traffic for reference.



(a) Register count

(b) Traffic breakdown, for $r = 0.5\, r_{\text{base}}$ including original benchmarks and 32-wide-block benchmarks (32).

Figure 1: Memory traffic analysis.

We discuss specific aspects of the benchmarks in the following paragraphs.

Persistent computation data The CFD kernel is compute-intensive and has a working set mostly made of computation data. Conversely, its memory access patterns are straightforward and can be computed from few addresses. This accounts for the modest affine vector ratio observed on Figure 1(b) and Figure 2(a).

Non warp-aligned memory blocking *Backprop*, *Hotspot* and *SRAD* access memory by blocks of 16×16 words. As a result, 32-wide vectors of memory addresses are the concatenation of two disjoint 16-wide affine vectors, and are not affine vectors. We believe this choice of blocking dimensions is also the result of an optimization for the Tesla architecture, which has a 16-wide memory pipeline and a limited 16KB shared memory [22].

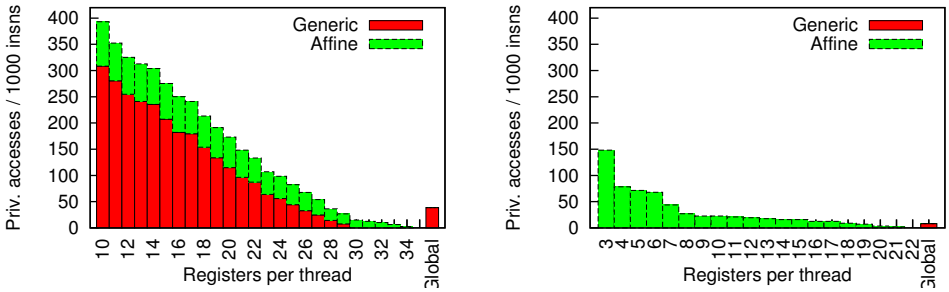
The applications were manually modified to use thread blocks of size 32×32. The results obtained with these updated benchmarks are shown on the right side of figure 1(b). The average affine ratio increases to 80%. These results suggests that current GPU computing applications may benefit from considering affine vectors at 16-word granularity. We consider 32-wide affine vectors in this paper as it simplifies the discussion, though the AVC can be generalized to vector sizes which are sub-multiples of the warp size, to offer a different tradeoff of accuracy to compression-ratio.

Persistent control data, local computation data Computation data in *Hotspot*, *Heartwall* and *Needleman-Wunsch* are live only at local scope. The inner loop body consists in one or a few memory loads which are followed by computations, then by one or more memory stores. All variables that persist across inner loop iterations are made of control data.

As an extreme point, *Needleman-Wunsch* kernels suffer from significant register pressure, and require 47 registers each on Fermi in 64-bit mode. Surprisingly, we find that 3 registers are enough to hold all generic vectors in each thread's private working set.

Address space expansion We observe on Figure 1(a) that kernels suffer from higher register pressure when compiled for Fermi than when compiled for Tesla. We found the register pressure increase is primarily caused by 64-bit addressing, which requires all pointers to occupy two 32-bit registers. Compared to Fermi in 32-bit mode, *Needleman-Wunsch* kernels require 20 additional registers each and *Heartwall* requires 12 additional registers when compiled in 64-bit mode. Aside from pointer size, we found that instruction set differences bear little impact on register allocation. The 32 high-order bits of 64-bit virtual addresses are highly likely to be constant in current applications. Thus, we expect the additional registers to be mostly made of uniform vectors.

The analysis conducted in this section shows that private memory traffic presents a very high amount of redundancy, and hints that private memory is highly compressible. On average, affine vectors account for 66% of memory traffic to the private memory space on unaltered benchmarks. When benchmark parameters are adjusted to better fit the micro-architecture model, this ratio reaches



(a) *CFD* (Compute Flux) (b) *Needleman-Wunsch*

Figure 2: Private memory traffic profiles of selected representative kernels. Private memory transactions are classified among affine and generic vectors and are normalized by the execution rate. The *Global* column offers the amount of memory traffic to the global memory space as reference, without distinction between affine and generic data.

80%.

4 Affine vector cache

We describe in this section the organization and the workings of the AVC.

4.1 Cache organization

The AVC complements the L1 cache of each SM. Figure 3 outlines the new cache hierarchy. Stores are forwarded either to the AVC or the L1 cache depending on whether the affine encoding succeeds. On loads, both cache tags are looked up. If a match is found in the AVC tags, the base b and stride s are retrieved from the AVC data array, then decoded to an expanded vector $\mathbf{d}$. Data fills and write-backs with the lower levels of the memory hierarchy are handled in a symmetric way.

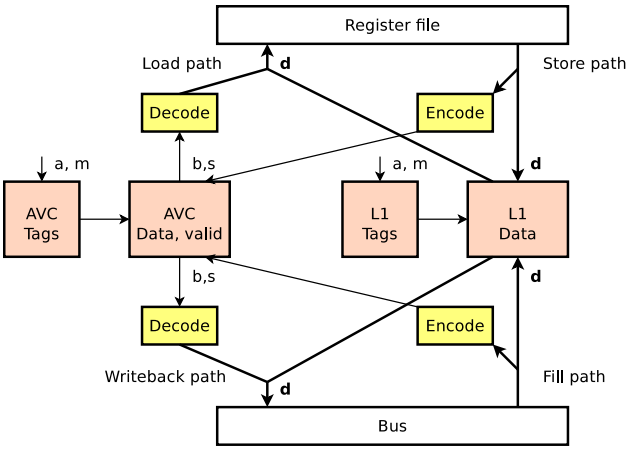


Figure 3: Overall architecture of the AVC and L1 cache.

4.2 Fine-grained coherence data

When a store instruction is predicated by a partially-set activity mask, it only writes to a subset of the destination cache block. Conventional caches implement partial writes to a cache block by ensuring the cache has ownership over the cache block considered and modifying the cached copy. However, random writes are generally not possible inside an affine vector. A read-modify-write implementation would produce a partially-affine vector, which would be considered as generic and excluded from the AVC. We found the read-modify-write solution essentially nullifies the benefits of the AVC on kernels with divergent control-flow such as LUD, and did not retain this option.

We propose instead to generalize predication to all memory transactions, and maintain cache coherence data at word-granularity (Figure 4). Read and write transactions are accompanied by the access mask m. The AVC is essentially equivalent to a sectored cache [25] with word-sized cache blocks.[2] Each cache line is subdivided into 16 vectors, and each vector is made of 32 cache blocks. Each block corresponds to one memory word and has one associated valid bit valid_i. Valid bits of a vector make a valid mask valid. The data itself is stored in the compact form (b, s), where $\log_2(s)$ is stored on 3 bits with a special value for $s = 0$. and associated with each vector. Valid bit masks valid are stored inside the data array rather than the tag array, in order to maintain the same tag lookup latency as the baseline cache architecture.

[2]We name *line* the part of the cache associated with a tag and *block* the part associated with a valid bit.

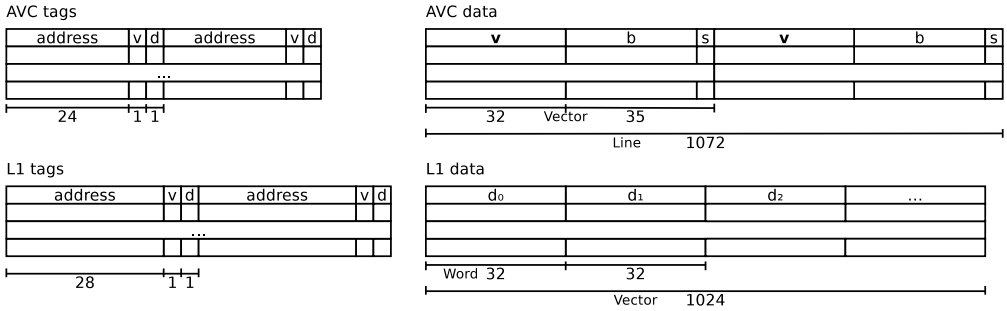


Figure 4: Contents of the tags and arrays of the AVC and L1 cache.

4.3 Cache access protocol

Unlike a conventional sectored cache, the AVC supports vectorized transactions, which access several blocks concurrently inside a vector. A vector is partially valid when its companion valid mask is partially set. Thus, a vectorized transaction can lead to both a (partial) cache hit and a (partial) cache miss. Figure 5 gives an example of a partial read miss. Partial misses are handled by first servicing the hit request, then replaying the instruction on behalf of the threads that encountered a miss. This partial-replay mechanism is similar to the one used to deal with bank conflicts on SIMD architectures with scatter-gather support [26].

The key idea behind mask-based coherency control is that the scope of variables is correlated with the structure of control flow in structured programming languages. A variable which is assigned at a given nesting level is likely to be used at the same or deeper nesting levels. This means that the activity mask of a load is likely to be a subset of the mask of the preceding store, and partial AVC hits are expected to be infrequent.

Cache coherency Every valid bit in the AVC indicates that the corresponding copy in the L1 is invalid. When a vector is evicted from the AVC, any partial copy present in the L1 must also be invalidated. Partial vectors from the AVC and L1 cache are merged together when written back, according to the valid mask of the vector from the AVC.

Conflicts A conflict happens when two partial vectors with different base and stride are mapped to the same vector in the AVC. The two following situations lead to conflicts:

1. Affine data written to the AVC conflict with the current AVC contents.

2. An AVC vector fill initiated by a read transaction conflicts with cached data.

Conflicts cause the older partial vector to be evicted from the AVC and replaced by the younger one. Any L1 partial copy is also invalidated and merged on the memory side.

Algorithm 1 details the processing of read transactions. In our formalism, the AVC Lookup operation returns a flag hit which is set on a line hit, in addition to the valid mask valid, base b and stride s. Likewise, the Encode operation returns a flag affine which is set when the input was successfully encoded into an affine vector (b, s).

The handling of write operations is summarized in Algorithm 2. Writes may update an affine vector by writing additional components without altering the base and stride. This situation happens in the case of extraneous thread divergence, as the instructions are executed twice with non-overlapping activity masks [13]. This case is handled using a comparator on the compact encoding. It can be considered as a form of silent store [18], although it does affect the valid bit mask.

Algorithm 1 Processing of a read request $(a, \mathbf{m})$

$(\text{hit}_a, \text{valid}, b, s) \leftarrow \text{AVC.Lookup}(a, \mathbf{m})$
$(\text{hit}_g, \mathbf{d}) \leftarrow \text{L1.Lookup}(a)$
if hit_a **and** $(\mathbf{m} \wedge \text{valid}) \neq 0$ **then**
  $d \leftarrow \text{Decode}(b, s)$ {AVC hit}
  if $\mathbf{m} \wedge \neg\text{valid} \neq 0$ **then**
    $\text{Replay}(\mathbf{m} \wedge \neg\text{valid})$ {Partial miss}
  end if
else if not hit_g **then**
  $\mathbf{d} \leftarrow \text{Memory.Read}(a)$ {Miss, fill}
  $(b, s, \text{affine}) \leftarrow \text{Encode}(\mathbf{d}, \mathbf{m})$
  if affine **then**
    $\text{AVC.Replace}(a, b, s, \mathbf{m})$
  else
    $\text{L1.Replace}(a, \mathbf{d})$
  end if
end if

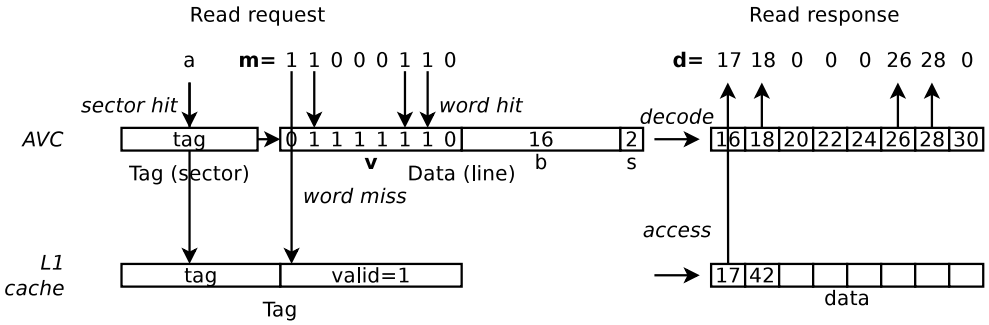


Figure 5: Access sequence on a partial read miss.

Algorithm 2 Processing of a write request $(a, \mathbf{m}, \mathbf{d})$

$(b, s, \text{affine}) \leftarrow \text{Encode}(\mathbf{d}, \mathbf{m})$
if affine **then**
  $(\text{hit}, \text{valid}, b', s') \leftarrow \text{AVC.Lookup}(a, \mathbf{m})$
  if hit **and** $(\text{valid} \wedge \neg\mathbf{m}) \neq 0$ **and** $(b, s) \neq (b', s')$ **then**
    $\text{L1.Invalidate}(a, \mathbf{m})$ {Conflict: writeback dirty data}
    $\text{AVC.Invalidate}(a, b, s, \mathbf{m})$
  end if
  $\text{AVC.Write}(\text{a}, \mathbf{m}, \text{b}, \text{s})$
else
  $\text{L1.Write}(a, \mathbf{m}, d)$
end if

5 Evaluation

We evaluate the effect of the AVC on memory pressure, examine the hardware implementation and overhead and study the impact on performance and board-level power.

5.1 Memory pressure analysis

We carry a sensitivity analysis across the 4 cache configurations listed on Table 2. All caches use a tree-based pseudo LRU replacement policy, and the other parameters are identical to those listed table 1. To allow a fair comparison, configurations are selected such that the area and power requirements of tag arrays and data arrays are virtually identical across all configurations. The remaining differences will be quantified in section 5.2.

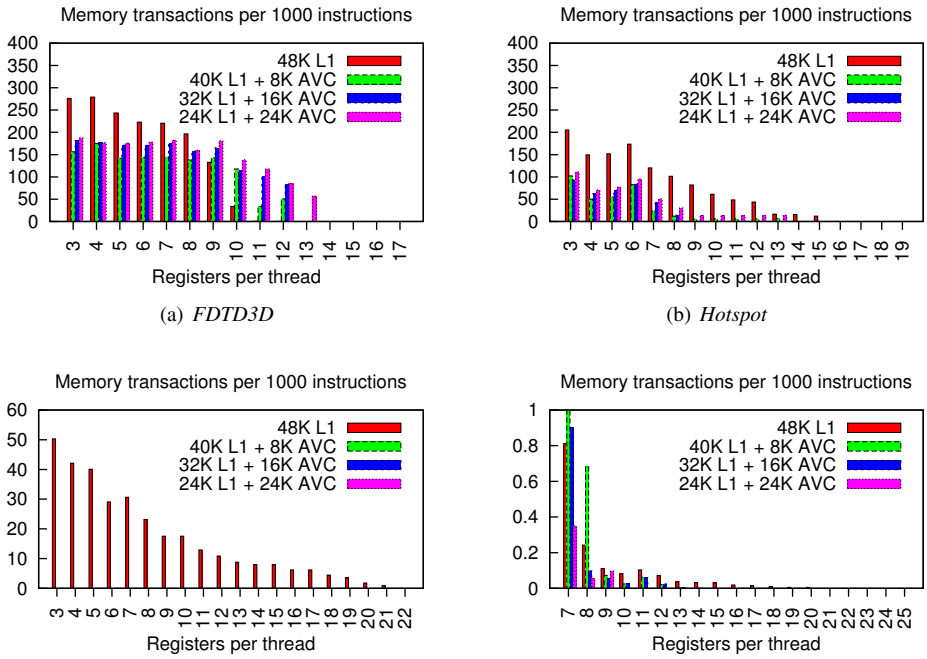
The primary objective of the AVC is to relax pressure on memory bandwidth. Accordingly, the metric used for evaluation is the memory traffic to lower-level caches generated by fills and writebacks of the AVC and L1 cache (referred as *memory traffic* in the rest of this section). We define r_{cache} as the smallest register count such that memory traffic is below one access every 10 000 instructions, at which point we consider it as negligible. The values of r_{cache} are given on table 3 for each configuration.

Table 2: Cache configurations studied

Configuration	L1 cache	AVC
Baseline	48K, 6-way	–
1	40K, 5-way	8K, direct-mapped
2	32K, 4-way	16K, 2-way
3	24K, 3-way	24K, 3-way

Overall, the cache pressure reductions gained from the AVC are consistent with the amount of affine vectors in private memory traffic observed in section 3. On average over all applications, the combined cache can fit respectively 3.7, 4.3 and 3.7 additional registers under configuration 1, 2 and 3 compared to the baseline. This corresponds to respective increases of 50%, 59% and 50% in total cache capacity. Profiles of selected kernels are shown in figure 6. The private memory traffic is zero for *Backprop2*, not shown, in any configuration, since its working set fits entirely in the smallest L1 cache.

Bandwidth amplification We observe in figure 6(d) that bandwidth pressure in *Heartwall* increases under configurations 1 and 2 with 7 registers. This seemingly surprising fact (*Heartwall*'s private memory traffic is 100% affine from Figure 1(b)) is explained by conflict misses in the lowest-associative AVC configurations. Conflict misses are especially detrimental to performance in a compressed sectored cache, as they require decoding and eviction of whole cache lines, resulting in a bandwidth amplification effect on the memory side. Possible solutions include i) increasing the associativity of the AVC, ii) adding an affine-aware victim cache, iii) combining the AVC and L1 tags in a skewed set-associative cache [27]. Alternatively, extending the affine compression to lower cache levels and memory would eliminate the bandwidth



(a) *FDTD3D* (b) *Hotspot* (c) *Needleman-Wunsch* (d) *Heartwall*

Figure 6: Classification of cumulative fills and writebacks on private memory.

amplification effect altogether.

5.2 Hardware implementation

In addition to the conventional cache hardware, the AVC design includes encoding and decoding units.

Encoding unit The main difficulty in encoding arises from the activity mask. The stride computation has to ignore the inactive components of the input vector, which are not known statically. This makes the computation challenging to implement efficiently in hardware.

However, we found that activity masks are *dense*, such that vectors have consecutive active components. Affine vectors with non-dense activity masks that have more than one bit set represent less than 0.1% of all affine vectors in our experiments. The proposed encoding unit estimates the stride by performing pair-wise *xor* operations between vector elements: $s = d_{2i} \oplus d_{2i+1}$, where $v_{2i} = v_{2i+1} = 1$. The *xor* comes from a subtracter in which most of the logic was optimized away by taking advantage of the alignment constraint of affine vectors. A priority encoder then selects the stride corresponding to the first valid pair of elements. The base b is computed from the estimated stride s and the value d_i of the first active vector component i as $b = d_i - i \cdot s$. The candidate pair (b, s) is then run through the decoding unit and compared with the source vector under the activity mask.

Decoding unit The decoding unit computes $d_i = b + i \cdot s$, which can be implemented by broadcasting b and s to all lanes, left-shifting each lane ID i according to s, then summing the result with b on each lane.

Overhead estimation The decoding unit is made of a row of thirty-two 5-by-3-bits shifters and 32-bit adders. A 32-bit adder heavily optimized for power can be implemented with a latency of 500 ps and dynamic energy of 1 pJ on a 90 nm process [24]. Accordingly, we conservatively bound the latency of the decoding unit to 1 ns, including routing. It fits within the timing budget of a single clock cycle given frequency targets of GPU architectures. From the same sources, the dynamic energy of both the encoding and decoding unit is estimated under 100 pJ. The AVC valid bits need to be accessed in addition to the tags and L1 data arrays. Estimations using CACTI [29] point to an access energy of 10 pJ for a 64-bit wide 16KB SRAM in 45 nm. The overall dynamic energy overhead is well under 1 nJ per access under the most conservative assumptions.

Table 3: Tolerable register pressure for various cache configurations. * Indicates the minimal allocatable register count was reached.

Benchmark	r_{base}	r_{cache}			
		B	1	2	3
Backprop2	12	7*	7*	7*	7*
CFD2	35	30	27	26	27
FDTD3D	22	13	14	14	14
SRAD2	12	6	7	6	7
SRAD1	10	8	8	8	8
Backprop1	13	7*	7*	7*	7*
Hotspot	27	17	14	9	14
FastWalshTransform2	8	5	4	4	4
BlackScholes	17	9	5	5	5
FastWalshTransform3	11	7	4	3*	4
Heartwall	25	10	9	9	8
LUD3	16	10	6	6	6
Needleman-Wunsch1	24	22	3*	3*	3*
BinomialOptions	14	7	6*	6*	6*

Extrapolating from a published ALU design [12] and accounting for process scaling, our conservative back-of-the-envelope area estimate is 80 μm^2 for both the encoding and decoding units in a 40 nm process. The additional 5 Kbits of SRAM cells account for another 1000 μm^2 according to detailed CACTI results [29]. A Fermi SM built on a 40 nm process is 15.6 mm^2 by our measurements on a die micrograph, so the area overhead is in the order of 10^{-4}. Increase in static power is expected to be commensurate with the difference in area.

5.3 Power-performance analysis

We simulate a single-SM GPU modeled after the Fermi micro-architecture at cycle level, based on available information [21, 31]. Dynamic power is estimated using a trace-driven linear model calibrated from micro-benchmark results [5], following a methodology inspired by Hong and Kim [16]. Relative power of Needleman-Wunsch, Backprop, Hotspot and SRAD are within 20% of experimental power measurements [3].

We focus on the 32K L1 + 16K AVC configuration, for $r = 0.5\, r_{\text{base}}$. The overhead considered for the AVC configuration is 1 extra cycle and (conservatively) 1 nJ per access. We assume that no decoding bypass exists, so the latency overhead applies to both affine and non-affine private memory loads. Figure 7(a) shows the performance of the AVC configuration normalized by baseline performance.

Kernels bound by memory throughput like FDTD3D, CFD and FastWalshTransform show some performance improvements. Performance of other kernels such as Needleman-Wunsch is not significantly affected, as multi-threading is able to hide the extra memory latency. However, memory accesses have a notable impact in terms of energy consumption, as found in figure 7(b). Heartwall encounters a slowdown of 0.3% with the AVC, due to the bandwidth amplification effect described section 5. Overall, performance improves by 5.7%, while energy is reduced by 11%.

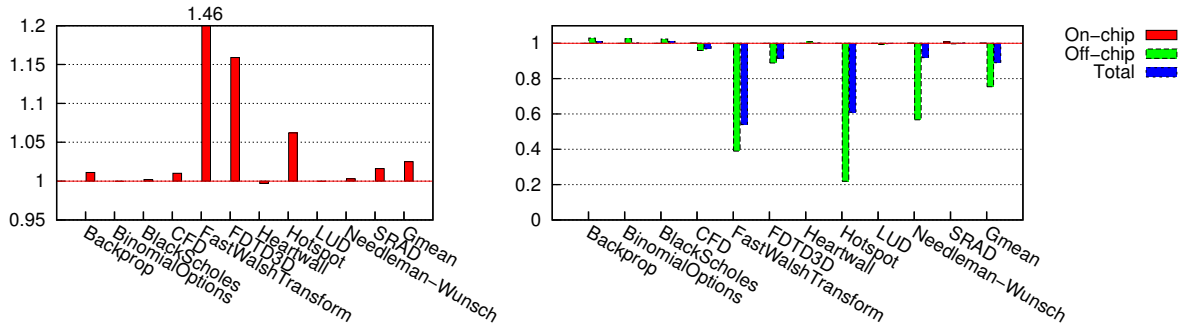
Our baseline architecture has no L2 data cache, although some GPUs do [21]. However, accessing the distributed L2 cache through the on-chip interconnect also consumes significant energy. The L2 aggregate capacity is also lower than the aggregate L1 capacity.

6 Related work

Cache compression that exploits regular patterns, such as Frequent Value Cache [32], Frequent Pattern Cache [1] or Zero-content Augmented Caches [10] has been studied for single-processor and multi-core architectures. This area of research has focused on compressing patterns that can appear out of context, such as small integers or zeroes, or leveraging correlations between different variables of the same thread. By contrast, the AVC takes advantage of correlations between several instances of the same program variable owned by different threads. Prior work uses this inter-thread data correlation effect to reduce the power consumption of the register files and datapaths on GPUs [6], or to improve the throughput of SMT CPUs [8]. We exploit the same effect in a different way by targeting data caches. The Mergeable cache architecture proposed by Biswas et al. merges identical cache lines belonging to different processes [2]. It targets coarse-grained multi-process workloads, while the AVC addresses fine-grained GPU multi-threading.

As memory bandwidth is a major concern in GPU design, a rich panel of memory compression techniques have been proposed or implemented for GPUs, notably lossless compression of the frame-buffer for power saving purposes [17, 28]. A memory channel compression scheme that may be in use in some contemporary GPUs identifies vectors that contain only 1, 2 or 4 distinct scalars to save memory bandwidth [11]. These techniques target off-chip memory rather than caches, so they are orthogonal to the AVC.

Intel's Larrabee project [26] and AMD's Graphics Core Next (GCN) project [9] describe graphics-oriented architectures based on explicit-SIMD cores. Both architectures



(a) Performance: speedup from baseline

(b) Energy: relative to baseline

Figure 7: Performance and energy consumption of the 32K L1+16K AVC configuration relative to the baseline 48K L1 configuration, for $r = 0.5r_{\text{base}}$.

provide a scalar unit and a scalar register file, acting in complement with the SIMD unit and register file. GCN also envisions a dedicated cache for scalar data. While the scalar cache serves a similar purpose as the AVC, it only operates on data that was marked explicitly as scalar by the compiler.

7 Conclusion

High-throughput implicit-SIMD architectures maintain a large number of thread contexts in flight, placing tremendous pressure on register files and local memory. However, we show that significant correlations exist between instances of long-lived thread-private variables, leading to affine vectors in memory. Under register pressure constraints allowing half of thread-private data to fit in the register file, we observe on average that 67% of the remaining half is composed of affine vectors. By exploiting this source of redundancy, an AVC replacing 33% of the L1 cache increases the total effective cache capacity by 59%, saving 11% of energy. As a microarchitectural feature, the AVC has no impact on the instruction set and software stack, and preserves the implicit-SIMD programming model. It can opportunistically detect affine vectors in complex and dynamic applications and it tolerates control-flow divergence.

The insights gained throughout our study suggest that two directions could be followed to apply inter-thread data correlations to parallel architectures. First, the compiler could be made aware of the AVC by prioritizing spills of variables that can be identified as affine through a static divergence analysis [7]. Second, affine compression could be extended upwards, to the register file, and downwards in the memory hierarchy, to lower-level caches and memory. In addition to the further compression benefits it would enable, it would eliminate the overhead of encoding and decoding affine data.

While on-line hardware data compression has been traditionally centered on improving the usable capacity of memories, we expect the focus to shift to link compression techniques that aim at reducing the amount of on-chip and off-chip communication. Data compression on memory traffic should become increasingly relevant with the advent of highly-parallel, highly-integrated architectures, as power and bandwidth become scarce resources.

References

[1] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. *SIGARCH Comp. Arch. News*, 32:212–223, March 2004.

[2] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-execution: multicore caching for data-similar executions. In *ISCA 36*, pages 164–173, 2009.

[3] 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 *WCS 2009*, pages 44–54, 2009.

[4] C. Collange, M. Daumas, D. Defour, and D. Parello. Barra: a parallel functional simulator for GPGPU. In *MASCOTS*, pages 351–360, 2010.

[5] C. Collange, D. Defour, and A. Tisserand. Power consumption of GPUs from a software perspective. In *ICCS 2009*, volume 5544 of *LNCS*, pages 922–931. Springer, 2009.

[6] C. Collange, D. Defour, and Y. Zhang. Dynamic detection of uniform and affine vectors in GPGPU

computations. In *Europar HPPC workshop*, volume LNCS 6043, pages 46–55, 2009.

[7] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira. Divergence analysis and optimizations. In *PACT*, october 2011.

[8] M. Dechene, E. Forbes, and E. Rotenberg. Multi-threaded instruction sharing. Technical report, North Carolina State University, 2010.

[9] E. Demers. Evolution of AMD's graphics core, and preview of Graphics Core Next. AMD Fusion Developer Summit keynote, june 2011.

[10] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *ICS'09*, pages 46–55, 2009.

[11] C. W. Everitt. Bandwidth compression for shader engine store operations. US Patent 7886116, assignee NVIDIA, February 2011.

[12] E. S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad. A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor. *IEEE JSSC*, 37(11):1433–1440, Nov 2002.

[13] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO '07*, pages 407–420, 2007.

[14] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53:58–66, November 2010.

[15] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *ISCA 38*, pages 235–246, 2011.

[16] S. Hong and H. Kim. An integrated GPU power and performance model. *SIGARCH Comput. Archit. News*, 38(3):280–289, 2010.

[17] Intel. *Intel OpenSource HD Graphics PRM Volume 3 Part 2: Display Registers – CPU Registers*, March 2010.

[18] K. M. Lepak, G. B. Bell, and M. H. Lipasti. Silent stores and store value locality. *IEEE Trans. Comput.*, 50:1174–1190, November 2001.

[19] J. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[20] J. Meng, J. Sheaffer, and K. Skadron. Exploiting inter-thread temporal locality for chip multithreading. In *IPDPS 2010*, april 2010.

[21] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, March 2010.

[22] NVIDIA. *NVIDIA CUDA Programming Guide, Version 4.0*, 2010.

[23] NVIDIA CUDA SDK, 2010. `http://www.nvidia.com/cuda/`.

[24] D. Patil, O. Azizi, M. Horowitz, R. Ho, and R. Ananthraman. Robust energy-efficient adder topologies. In *ARITH'18*, pages 16–28, 2007.

[25] S. Przybylski. The performance impact of block sizes and fetch strategies. *SIGARCH Comp. Arch. News*, 18:160–169, May 1990.

[26] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[27] A. Seznec. Concurrent support of multiple page sizes on a skewed associative TLB. *IEEE Trans. Comp.*, 53(7):924–927, july 2004.

[28] H. Shim, N. Chang, and M. Pedram. A compressed frame buffer to reduce display power consumption in mobile systems. In *ASP-DAC'04*, pages 818–823, 2004.

[29] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical report, HP Labs, 2008.

[30] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC'08*, pages 1–11, 2008.

[31] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPAS 2010*, 2010.

[32] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. *SIGPLAN Not.*, 35:150–159, November 2000.