Stash: Have Your Scratchpad and Cache It Too *

Rakesh Komuravelli† Matthew D. Sinclair† Johnathan Alsop† Muhammad Huzaifa†
Maria Kotsifakou† Prakalp Srivastava† Sarita V. Adve†‡ Vikram S. Adve†‡
† University of Illinois at Urbana-Champaign
‡ École Polytechnique Fédérale de Lausanne
hetero@cs.illinois.edu

Abstract

Heterogeneous systems employ specialization for energy efficiency. Since data movement is expected to be a dominant consumer of energy, these systems employ specialized memories (e.g., scratchpads and FIFOs) for better efficiency for targeted data. These memory structures, however, tend to exist in local address spaces, incurring significant performance and energy penalties due to inefficient data movement between the global and private spaces. We propose an efficient heterogeneous memory system where specialized memory components are tightly coupled in a unified and coherent address space. This paper applies these ideas to a system with CPUs and GPUs with scratchpads and caches.

We introduce a new memory organization, stash, that combines the benefits of caches and scratchpads without incurring their downsides. Like a scratchpad, the stash is directly addressed (without tags and TLB accesses) and provides compact storage. Like a cache, the stash is globally addressable and visible, providing implicit data movement and increased data reuse. We show that the stash provides better performance and energy than a cache and a scratchpad, while enabling new use cases for heterogeneous systems. For 4 microbenchmarks, which exploit new use cases (e.g., reuse across GPU compute kernels), compared to scratchpads and caches, the stash reduces execution cycles by an average of 27% and 13% respectively and energy by an average of 53% and 35%. For 7 current GPU applications, which are not designed to exploit the new features of the stash, compared to scratchpads and caches, the stash reduces cycles by 10% and 12% on average (max 22% and 31%) respectively, and energy by 16% and 32% on average (max 30% and 51%).

1. Introduction

Specialization is a natural path to energy efficiency. There has been a significant amount of work on compute specialization and it seems clear that future systems will support some collection of heterogeneous compute units (CUs). However, the memory hierarchy is expected to become a dominant consumer of energy as technology continues to scale [18, 22]. Thus, efficient data movement is essential for energy efficient heterogeneous systems.

To provide efficient data movement, modern heterogeneous systems use specialized memories; e.g., scratchpads and FIFOs. These memories often provide better efficiency for specific access patterns but they also tend to exist in disjoint, private address spaces. Transferring data to/from these private address spaces incurs inefficiencies that can negate the benefits of specialization. We propose an efficient heterogeneous memory system where specialized memory components are tightly coupled in a unified and coherent address space. Although industry has recently transitioned to more tightly coupled heterogeneous systems with a single unified address space and coherent caches [1, 3, 20], specialized components such as scratchpads are still within private, incoherent address spaces.

In this paper, we focus on heterogeneous systems with CPUs and GPUs where the GPUs access both coherent caches and private scratchpads. We propose a new memory organization, stash, that combines the performance and energy benefits of caches and scratchpads. Like a scratchpad, the stash provides compact storage and does not have conflict misses or overheads from tags and TLB accesses. Like a cache, the stash is globally addressable and visible, enabling implicit, on-demand, and coherent data movement and increasing data reuse. Table 1 compares caches and scratchpads (Section 2 discusses the stash).

1.1. Caches

Caches are a common memory organization in modern systems. Their software transparency makes them easy to program, but caches have several inefficiencies:

Indirect, hardware-managed addressing: Cache loads and stores specify addresses that hardware must translate to determine the physical location of the accessed data. This *indirect addressing* implies that each cache access (a hit or a miss) incurs (energy) overhead for TLB lookups and tag comparisons. Virtually tagged caches do not require TLB lookups on hits, but they incur additional overhead, including dealing with synonyms, page mapping and protection changes, and cache coherence [8]. Furthermore, the indirect, hardware-managed addressing also results in unpredictable hit rates due to cache conflicts, causing pathological performance (and energy) anomalies, a notorious problem for real-time systems.

Inefficient, cache line based storage: Caches store data at fixed cache line granularities which wastes SRAM space when a program does not access the entire cache line (e.g., when a program phase traverses an array of large objects but accesses only one field in each object).

*This work was supported in part by Intel Corporation through the Illinois/Intel Parallelism Center at Illinois, the National Science Foundation under grants CCF-1018796 and CCF-1302641, a Qualcomm Innovation Fellowship for Komuravelli and Sinclair, and by the Center for Future Architectures Research (C-FAR), one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

ISCA '15, June 13 - 17, 2015, Portland, OR, USA
 ISBN 978-1-4503-3402-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750374

Feature	Benefit	Cache	Scratchpad	Stash
Directly addressed	No address translation hardware access	✗ (if physically tagged)	✓	✓ (on hits)
	No tag access	✗	✓	✓
	No conflict misses	✗	✓	✓
Compact storage	Efficient use of SRAM storage	✗	✓	✓
Global addressing	Implicit data movement from/to structure No pollution of other memories On-demand loads into structures	✓	✗	✓
Global visibility	Lazy writebacks to global address space (AS) Reuse across compute kernels and application phases	✓	✗	✓

Table 1: Comparison of cache, scratchpad, and stash.

1.2. Scratchpads

Scratchpads (referred to as shared memory in CUDA) are local memories that are managed in software, either by the programmer or through compiler support. Unlike caches, scratchpads are directly addressed so they do not have overhead from TLB lookups or tag comparisons (this provides significant savings: 34% area and 40% power [7] or more [26]). Direct addressing also eliminates the pathologies of conflict misses and has a fixed access latency (100% hit rate). Scratchpads also provide compact storage since the software only brings useful data into the scratchpad. These features make scratchpads appealing, especially for real-time systems [5, 37, 38]. However, scratchpads also have some inefficiencies:

Not Globally Addressable: Scratchpads use a separate address space disjoint from the global address space, with no hardware mapping between the two. Thus, extra instructions must *explicitly* move data between the two spaces, incurring performance and energy overhead. Furthermore, in current systems the additional loads and stores typically move data via the core's L1 cache and its registers, *polluting* these resources and potentially evicting (spilling) useful data. Scratchpads also do not perform well for applications with *on-demand loads* because today's scratchpads preload all elements. In applications with control/data dependent accesses, only a few of the preloaded elements will be accessed.

Not Globally Visible: A scratchpad is visible only to its local CU. Thus dirty data in the scratchpad must be explicitly written back to the corresponding global address before it can be used by other CUs. Further, all global copies in the scratchpad must be freshly loaded if they could have been written by other CUs. In typical GPU programs, there is no data sharing within a kernel;[1] therefore, the writebacks must complete and the scratchpad space deallocated by the end of the kernel. With more fine-grained sharing, these actions are needed at the more frequent fine-grained synchronization phases. Thus, the scratchpads' lack of global visibility incurs potentially unnecessary *eager writebacks* and precludes *reuse* of data across GPU kernels and application synchronization phases.

1.2.1. Example and Usage Modes

Figure 1a shows how scratchpads are used. The code at the top reads one field, $fieldX$ (of potentially many), from an array of structs (AoS) data structure, *aosA*, into an explicit scratchpad copy. It performs some computations using the scratchpad copy and then writes back the result to *aosA*. The bottom of Figure 1a shows some of the corresponding steps in hardware. First, the hardware must explicitly copy $fieldX$ into the scratchpad. To achieve this, the application issues an *explicit load* of the corresponding global address to the L1 cache (event 1). On an L1 miss, the hardware brings $fieldX$'s cache line into the L1, *polluting* it as a result (events 2 and 3). Next, the hardware sends the data value from the L1 to the core's register (event 4). Finally, the application issues an *explicit store* instruction to write the value from the register into the corresponding scratchpad address (event 5). At this point, the scratchpad has a copy of $fieldX$ and the application can finally access the data (events 6 and 7). Once the application is done modifying the data, the dirty scratchpad data is *explicitly written back* to the global address space, requiring loads from the scratchpad and stores into the cache (not shown in the figure).

We refer to this scratchpad usage mode, where data is moved explicitly from/to the global space, as the *global-unmapped* mode. Scratchpads can also be used in *temporary* mode for private, temporary values. Temporary values do not require global address loads or writebacks because they are discarded after their use (they trigger only events 6 and 7).

1.3. Stash

To address the inefficiencies of caches and scratchpads we introduce a new memory organization, ***stash***, which combines the best properties of the cache and scratchpad organizations. Similar to a scratchpad, the stash is software managed, directly addressable, and can compactly map non-contiguous global memory elements to obtain the benefits of SoA or AoS format without any software changes or data transformations in memory. In addition, the stash also has a mapping between the global and local stash address spaces. Software provides this mapping; the stash hardware uses it whenever a translation between the two address spaces is required (e.g., misses, writebacks, and remote requests). The mapping allows the stash to avoid the explicit data movement of the scratchpad and instead implicitly move data between address spaces, like a cache. As a result, stash values are globally visible and replicable, enabled by relatively straightforward extensions to any underlying coherence protocol (see Section 4.3).

There has been a significant amount of prior work on optimizing the behavior of private memories such as scratchpads. This includes methods for directly transferring data from the memory to the scratchpad without polluting registers

[1] A kernel is the granularity at which the CPU invokes the GPU and it executes to completion on the GPU.

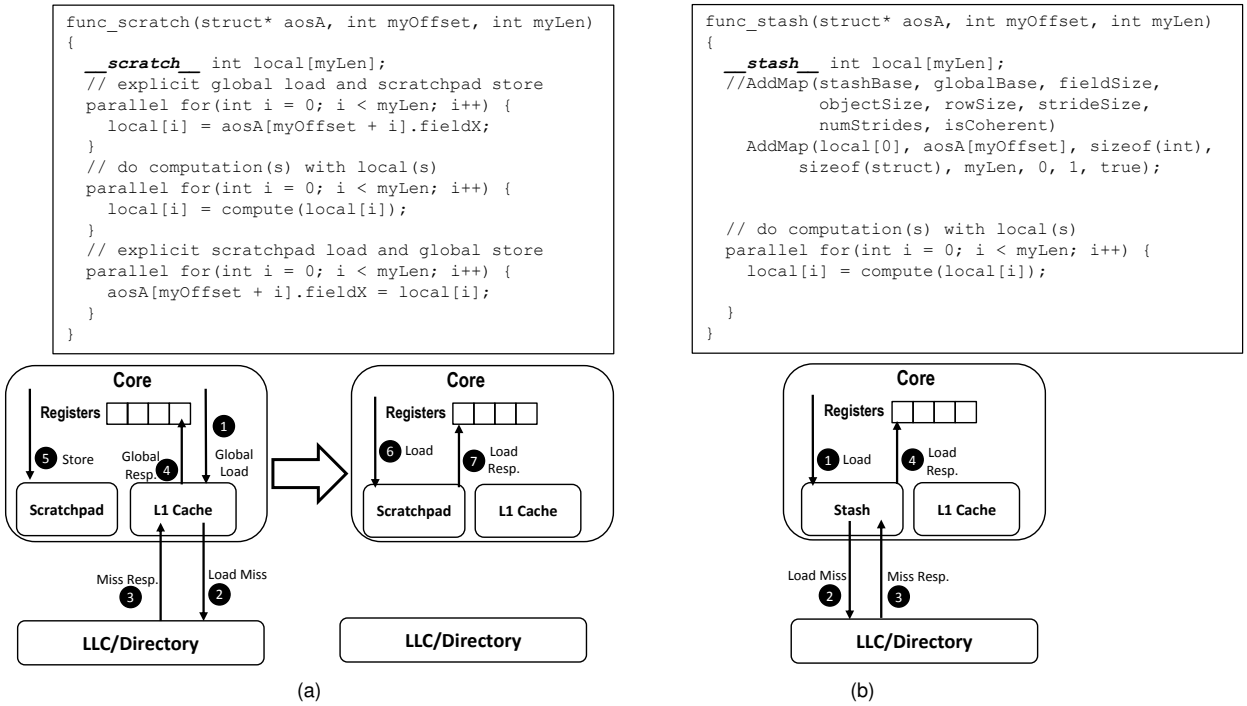


Figure 1: Codes and hardware events to copy data from the corresponding global address for (a) scratchpad and (b) stash (events to write data back to the global address are not shown).

or caches [4, 9, 21], changing the data layout for increased compaction [11, 13], removing tag checks for caches [33, 39], and virtualizing private memories [15, 16, 17, 28]. Each of these techniques mitigates some, but not all, of the inefficiencies of scratchpads or caches. While a quantitative comparison with all of the techniques is outside the scope of this work, we provide a detailed qualitative comparison for all (Section 7) and quantitatively compare our results to the closest technique to our work: scratchpad enhanced with a DMA engine.

The stash is a forward looking memory organization designed both to improve current applications and increase the use cases that can benefit from using scratchpads. Stash can be applied to any CU on an SoC including the CPU. In this paper we focus on GPUs which already have benchmarks that exploit scratchpads. However, because GPU scratchpads are not globally addressable or visible, current GPU applications that use the scratchpad cannot exploit these features. Thus, to demonstrate the additional benefits of the stash, we evaluate it for both current GPU applications and microbenchmarks designed to show future use cases.

For four microbenchmarks, which exploit new use cases, the stash outperforms all other configurations: compared to a scratchpad+cache hierarchy, a cache-only hierarchy, and scratchpads with DMA support, the stash reduces cycles by an average of 27%, 13%, and 14%, respectively, and reduces energy consumption by an average of 53%, 35%, and 32%, respectively. Furthermore, even for current GPU applications, which are not designed to exploit the new use cases, the stash outperforms both scratchpad+cache and cache-only configurations while providing comparable results to scratchpads with a DMA enhancement. Compared to a scratchpad+cache hierarchy and a cache-only hierarchy across seven modern GPU applications, the stash reduces cycles by 10% and 12% on average (max 22% and 31%), respectively, while decreasing energy by 16% and 32% on average (max 30% and 51%), respectively. These results demonstrate the ability of the stash to provide high performance and energy efficiency for modern applications while also demonstrating new use cases.

2. Stash Overview

The stash is a new SRAM organization that combines the advantages of scratchpads and caches, as summarized in Table 1. Stash has the following features:

Directly addressable: Like scratchpads, a stash is directly addressable and data in the stash is explicitly allocated by software (either the programmer or the compiler).

Compact storage: Since it is software managed, only data that is accessed is brought into the stash. Thus, like scratchpads, stash enjoys the benefit of a compact storage layout, and unlike caches, it only stores useful words from a cache line.

Physical to global address mapping: In addition to being able to generate a direct, physical stash address, software also specifies a mapping from a contiguous set of stash addresses to a (possibly non-contiguous) set of global addresses. Our architecture can map to a 1D or 2D, possibly strided, tile of global addresses.[2] Hardware maintains the mapping between the stash and global space.

Global visibility: Like a cache, stash data is globally visible through a coherence mechanism (described in Section 4.3).

[2]Our design can easily be extended to other access patterns.

A stash, therefore, does not need to eagerly writeback dirty data. Instead, data can be reused and lazily written back only when the stash space is needed for a new allocation (similar to cache replacements). If another CU needs the data, it will be forwarded through the coherence mechanism. In contrast, for scratchpads in current GPUs, data is written back to global memory (and flushed) at the end of a kernel, resulting in potentially unnecessary and bursty writebacks with no reuse across kernels.

The first time a load occurs to a newly mapped stash address, it implicitly copies the data from the mapped global space to the stash (analogous to a cache miss). Subsequent loads for that address immediately return the data from the stash (analogous to a cache hit, but with the energy benefits of direct addressing). Similarly, no explicit stores are needed to write back the stash data to its mapped global location. Thus, the stash enjoys all the benefits of direct addressing of a scratchpad on hits (which occur on all but the first access), without the overhead incurred by the additional loads and stores that scratchpads require for explicit data movement.

Figure 1b transforms the code from Figure 1a for a stash. The stash is directly addressable and stores data compactly just like a scratchpad but does not have any explicit instructions for moving data between the stash and the global address space. Instead, the stash has an *AddMap* call that specifies the mapping between the two address spaces (discussed further in Section 3). In hardware (bottom part of the figure), the first load to a stash location (event 1) implicitly triggers a global load (event 2) if the data is not already present in the stash. Once the load returns the desired data to the stash (event 3), it is sent to the core (event 4). Subsequent accesses directly return the stash data without consulting the global mapping.

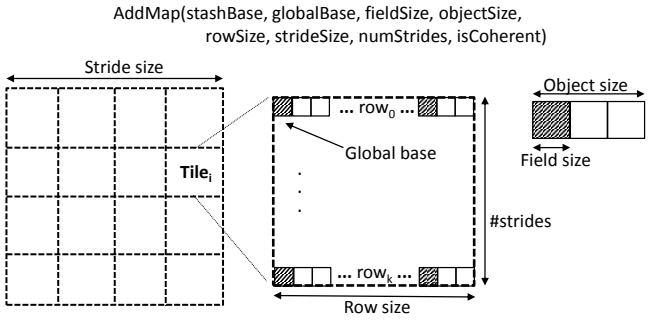
3. Stash Software Interface

We envision the programmer or the compiler will map a (possibly non-contiguous) part of the global address space to the stash. For example, programmers writing applications for today's GPU scratchpads already effectively provide such a mapping. There has also been prior work on compiler methods to automatically infer this information [5, 23, 30]. The mapping of the global address space to stash requires strictly less work compared to that of a scratchpad as it avoids the need for explicit loads and stores between the global and stash address spaces.

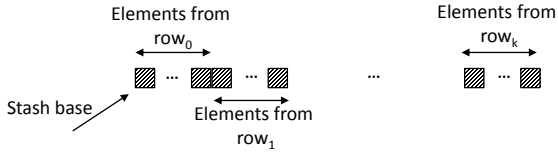
3.1. Specifying Stash-to-Global Mapping

The mapping between global and stash address spaces is specified using two intrinsic functions. The first intrinsic, *AddMap*, is called to communicate a new mapping to the hardware. We need an *AddMap* call for every data structure (a linear array or a 2D tile of an AoS structure) that is mapped to the stash.

Figure 1b shows an example usage of *AddMap* along with its definition. Figures 2a and 2b respectively show an example 2D tiled data structure in the global address space and the mapping of one field of one of the 2D tiles in the 1D stash address space. The first two parameters of *AddMap* specify the stash and global virtual base addresses for the given tile, as shown in Figure 2 (scratchpad base described in Section 4).



(a) 2D tiles in global address space (left), a 2D tile (center), and an object (right).



(b) 1D stash space with one of the fields of the AoS.

Figure 2: Mapping a global 2D AoS tile to a 1D stash address space.

Figure 2a also shows the various parameters used to describe the object and the tile. The field size and the object size provide information about the global data structure (field size = object size for scalar arrays). The next three parameters specify information about the tile in the global address space: the row size of the tile, global stride between two rows of the tile, and number of strides. Finally, *isCoherent* specifies the operation mode of the stash (discussed in Section 3.3). Figure 2b shows the 1D mapping of the desired individual fields from the 2D global AoS data structure.

The second intrinsic function, *ChgMap*, is used when there is a change in mapping or the operation mode of a set of global addresses mapped to the stash. *ChgMap* uses all of the *AddMap* parameters and adds a field to identify the map entry it needs to change (an index in a hardware table, discussed in Section 4.1.2).

3.2. Stash Load and Store Instructions

The load and store instructions for a stash access are similar to those for a scratchpad. On a hit, the stash just needs to know the requested address. On a miss, the stash also needs to know which stash-to-global mapping it needs to use (an index in a hardware table, discussed in Section 4.1.2, similar to that used by *ChgMap* above). This information can be encoded in the instruction in at least two different ways without requiring extensions to the ISA. CUDA, for example, has multiple address modes for LD/ST instructions - register, register-plus-offset, and immediate addressing. The register-based addressing schemes hold the stash (or scratchpad) address in the register specified by the *register* field. We can use the higher bits of the register for storing the map index (since a stash address does not need all the bits of the register). Alternatively, we can use the register-plus-offset addressing scheme, where register holds the stash address and $offset$ holds the map index (in CUDA, offset is currently ignored

when the local memory is configured as a scratchpad).

3.3. Usage Modes

Stash data can be used in four different modes:

Mapped Coherent: This mode is based on Figure 1b – it provides a stash-to-global mapping and the stash data is globally addressable and visible.

Mapped Non-coherent: This mode is similar to Mapped Coherent except that the stash data is not globally visible. As a result, any modifications to local stash data are not reflected in the global address space. We use the *isCoherent* bit to differentiate between the Mapped-Coherent and Mapped Non-Coherent usage modes.

Global-unmapped and Temporary: These two modes are based on the scratchpad modes described in Section 1.2.1. Unlike Mapped Coherent and Mapped Non-coherent, these modes do not need an AddMap call, since they do not have global mappings. These modes allow programs to fall back, if necessary, to how scratchpads are currently used and ensure we support all current scratchpad code in our system.

4. Stash Hardware Design

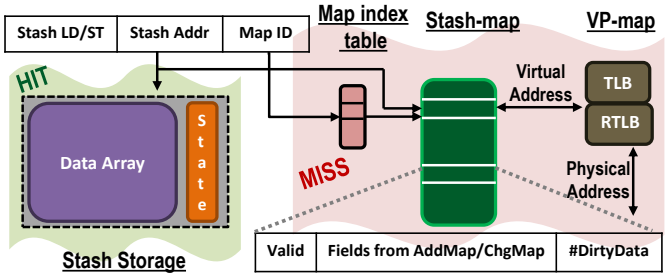


Figure 3: Stash hardware components.

This section describes the design of the stash hardware, which provides stash-to-global and global-to-stash address translations for misses, writebacks, and remote requests. For simplicity and without loss of generality we use NVIDIA's CUDA terminology to explain how the stash works in the context of a GPU. On current GPUs, every thread block gets a separate scratchpad allocation. The runtime translates the program-specified scratchpad address to a physical location in the thread block's allocated space. We assume a similar allocation and runtime address mapping mechanism for the stash.

4.1. Stash Components

Figure 3 shows the stash's four hardware components: (1) *stash storage*, (2) *map index table*, (3) *stash-map*, and (4) *VP-map*. This section describes each component. Section 4.2 describes how they enable the different stash operations.

4.1.1. Stash Storage

The stash storage component provides data storage for the stash. It also contains state bits to identify hits and misses (depending on the coherence protocol) and to aid writebacks (explained in Section 4.2).

4.1.2. Map Index Table

The per thread block map index table provides an index into the thread block's stash-map entries. Each *AddMap* allocates an entry into the map index table. Assuming a fixed ordering of *AddMap* calls, the compiler can determine which table entry corresponds to a mapping – it includes this entry's ID in future stash instructions corresponding to this mapping (using the format from Section 3). The size of the table is the maximum number of *AddMap*s allowed per thread block (our design allocates up to four entries per thread block). If the compiler runs out of entries, it cannot map any more data to the stash.

4.1.3. Stash-map

The stash-map contains an entry for each mapped stash data partition, shown in Figure 3. Each entry contains information to translate between the stash and global virtual address spaces, determined by the fields in *AddMap* or *ChgMap*. We precompute most of the information required for the translations at the *AddMap* and *ChgMap* call and do not need to store all the fields from these calls. With the precomputed information, only six arithmetic operations are required per miss (details in [24]). In addition to information for address translation, the stash-map entry has a *Valid* bit (denotes if the entry is valid) and a *#DirtyData* field used for writebacks.

We implemented the stash-map as a circular buffer with a tail pointer. We add and remove entries to the stash-map in the same order for easy management of stash-map's fixed capacity. The number of entries should be at least the maximum number of thread blocks a core can execute in parallel multiplied by the maximum number of *AddMap* calls allowed per thread block. We found that applications did not simultaneously use more than four map entries per thread block. Assuming up to eight thread blocks in parallel, 32 map entries are sufficient but we use 64 entries to allow additional lazy writebacks.

4.1.4. VP-map

We need the virtual-to-physical translations for each page of a stash-to-global mapping because every mapping can span multiple virtual pages. VP-map uses two structures for this purpose. The first structure, *TLB*, provides a virtual to physical translation for every mapped page, required for stash misses and writebacks. We can leverage the core's TLB for this purpose. For remote requests which come with a physical address, we need a reverse translation. The second structure, *RTLB*, provides the reverse translation and is implemented as a CAM over physical pages.[3]

Each entry in the VP-map has a pointer (not shown in Figure 3) to a stash-map entry that indicates the latest stash-map entry that requires the given translation. When a stash-map entry is replaced, any entries in the VP-map that have a pointer to that map entry are no longer needed. We remove these entries by walking the *RTLB* or *TLB*. By keeping each *RTLB* entry (and each *TLB* entry, if kept separate from system TLB) around until the last mapping that uses it is removed, we guarantee that we never miss in the *RTLB* (see Section 4.2). We assume that the number of virtual-to-physical translations required by all active mappings is less than the size of the VP-map (for the applications we studied, 64 entries were sufficient to support all the thread blocks running in parallel) and that the compiler or programmer is aware of this requirement.

[3]To reduce area, the TLB and RTLB can be merged into a single structure.

4.2. Operations

Next we describe how we implement the stash operations.

Hit: On a hit (determined by coherence bits as discussed in Section 4.3), the stash acts like a scratchpad, accessing only the storage component.

Miss: A miss needs to translate the stash address into a global physical address. Stash uses the index to the map index table provided by the instruction to determine its stash-map entry. Given the stash address and the stash base from the stash-map entry, we can calculate the stash offset. Using the stash offset and the other fields of the stash-map entry, we can calculate the virtual offset (details in [24]). Once we have the virtual offset, we add it to the virtual base of the stash-map entry to obtain the missing global virtual address. Finally, using the VP-map we can determine the corresponding physical address which is used to handle the miss.

Additionally, a miss must consider if the data it replaces needs to be written back and a store miss must perform some bookkeeping to facilitate a future writeback. We describe these actions next.

Lazy Writebacks: Stash writebacks (only for *Mapped Coherent entries*) happen lazily; i.e., the writebacks are triggered only when the space is needed by a future stash allocation similar to cache evictions.

On a store, we need to maintain the index of the current stash-map entry for a future writeback. Naively we could store the stash-map entry's index per word and write back each word as needed but this is not efficient. Instead, we store the index at a larger, chunked granularity, say 64B, and perform writebacks at this granularity.[4] To know when to update this per chunk stash-map index, we have a dirty bit per stash chunk. On a store miss, if this dirty bit is not set, we set it and update the stash-map index. We also update the *#DirtyData* counter of the stash-map entry to track the number of dirty stash chunks in the corresponding stash space. The per chunk dirty bits are unset when the thread block completes so that they are ready for use by a future thread block using the same stash chunk.

Later, if a new mapping needs to use the same stash location, the old dirty data needs to be written back. We (conceptually) use a writeback bit per chunk to indicate the need for a writeback (Section 4.4 discusses using bits already present for coherence states for this purpose). This bit is set for all the dirty stash chunks at the end of a thread block and checked on each access to trigger any needed writeback. To perform a writeback, we use the per chunk stash-map index to access the stash-map entry – similar to a miss, we determine the dirty data's physical address. We write back all dirty words in a chunk on a writeback (we leverage per word coherence state to determine the dirty words). On a writeback, the *#DirtyData* counter of the map entry is decremented and the writeback bit of the stash chunk is reset. When the *#DirtyData* counter reaches zero, the map entry is marked as invalid.

AddMap: An *AddMap* call advances the stash-map's tail, and sets the next entry of its thread block's map index table to point to this tail entry. It updates the stash-map tail entry with its parameters, does the needed precomputations for future address translations, and sets the *Valid* bit (Section 4.1.3). It also invalidates any entries from the VP-map that have the new stash-map tail as the back pointer.

For every virtual page mapped, an entry is added to the VP-map's *RTLB* (and the *TLB*, if maintained separately from the system's TLB). If the system TLB has the physical translation for this page, we populate the corresponding entries in VP-map (both in *RTLB* and *TLB*). If the translation does not exist in the TLB, the physical translation is acquired at the subsequent stash miss. For every virtual page in a new map entry, the stash-map pointer in the corresponding entries in VP-map is updated to point to the new map entry. In the unlikely scenario where the VP-map becomes full and has no more space for new entries, we evict subsequent stash-map entries (using the procedure described here) until there are enough VP-map entries available. The VP-map is sized to ensure that this is always possible. This process guarantees that we never miss in the RTLB for remote requests.

If the stash-map entry being replaced was previously valid (*Valid* bit set), then it indicates an old mapping has dirty data that has not yet been (lazily) written back. To ensure that the old mapping's data is written back before the entry is reused, we initiate its writebacks and block the core until they are done. Alternately, a scout pointer can stay a few entries ahead of the tail, triggering non-blocking writebacks for valid stash-map entries. This case is rare because usually a new mapping has already reclaimed the stash space held by the old mapping, writing back the old dirty data on replacement.

ChgMap: *ChgMap* updates an existing stash-map entry with new mapping information. If the mapping points to a new set of global addresses, we need to issue writebacks for any dirty data of the old mapping (only if *Mapped Coherent*) and mark all the remapped stash locations as invalid (using the coherence state bits in Section 4.3). A *ChgMap* can also change the usage mode for the same chunk of global addresses. If an entry is changed from coherent to non-coherent, then we need to issue writebacks for the old mapping because the old mapping's stores are globally visible. However, if the entry is modified from non-coherent to coherent, then we need to issue ownership/registration requests for all dirty words in the new mapping according to the coherence protocol (Section 4.3).

4.3. Coherence Protocol Extensions for Stash

All *Mapped Coherent* stash data must be kept coherent. We can extend any coherence protocol to provide this support (e.g., a traditional hardware coherence protocol such as MESI or a software-driven hardware coherence protocol like DeNovo [14, 35, 36]) as long as it supports the following three features:

1. Tracking at word granularity: Stash data must be tracked at word granularity because only useful words from a given cache line are brought into the stash.[5]

2. Merging partial cache lines: When the stash sends data to a cache (either as a writeback or a remote miss response), it

[4] This requires data structures in the memory to be aligned at the chosen chunk granularity.

[5] We can support byte granularity accesses if all (stash-allocated) bytes in a word are accessed by the same CU at the same time; i.e., there are no word-level data races. None of the benchmarks we studied have byte granularity accesses.

may send only part of a cache line. Thus the cache must be able to merge partial cache lines.

3. Map index for physical-to-stash mapping: When data is modified by a stash, the directory needs to record the modifying core (as usual) and also the stash-map index for that data (so a remote request can determine where to obtain the data).

It is unclear which is the best underlying coherence protocol to extend for the stash since protocols for tightly coupled heterogeneous systems are evolving rapidly and represent a moving target [19, 32]. Below we discuss how the above features can be incorporated within traditional directory-based hardware protocols and the recent DeNovo software-driven hardware protocol [14]. For our evaluations, without loss of generality, we choose the latter since it incurs lower overhead, is simpler, and is closer to current GPU memory coherence strategies (e.g., it relies on cache self-invalidations rather than writer-initiated invalidations and it does not use directories).

Traditional protocols: We can support the above features in a traditional single-writer directory protocol (e.g., MESI) with minimal overhead by retaining coherence state at line granularity, but adding a bit per word to indicate whether its up-to-date copy is present in a cache or a stash. Assuming a shared last level cache (LLC), when a directory receives a stash store miss request, it transitions to modified state for that line, sets the above bit in the requested word, and stores the stash-map index (obtained with the miss request) in the data field for the word at the LLC. This straightforward extension, however, is susceptible to false-sharing (similar to single-writer protocols for caches) and the stash may lose the predictability of a guaranteed hit after an initial load. To avoid false-sharing, we could use a sector-based cache with word-sized sectors, but this incurs heavy overhead with conventional hardware protocols (state bits and sharers list per word at the directory).

Sectored software-driven protocols: DeNovo [14, 35, 36] is a software-driven hardware coherence protocol that has word granularity sectors (coherence state is at word granularity, but tags are at conventional line granularity) and naturally does not suffer from false-sharing.

DeNovo exploits software-inserted self-invalidations at synchronization points to eliminate directory overhead for tracking the sharers list. Moreover, DeNovo requires fewer coherence state bits because it has no transient states. This combination allows DeNovo's total state bits overhead to be competitive with (and more scalable than) line-based MESI [14].

We extended the line-based DeNovo protocol from [14] (with line granularity tags and word granularity coherence, but we do not use DeNovo's regions). This protocol was originally proposed for multi-core CPUs and deterministic applications. Later versions of DeNovo support non-deterministic codes [36, 35], but our applications are deterministic. Although GPUs support non-determinism through operations such as atomics, these are typically resolved at the shared cache and are trivially coherent. We assume that software does not allow concurrent conflicting accesses to the same address in both cache and stash of a given core within a kernel. Our protocol requires the following extensions to support stash operations:

Stores: Similar to an MSI protocol, the DeNovo coherence protocol has three states. Stores miss when in *Shared* or *Invalid* state and hit when in *Registered* state. All store misses need to obtain registration (analogous to MESI's ownership) from the LLC/directory. In addition to registering the core ID at the directory, registration requests for words in the stash also need to include the corresponding stash-map index. As with the original DeNovo protocol, a system with a shared LLC stores the registered core ID in the data array of the LLC. The stash-map index can also be stored along with the core ID and its presence can be indicated using a bit in the same LLC data word. Thus, the LLC continues to incur no additional storage overhead for keeping track of remotely modified data.

Self-invalidations: At the end of a kernel we keep the data that is registered by the core (specified by the coherence state) but self-invalidate the rest of the entries to make the stash space ready for any future new allocations. In contrast, a scratchpad invalidates all entries (after explicitly writing the data back to the global address space).

Remote requests: Remote requests for stash that are redirected via the directory come with a physical address and a stash-map index (stored at the directory during the request for registration). Using the physical address, VP-map provides us with the corresponding virtual address. Using the stash-map index, we can obtain all the mapping information from the corresponding stash-map entry. We use the virtual base address from the entry and virtual address from the VP-map to calculate the virtual offset. Once we have the virtual offset, we use the map-entry's other fields to calculate the stash offset and add it to the stash base to get the stash address.

4.4. State Bits Overhead for Stash Storage

With the above DeNovo protocol, we compute the total state bits overhead in the stash storage component (Section 4.1) as follows. Each word (4B) needs 2 bits for the protocol state. Each stash chunk needs 6 bits for the stash-map index (assuming a 64 entry stash-map). Although Section 4.2 discussed a separate per-chunk writeback bit, since DeNovo has only 3 states, we can use the extra state in its two coherence bits to indicate the need for a writeback. Assuming 64B chunk granularity, all of the above sum to 39 bits per stash chunk, with a $\sim$8% overhead to the stash storage. Although this overhead might appear to be of the same magnitude as that of tags of conventional caches, only the two coherence state bits are accessed on hits (the common case).

4.5. Stash Optimization: Data Replication

It is possible for two allocations in stash space to be mapped to the same global address space. This can happen if the same read-only data is simultaneously mapped by several thread blocks in a CU, or if read-write data mapped in a previous kernel is mapped again in a later kernel on the same CU. By detecting this replication and copying replicated data between stash mappings, it is possible to avoid expensive misses.

To detect data replication, on an *AddMap* or *ChgMap* (infrequent operations), the map is searched for the virtual base address of the entry being added to the map. If there is a match, we compare the tile specific parameters to confirm if the two mappings indeed perfectly match. If there is a match, we set a bit, *reuseBit*, and add a pointer to the old mapping in the new

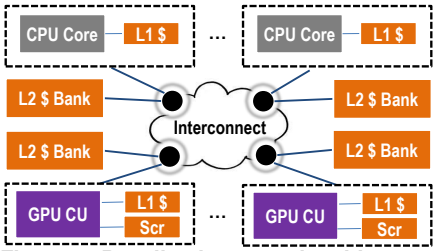


Figure 4: Baseline integrated architecture.

map entry. On a load miss, if the *reuseBit* is set, we first check the corresponding stash location of the old mapping and copy the value over if present. If not, we issue a miss to the registry.

If the new map entry is non-coherent and both the old and new map entries are for the same allocation in the stash, we need to write back the old dirty data. This is because any new updates to the same stash location should not be globally visible. Instead, if the new map entry is coherent and both the old and new map entries are for different allocations in the stash, we need to send new registration requests for the new map entry because we need to update the directory that the stash allocation corresponding to the new map entry holds the latest up-to-date copy.

5. Methodology

5.1. Baseline Heterogeneous Architecture

Figure 4 shows our baseline heterogeneous architecture, a tightly integrated CPU-GPU system with a unified shared memory address space and coherent caches. The system is composed of multiple CPU and GPU cores, which are connected via an interconnection network. Each GPU Compute Unit (CU), which is analogous to an NVIDIA SM, has a separate node on the network. All CPU and GPU cores have an attached block of SRAM. For CPU cores, this is an L1 cache, while for GPU cores, it is divided into an L1 cache and a scratchpad. Each node also has a bank of the L2 cache, which is shared by all CPU and GPU cores. The stash is located at the same level as the GPU L1 caches and both the cache and stash write their data to the backing L2 cache bank. All L1 caches use a writeback policy and the DeNovo coherence protocol. For a detailed justification of these design choices see [24].

5.2. Simulation Environment

To model a tightly coupled memory system, we created an integrated CPU-GPU simulator. We used the Simics full-system functional simulator to model the CPUs, the Wisconsin GEMS memory timing simulator [29], and GPGPU-Sim v3.2.1 [6] to model the GPU. We use Garnet [2] to model a 4x4 mesh interconnect that has a GPU CU or a CPU core at each node. We use CUDA 3.1 [31] for the GPU kernels in the applications since this is the latest version of CUDA that is fully supported in GPGPU-Sim. Table 2 summarizes the common key parameters of our simulated systems. Our GPU is similar to an NVIDIA GTX 480.

We extended GPUWattch [25] to measure the energy of the GPU CUs and the memory hierarchy including all stash components. To model the stash storage we extended GPUWattch's scratchpad model by adding additional state bits. We model the stash-map as an SRAM structure and the VP-map as a CAM unit. Finally, we model the operations for the address translations by adding an ALU for each operation using an in-built ALU model in GPUWattch. The sizes for these hardware components are listed in Table 2.

CPU Parameters	
Frequency	2 GHz
Cores (microbenchmarks, apps)	15, 1
GPU Parameters	
Frequency	700 MHz
CUs (microbenchmarks, apps)	1, 15
Scratchpad/Stash Size	16 KB
Number of Banks in Stash/Scratchpad	32
Memory Hierarchy Parameters	
TLB & RTLB (VP-map)	64 entries each
Stash-map	64 entries
Stash address translation	10 cycles
L1 and Stash hit latency	1 cycle
Remote L1 and Stash hit latency	35−83 cycles
L1 Size (8 banks, 8-way assoc.)	32 KB
L2 Size (16 banks, NUCA)	4 MB
L2 hit latency	29−61 cycles
Memory latency	197−261 cycles

Table 2: Parameters of the simulated heterogeneous system.

For our NoC energy measurements we use McPAT v.1.1 [27].[6] We do not measure the CPU core or the CPU L1 cache energy as our proposed stash design is implemented on the GPU. But we do measure the network traffic traveling to and from the CPU in order to capture any network traffic variations caused by the stash.

5.3. Simulated Memory Configurations

Our baseline architecture has scratchpads as described in Section 5.1. To evaluate the stash, we replaced the scratchpads with stashes, following the design in Section 4 (we used the DeNovo protocol and included the data replication optimization). We also compare stash to a cache-only configuration (i.e., all data allocated as global and accessed from the cache).

Additionally, we compare the stash to scratchpads enhanced with a DMA engine. Our DMA implementation is based on the D^{2}MA design [21]. D^{2}MA provides DMA capability for scratchpad loads on discrete GPUs and supports strided DMA mappings. D^{2}MA adds special load instructions and a hardware prefetch engine to preload all scratchpad words. Unlike D^{2}MA, our implementation blocks memory requests at a core granularity instead of a warp granularity, supports DMAs for stores in addition to loads, and runs on a tightly-coupled system. We conservatively do not charge additional energy for the DMA engine that issues the requests.

Overall we evaluate the following configurations:

1. *Scratch*: 16 KB Scratchpad + 32 KB L1 Cache. All memory accesses use the type specified by the original application.
2. *ScratchG*: *Scratch* with all global accesses converted to scratchpad accesses.
3. *ScratchGD*: *ScratchG* configuration with DMA support.

[6] We use McPAT's NoC model instead of GPUWattch's because our tightly coupled system more closely resembles a multi-core system's NoC.

4. *Cache*: 32 KB L1 Cache with all scratchpad accesses in the original application converted to global accesses.[7]
5. *Stash*: 16 KB Stash + 32 KB L1 Cache. The scratchpad accesses in the *Scratch* configuration were converted to stash accesses.
6. *StashG*: *Stash* with all global accesses converted to stash accesses.

5.4. Workloads

We present results for a set of benchmark applications to evaluate the effectiveness of the stash design on existing code. However, these existing applications are tuned for execution on a GPU with current scratchpad designs that do not efficiently support data reuse, control/data dependent memory accesses, and accessing specific fields from an AoS format. As a result, modern GPU applications typically do not use these features. However stash is a forward looking memory organization designed both to improve current applications and increase the use cases that can benefit from using scratchpads. Thus, to demonstrate the benefits of the stash, we also evaluate it for microbenchmarks designed to show future use cases.

5.4.1. Microbenchmarks

We evaluate four microbenchmarks: Implicit, Pollution, On-demand, and Reuse. Each microbenchmark is designed to emphasize a different benefit of the stash design. All four microbenchmarks use an array of elements in AoS format; each element in the array is a struct with multiple fields. The GPU kernels access a subset of the structure's fields; the same fields are subsequently accessed by the CPU to demonstrate how the CPU cores and GPU CUs communicate data that is mapped to the stash. We use a single GPU CU for all microbenchmarks. We parallelize the CPU code across 15 CPU cores to prevent the CPU accesses from dominating execution time. The details of each microbenchmark are discussed below.

Implicit highlights the benefits of the stash's implicit loads and lazy writebacks as highlighted in Table 1. In this microbenchmark, the stash maps one field from each element in an array of structures. The GPU kernel updates this field from each array element. The CPUs then access this updated data.

Pollution highlights the ability of the stash to avoid cache pollution through its use of implicit loads that bypass the cache. Pollution's kernel reads and writes one field in two AoS arrays *A* and *B*; *A* is mapped to the stash or scratchpad while *B* uses the cache. *A* is sized to prevent reuse in the stash in order to demonstrate the benefits the stash obtains by not polluting the cache. *B* can fit inside the cache only without pollution from *A*. Both stash and DMA achieve reuse of *B* in the cache because they do not pollute the cache with explicit loads and stores.

On-demand highlights the on-demand nature of stash data transfer and is representative of an application with fine-grained sharing or irregular accesses. The On-demand kernel reads and writes only one element out of 32, based on a runtime condition. Scratchpad configurations (including ScratchGD) must conservatively load and store every element that may be accessed. Cache and stash, however, can identify a miss and generate a memory request only when necessary.

[7] Because the *Cache* configuration has 16 KB less SRAM than other configurations, we also examined a 64 KB cache (GEMS only allows power-of-2 caches). The 64 KB cache was better than the 32 KB cache but had worse execution time and energy consumption than *StashG* despite using more SRAM.

Hardware Unit	Hit Energy	Miss Energy
Scratchpad	55.3 pJ	–
Stash	55.4 pJ	86.8 pJ
L1 cache	177 pJ	197 pJ
TLB access	14.1 pJ	14.1 pJ[8]

Table 3: Per access energy for various hardware units.

Reuse highlights the stash's data compaction and global visibility and addressability. This microbenchmark repeatedly invokes a kernel which accesses a single field from each element of a data array. The relevant fields of the data array can fit in the stash but not in the cache because it is compactly stored in the stash. Thus, each subsequent kernel can reuse data that has been loaded into the stash by a previous kernel and lazily written back. In contrast, the scratchpad configurations (including ScratchGD) are unable to exploit reuse because the scratchpad is not globally visible. Cache cannot reuse data because it is not able to compact data.

5.4.2. Applications

We selected seven larger benchmark applications to evaluate the effectiveness of the stash for current GPU applications. The applications are from Rodinia [12] (LUD - 256x256, Backprop/BP - 32 KB, NW - 512x512, and Pathfinder/PF - 10x100K), Parboil [34] (SGEMM - A: 128x96, B: 96x160 and Stencil: 128x128x4, 4 iterations), and Computer Vision [10] (SURF- 66 KB image). All applications were selected for their use of the scratchpad. We manually modified each application to use a unified shared memory address space (i.e., we removed all explicit copies between the CPU and GPU address spaces) and added the appropriate map calls based on the different stash modes of operation (from Section 3.3). For information on each application's mapping types, see [24]. We use only a single CPU core and do not parallelize these applications because they perform very little work on the CPU.

6. Results

6.1. Access Energy Comparisons

Table 3 shows per access energy for various hardware components used in our simulations. The table shows that scratchpad access energy (no misses for scratchpad accesses) is 29% of the L1 cache hit energy. Stash's hit energy is comparable to that of scratchpad and its miss energy is 41% of the L1 cache miss energy. Thus accessing the stash is more energy-efficient than a cache and the stash's hit energy is comparable to that of a scratchpad.

6.2. Microbenchmarks

Figure 5 shows the execution time, energy, GPU instruction count, and network traffic for our microbenchmarks using scratchpad (Scratch), cache (Cache), scratchpad with DMA (ScratchGD), and stash (Stash). We omit the remaining configurations (ScratchG and StashG) because the microbenchmarks (except Pollution) do not have any global memory accesses, so ScratchG is identical to Scratch and StashG is identical to

[8] We do not model a TLB miss, so all our TLB accesses are charged as if they are hits.

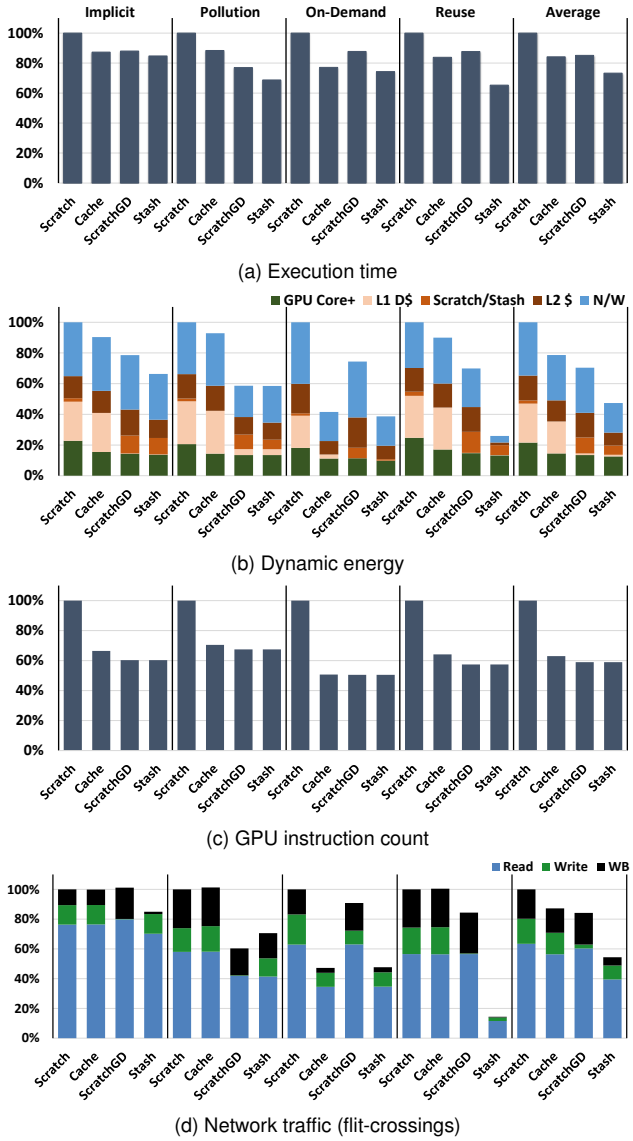


Figure 5: Comparison of microbenchmarks. The bars are normalized to the *Scratch* configuration.

Stash. The energy bars are subdivided by where energy is consumed: GPU core,[9] L1 cache, scratchpad/stash, L2 cache, or network. Network traffic bars are subdivided by message type: read, write, or writeback.

Our results show that, on average, the stash reduces execution time and consumes less energy than the scratchpad, cache, and DMA configurations – 13%, 27%, and 14% lower execution time, respectively and 35%, 53%, and 32% less energy, respectively. Overall, the microbenchmark results show that (a) the stash performs better than scratchpad, caches, and scratchpad with DMA; and (b) data structures and access patterns that are currently unsuitable for scratchpad storage can be efficiently mapped to stash. Next we discuss the sources of these benefits for each configuration.

[9]GPU core+ includes the instruction cache, constant cache, register file, SFU, FPU, scheduler, and the core pipeline.

Scratchpad vs. Stash

Compared with the scratchpad configuration, stash provides the following global addressing and global visibility benefits: *Implicit data movement*: By implicitly transferring data to local memory, *Stash* executes 40% fewer instructions than *Scratch* for the Implicit benchmark and decreases execution time by 15% and energy consumption by 34%.
No cache pollution: Unlike scratchpads, stash does not access the cache when transferring data to or from local memory. By avoiding cache pollution, *Stash* consumes 42% less energy and reduces execution time by 31% in the Pollution benchmark.
On-demand loads into structures: The On-demand microbenchmark results show the advantages of on-demand loads. Since stash only transfers the data it accesses into local memory, *Stash* reduces energy consumption by 61% and execution time by 26% relative to *Scratch*, which must transfer the entire data array to and from the local memory.
Lazy writebacks/Reuse: The Reuse microbenchmark demonstrates how the the stash's lazy writebacks enable data reuse across kernels. By avoiding repeated transfers, *Stash* consumes 74% less energy and executes in 35% less time than *Scratch*.

The primary benefit of scratchpad is its energy efficient access. Scratchpad has less hardware overhead than stash and does not require a state check on each access. However, the software overhead required to load and write out data limits the use cases of scratchpad to regular data that is accessed frequently within a kernel. By adding global visibility and global addressability, stash memory eliminates this software overhead and can attain the energy efficiency of scratchpad (and higher) on a much larger class of programs.

Cache vs. Stash

Compared to cache, stash benefits from direct addressability and compact storage. With direct addressability, stash accesses do not need a tag lookup, do not incur conflict misses, and only need to perform address translation on a miss. Thus a stash access consumes *less energy* than a cache access for both hits and misses and the stash reduces energy by 35% on average.

In addition to the benefits from direct addressing, the Pollution and Reuse microbenchmarks also demonstrate the benefits of *compact storage*. In these microbenchmarks the cache configuration repeatedly evicts and reloads data because it is limited by associativity and cache line storage granularity. Thus it cannot efficiently store a strided array. Because the stash provides compact storage and direct addressability, it outperforms the cache for these microbenchmarks: up to 71% in energy and up to 22% in execution time.

Cache is able to store much more irregular structures and is able to address a much larger global data space than stash. However, when a data structure is linearizable in memory and can fit compactly in the stash space, stash can provide much more efficient access than cache with significantly less overhead than scratchpad.

ScratchGD vs. Stash

Applying DMA to a scratchpad configuration mitigates many of the scratchpad's inefficiencies by preloading the data directly into the scratchpad. Even so, such a configuration

still lacks many of the benefits of global addressability and visibility present in stash. First, since scratchpads are not globally addressable, DMA must explicitly transfer all data to and from the scratchpad before and after each kernel. All threads must wait for the entire DMA load to complete before accessing the array, which can stall threads unnecessarily and create bursty traffic in the network. Second, DMA must transfer all data in the mapped array whether or not it is accessed by the program. The *On-demand* microbenchmark highlights this problem: when accesses are sparse and unpredictable stash achieves 48% lower energy and 48% less network traffic. Third, since scratchpad is not globally visible, DMA is unable to take advantage of *reuse* across kernels; therefore, stash sees 83% traffic reduction, 63% energy reduction, and 26% execution time reduction in the Reuse microbenchmark. DMA also incurs additional local memory accesses compared with stash because it accesses the scratchpad at the DMA load, the program access, and the DMA store. These downsides cause DMA to consume additional energy than the stash: Stash's stash/scratchpad energy component is 46% lower than DMA's on average.

Pollution's network traffic is 17% lower with DMA compared to stash. In Pollution, the stash's registration requests increase traffic when data is evicted from the stash before its next use because stash issues both registration and writeback requests while DMA only issues writeback requests. In general though, global visibility and addressability improve performance and energy and make stash feasible for a wider range of data access patterns.

These results validate our claim that the stash combines the advantages of scratchpads and caches into a single efficient memory organization. Compared to a scratchpad, the stash is globally addressable and visible; compared to a cache, the stash is directly addressable with more efficient lookups and provides compact storage. Compared with a non-blocking DMA engine, the stash is globally addressable and visible and can transfer data on-demand. Overall, the stash configurations always outperform the scratchpad and cache configurations, even in situations where a scratchpad or DMA engine would not traditionally be used, while also providing decreased network traffic and energy consumption.

6.3. Applications

Figure 6 shows execution time and energy for our 7 applications on all configurations except *ScratchGD*, normalized to *Scratch*. We omit the network traffic and instruction count graphs [24] for space. We also omit details on *ScratchGD* for space and because its results are similar to *StashG*.[10] Current applications do not fully exploit the benefits of stash over a DMA-enhanced scratchpad (e.g., on-demand accesses and cross-kernel reuse); nevertheless, our results show that the stash is comparable to such an enhanced scratchpad [24] for current applications and significantly better for future use cases (Section 6.2).

Compared to other configurations in Figure 6, stash improves both performance and energy: compared to *Scratch* (the best scratchpad version) and *Cache*, on average *StashG* (the best stash version) reduces execution time by 10% and 12% (max 22% and 31%), respectively, while decreasing energy by 16% and 32% (max 30% and 51%), respectively. Next we analyze these results in more detail.

Scratch vs. ScratchG vs. Cache

Figure 6a shows that *ScratchG* is usually worse than (and at best comparable to) *Scratch*, in terms of both execution time and energy (7% and 12% worse, respectively, on average). This is because the global accesses that are converted to scratchpad accesses increase the overall instruction count (the global accesses are better off being global as there is no temporal locality for these accesses).

Comparing *Cache* and *Scratch*, we find that converting scratchpad accesses to global (cache) accesses, in general, does not improve performance and increases energy consumption (PF is one notable exception).

These results show that overall, the allocation of memory locations to the scratchpad in the original applications is reasonable for our baseline system. We therefore compare only the *Scratch* configuration to *Stash* and *StashG* below.

Scratch vs. Stash vs. StashG

In contrast to the scratchpad, moving all global data to the stash is (modestly) beneficial – compared to *Stash*, *StashG* reduces execution time by 3% and energy by 7% on average. This shows that more data access patterns can take advantage of the stash.

Figure 6a shows that both stash configurations reduce execution time compared to *Scratch*. *Stash* reduces execution time compared to *Scratch* by exploiting the stash's global addressability and visibility. *StashG* shows a further modest improvement in execution time by exploiting the stash's ability to remove index computations (index computations performed by the core for a global access are now performed more efficiently by the $stash-map$ in hardware). As a result, *StashG* reduces execution time by an average of 10% (max 22%) compared to *Scratch*.

Figure 6b shows that the stash configurations also reduce energy. Compared to *Scratch*, *Stash* uses the stash's global addressability to remove explicit copies and reduce both GPU core energy and L1 cache energy. By converting global accesses to stash accesses, *StashG* reduces energy even further. There are two positive energy implications when the global accesses are converted to stash accesses: (i) a stash access is more energy-efficient compared to a cache access; and (ii) the index computations performed by the stash-map mentioned above consume less energy (which reduces 'GPU core+' portion for *StashG* compared to *Stash*). These advantages help *StashG* to reduce energy by 16% (max 30%) compared to *Scratch*.

Overall, these results show that the stash effectively combines the benefits of scratchpads and caches to provide higher performance and lower energy even for current applications that are not designed to exploit the stash's new use cases.

7. Related Work

There is much prior work on improving private memories for CPUs and GPUs. Table 4 compares the most closely related work to stash using the benefits from Table 1:

[10] Adding DMA to *ScratchG* outperforms adding it to *Scratch*.

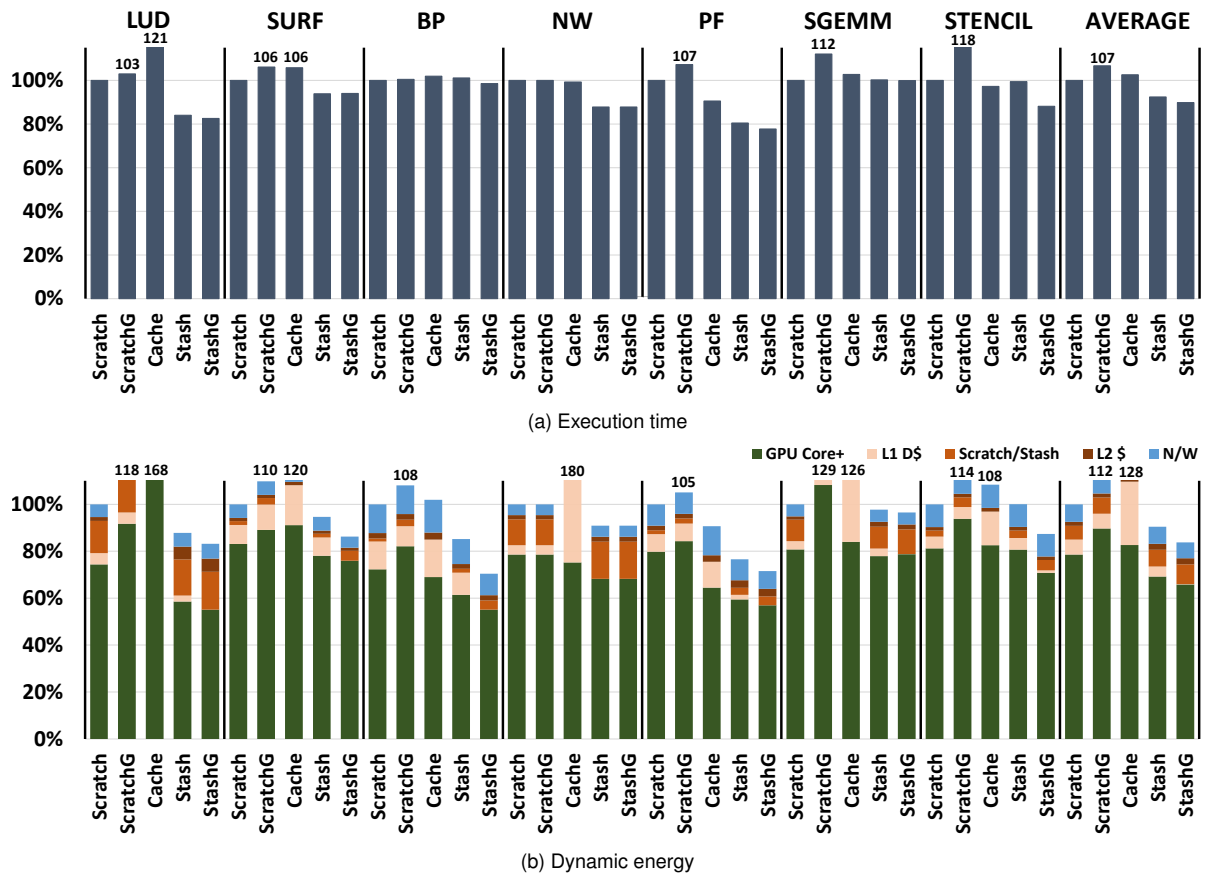


Figure 6: Comparison of configurations for the seven benchmarks. The bars are normalized to the *Scratch* configuration.

Feature	Benefit	Bypass L1 [4]	Change Data Layout [11, 13]	Elide Tag [33, 39]	Virtual Private Mems [15, 16, 17, 28]	DMAs [9, 21]	Stash
Directly addressed	No address translation HW access	✓	✗	✗, ✓	✓	✓	✓ (on hits)
	No tag access	✓	✗	✓ (on hits)	✗	✓	✓
	No conflict misses	✓	✗	✗	✓	✓	✓
Compact storage	Efficient use of SRAM storage	✓	✓	✗	✓	✓	✓
Global addressing	Implicit data movement	✗	✓	✓	✗	✗	✓
	No pollution of other memories	✓	✓	✓	✓	✓	✓
	On-demand loads into structure	✗	✓	✓	✗	✗	✓
Global visibility	Lazy writebacks to global AS	✗	✓	✓	✗	✗	✓
	Reuse across kernels or phases	✗	✓	✓	Partial	✗	✓
Applied to GPU		✓	✗, ✓	✗	✗, ✗, ✗, ✓	✓	✓

Table 4: Comparison of stash and prior work.

Bypassing L1: (MUBUF [4])): L1 bypass does not pollute the L1 when transferring data between global memory and the scratchpad, but does not offer any other benefits of the stash.

Change Data Layout: (Impulse [11], Dymaxion [13]): By compacting data that will be accessed together, this technique provides an advantage over conventional caches, but does not explicitly provide other benefits of scratchpads.

Elide Tag: (TLC [33], TCE [39]): This technique optimizes conventional caches by removing the need for tag accesses (and TLB accesses for TCE) on hits. Thus, this technique provides some of the benefits scratchpads provide in addition to the benefits of caches. However, it relies on high cache hit rates (which are not common for GPUs) and does not remove conflict misses or provide compact storage of the stash.

Virtual Private Memories (VLS [17], Hybrid Cache [15], BiN [16], Accelerator Store [28]): Virtualizing private memories like scratchpads provides many of the benefits of scratchpads and caches. However, it requires tag checks and has explicit data movement which prevents lazily writing back data to the global address space. Furthermore, these techniques do not support on-demand loads[11] and only partially support reuse.[12]

[11] VLS does on-demand accesses after thread migration on a context switch, but the initial loads into the virtual private store are through DMA. Although we do not discuss context switches here, the stash's global visibility and coherence means that we can lazily write back stash data on context switches.

[12] In VLS, if two cores want to read/write the same global data conditionally in alternate phases in their VLS's, then the cores have to write back the data at the end of the phase even if the conditional writes don't happen.

DMAs: (CudaDMA [9], D^{2}MA [21]): A DMA engine on the GPU can efficiently move data into the scratchpad without incurring excessive instruction overhead and polluting other memories. However, as discussed earlier, it does not provide the benefits of on-demand loads (beneficial with control divergence), lazy writebacks, and reuse across kernels.

In summary, while these techniques provides some of the same benefits as the stash, *none of them provide all of the benefits*.

8. Conclusion

We present a new memory organization, stash, that combines the performance and energy benefits of caches and scratchpads. Like a scratchpad, the stash provides compact storage and does not have conflict misses or overheads from tags and TLB accesses. Like a cache, the stash is globally addressable and visible, enabling implicit and on-demand data movement and increased data reuse. The stash is a forward looking memory organization that improves results for current applications and increases use cases for heterogeneous systems.

Our work enables many new research opportunities. We plan to make the stash even more efficient by applying optimizations such as prefetching, providing a flexible (vs. cache line based) communication granularity, and a mechanism to bypass the indirection of the registry lookup by determining the remote location of data. The stash's ability to reuse data also opens up possibilities for new stash-aware scheduling algorithms. Using dynamically reconfigurable SRAMs will allow the stash and cache to use different sizes for different applications. We also plan to explore automating stash mappings with compiler support. More broadly, we would like to expand the stash idea to other specialized private memories in the context of other compute units (e.g., CPUs and custom accelerators) to include them in the unified, coherent address space without losing the benefits of specialization.

References

[1] "HSA Platform System Architecture Specification," 2015.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *ISPASS*, 2009.

[3] AMD, "Compute Cores," https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf.

[4] AMD, "Sea Islands Series Instruction Set Architecture," http://developer.amd.com/wordpress/media/2013/07/AMD_Sea_Islands_Instruction_Set_Architecture.pdf.

[5] O. Avissar, R. Barua, and D. Stewart, "An Optimal Memory Allocation Scheme for Scratchpad-based Embedded Systems," *TACO*, 2002.

[6] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[7] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems," in *CODES*, 2002.

[8] A. Basu, M. D. Hill, and M. M. Swift, "Reducing Memory Reference Energy with Opportunistic Virtual Caching," in *ISCA*, 2012.

[9] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization," in *SC*, 2011.

[10] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," in *Computer Vision – ECCV*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, vol. 3951.

[11] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a Smarter Memory Controller," in *HPCA*, 1999.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.

[13] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems," in *SC*, 2011.

[14] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *PACT*, 2011.

[15] J. Cong, M. A. Ghodrat, M. Gill, C. Liu, and G. Reinman, "BiN: A buffer-in-NUCA Scheme for Accelerator-rich CMPs," in *ISLPED*, 2012.

[16] J. Cong, K. Gururaj, H. Huang, C. Liu, G. Reinman, and Y. Zou, "An Energy-efficient Adaptive Hybrid Cache," in *ISLPED*, 2011.

[17] H. Cook, K. Asanovic, and D. A. Patterson, "Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments," UC Berkeley, Tech. Rep., 2009.

[18] DoE, "Top Ten Exascale Research Challenges," http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf, 2014.

[19] B. Hechtman, S. Che, D. Hower, Y. Tian, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood, "QuickRelease: A throughput-oriented approach to release consistency on GPUs," in *HPCA*, 2014.

[20] IntelPR, "Intel Discloses Newest Microarchitecture and 14 Nanometer Manufacturing Process Technical Details," *Intel Newsroom*, 2014.

[21] D. A. Jamshidi, M. Samadi, and S. Mahlke, "D2MA: Accelerating Coarse-grained Data Transfer for GPUs," in *PACT*, 2014.

[22] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, 2011.

[23] F. Kjolstad, T. Hoefler, and M. Snir, "Automatic Datatype Generation and Optimization," in *PPoPP*, 2012.

[24] R. Komuravelli, "Exploiting Software Information for an Efficient Memory Hierarchy," Ph.D. dissertation, The University of Illinois at Urbana-Champaign, 2014.

[25] J. Leng, T. Hetherington, A. El Tantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013.

[26] C. Li, Y. Yang, D. Hongwen, Y. Shengen, F. Mueller, and H. Zhou, "Understanding the Tradeoffs Between Software-Managed vs. Hardware-Managed Caches in GPUs," in *ISPASS*, 2014.

[27] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[28] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The Accelerator Store: A Shared Memory Framework for Accelerator-Based Systems," *TACO*, 2012.

[29] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Computer Architecture News*, 2005.

[30] N. Nguyen, A. Dominguez, and R. Barua, "Memory Allocation for Embedded Systems with a Compile-time-unknown Scratchpad Size," *TECS*, 2009.

[31] NVIDIA, "CUDA SDK 3.1," http://developer.nvidia.com/object/cuda_3_1_downloads.html.

[32] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *MICRO*, 2013.

[33] A. Sembrant, E. Hagersten, and D. Black-Shaffer, "TLC: A Tag-less Cache for Reducing Dynamic First Level Cache Energy," in *MICRO*, 2013.

[34] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Dept. of ECE and CS, Univ. of Illinois at Urbana-Champaign, Tech. Rep., 2012.

[35] H. Sung and S. V. Adve, "Supporting Arbitrary Sychronization without Writer-Initiated Invalidations," in *ASPLOS*, 2015.

[36] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient Hardware Support for Disciplined Non-determinism," in *ASPLOS*, 2013.

[37] S. Udayakumaran and R. Barua, "Compiler-decided Dynamic Memory Allocation for Scratchpad Based Embedded Systems," in *CASES*, 2003.

[38] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic Allocation for Scratchpad Memory Using Compile-time Decisions," *TECS*, 2006.

[39] Z. Zheng, Z. Wang, and M. Lipasti, "Tag Check Elision," in *ISLPED*, 2014.