# S-L1: Resurrecting the GPU L1 Cache by Emulating it in Software to Improve GPGPU Data Processing Performance


## Abstract

*GPU L1 caches are ineffective for data-intensive GPGPU applications, because they are too small (e.g., 48K per 192 processor cores), and their cache lines are too large (e.g., 128B). This results in excessive cache trashing and a low hit rate, given the large number of threads that are expected to run on the GPU. GPU L1 caches are so ineffective that recent GPU models have L1 caching of application data disabled by the vendor. The ineffectiveness of the L1, coupled with the long latencies to access L2 and DRAM, makes the GPU memory hierarchy the primary bottleneck for data-intensive applications, resulting in low core utilization and poor application performance.*

*In this paper we propose and evaluate an L1 cache for GPUs implemented entirely in software, using the on-chip scratch-pad memory called "shared memory". No changes to GPU source code is required — compiler transformations insert the necessary code. Initial performance evaluation suggests that despite the 8% performance overhead associated with our software-based caching layer, we were able to achieve speedups of between 0.89 and 6.4 (2.45 avg) on ten GPU-local streaming applications. Combining software-L1 with BigKernel, the fastest known technique accelerating GPU applications processing large data sets located in CPU memory, leads to speedups between 1.04 and 1.45 (1.18 avg.) over BigKernel alone, and speedups between 1.07 and 11.24 (4.32 avg.) over the fastest CPU multicore implementations.*


## 1. Introduction

Our research focus is on using Graphical Processing Units (GPUs) to accelerate what in the commercial world is popularly referred to as "big data" computations. These computations are dominated by functions that filter, transform, aggregate, consolidate, or partition huge input data sets. They typically involve simple operations on the input data, are trivially parallelizable, and the input data exhibits no (or very low) reuse. In the GPU world these type of computations are referred to as *streaming computations*.

GPUs, on the surface, appear to be ideal accelerators for streaming computations: with their many processing cores, today's GPUs have 10X the compute power of modern CPUs, and they have close to 6X the memory bandwidth of modern CPUs,[1] yet are priced as commodity components.

[1] For example, Nvidia GTX Titan has 4.5 TFLOPS of compute power and 288 GB/s of memory bandwidth, while Intel Ivy Bridge has 460 GFLOPS of compute power and 52 GB/s of memory bandwidth.

However, a number of issues had until recently prevented effective acceleration in practice. GPUs and CPUs have separate memories so that the input data must first be copied over to GPU memory, causing extra overhead [6]. Moreover, the PCIe link that connects the two memories has limited bandwidth and transferring data over the PCIe bus at close to theoretically maximum bandwidth is non-trivial [13]. Finally, the high GPU memory bandwidth can only be exploited when GPU threads executing simultaneously access memory in a *coalesced* fashion, where the accessed memory locations adjacent to each other. Recent research efforts have mitigated these issues to a large extent. For example, one research group reported speedups on seven realistic streaming computations of between 0.74 and 7.3 (3.0 avg.) over the most efficient CPU multicore implementations and between 4.0 and 35.0 (14.2 avg.) over the most efficient single CPU core implementations.

These efforts have shifted the primary bottleneck preventing higher GPU core utilization from the PCIe link to the GPU-side memory hierarchy. In particular, three factors currently prevent further improvements in core utilization. First, the GPU L1 caches are entirely ineffective [8]. For the number of cores typical in modern GPUs, the L1 caches are too small and their cache line sizes are disproportionally large given the small cache size. For example, the L1 on the Nvidia GTX 680 we used to run our experiments can be configured to be at most 48KB per 192 cores and the cache line size is 128B. At best, this leaves just two cache lines per core. Yet GPGPU best practices expect many threads to run simultaneously per core (supported by 340 4-byte registers per core), each having multiple memory accesses in-flight.

Given a large number of executing threads, each issuing multiple memory accesses, cache lines are evicted before there is any reuse, causing a high degree of cache thrashing and attendantly low L1 hit rate. As an example, Figure 1 depicts the L1 hit rate as a function of the number of threads executing when running the Unix word count utility, WC. The hardware L1 has proven to be so ineffective that some recent GPU chip sets, like the Nvidia GTX 680, only allow the use of L1 for register spills and stack caching and not for application data. Moreover, if historical trends are any indication (see Section 2.2), we can not expect GPU L1 caches to become significantly more effective any time in the near future.

A second factor preventing further improvements in core utilization is the high latency to L2 and DRAM. On the Nvidia GeForce GTX 680, the latency of a single, isolated access to L2 and DRAM is 210 and 340 cycles, respectively, and increases substantially when the number of threads/accesses

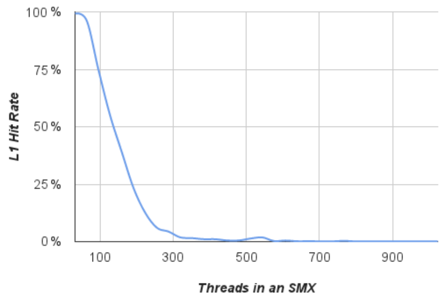


**Figure 1:** L1 hit rate when running wc on an older NVidia 560 GPU that supports the caching of application data. The target data is partitioned into *n* chunks with each chunk assigned to a thread for processing. With effective caching, the first access of each thread results in a miss, but the subsequent 127 accesses result in cache hits. Without effective caching, these accesses result in 128 misses.

goes up. Hardware multithreading can hide some of this latency, but far from all of it, as we show in Section 2.3.

Thirdly, bottlenecks on the path from L2/DRAM to GPU cores prevent scaling of memory throughput and can prevent exploitation of the full L2 and DRAM bandwidth, as we will also show in Section 2.3. Thus, for memory intensive applications, average memory access latencies will be significantly higher than the latency of single, isolated access.

The above three factors noticeably affect the workloads we are interested in, as many of the big-data computations are completely memory bound and dominated by character accesses, where a trivial amount of computation is needed per access. The end result: extremely low GPU core utilization.

In this paper, we address the issues in the GPU memory hierarchy by proposing and evaluating an L1-level cache implemented entirely in software. The software-L1 cache (S-L1) is located in software-managed GPU *shared memory*, which is positioned at the same level as the L1, has the same access latency as the L1 (80 cycles on the Nvidia GTX 680), and is also small (max. 48KB per GTX 680 multiprocessor). S-L1 does not require modifications to the GPU application code; instead the compiler inserts the code required to implement S-L1.

The design of the S-L1 is guided by three key principles to deal with the small size of the shared memory:

1. **Private cache segments**: the S-L1 is partitioned into thread-private cache lines, instead of having all threads share the cache space;
2. **Smaller cache lines**: the cache-line size is, at 16B, smaller than what is typical in GPUs, allowing a larger number of cache-lines to be shared per thread; and
3. **Selective caching**: the data of only a select number of data structures are cached.

The objective of this design is to significantly decrease average memory access times and minimize S-L1 cache thrashing. It is implemented entirely in software using a fairly straightforward runtime scheme where the code to manage and use the cache is added by using simple compiler transformations. The specific parameters of the S-L1 cache are determined at runtime during an initial brief monitoring phase, which also identifies the potential cache hit rate of each data structure. After the monitoring phase, the computation is executed using the S-L1 cache for the *n* data structures that have the highest cache hit rates, where *n* is selected based on the amount of cache space available to each thread.

In our experimental evaluation, S-L1 achieves speedups of between 0.89 and 6.4 (2.45 avg) on ten GPU-local applications even though each memory access requires the additional execution of a minimum of 4 instructions (and up to potentially hundreds of instructions). Combining S-L1 with BigKernel, the fastest known technique accelerating GPU applications processing large data sets located in CPU memory, leads to speedups between 1.04 and 1.45 (1.18 avg.) over BigKernel alone, and speedups between 1.07 and 11.24 (4.32 avg.) over the fastest CPU multicore implementations.

This paper makes the following two specific contributions:

1. we characterize the performance behavior of the GPU memory hierarchy and identify some of its bottlenecks using a number of experiments, and
2. we propose S-L1, a level-1 cache implemented entirely in software and evaluate its performance; novel features of S-L1 include a run-time scheme to automatically determine the parameters to configure the cache, selective caching, and thread-specific cache partitions.

The paper is organized as follows. Section 2 describes background information about GPUs, some architectural trends we have observed over the past years, and a few micro-architectural behavior that motivated us to do this work. We described the design and implementation of S-L1 in Section 3 and present the results of our experimental performance evaluations in Section 4. Section 5 discusses related work, and we close with concluding remarks in Section 6.

## 2. Background

We first describe the architecture of the Nvidia GTX 680 to provide a brief overview of typical current GPU architectures. This subsection can be skipped by readers already familiar with GPU architectures.

We then present several GPU architectural trends to provide insight as to where future GPU architectures might be headed. We use these trends to motivate implementing our L1 cache.

Finally, the third subsection presents several limitations of the GPU memory hierarchy and offers some insights into the nature of those limitations, further motivating our S-L1 design.

### 2.1. GPU Background

Figure 2 shows the high-level architecture of the Nvidia GTX 680. We describe this particular chip because it was used in our experimental evaluation, but also because it is representative of current GPUs. In particular, the most recent offering by Nvidia, the GTX 780 uses the same basic Kepler architecture [23],

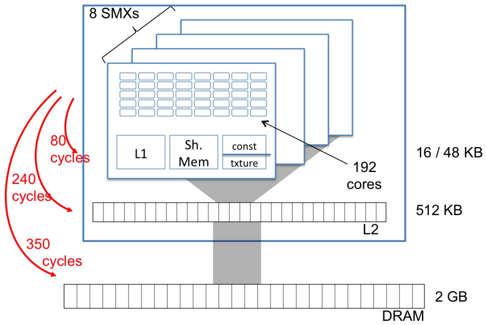


**Figure 2:** Architecture of the Nvidia GTX 680.

although the number of cores, memory/cache sizes, and some micro-architectural details will differ.

The GTX 680 consists of eight *streaming multiprocessors* (SMX), each of which contains 192 computing cores, 64K 4-byte registers, 16KB-48KB L1 cache and 16KB-48KB software managed on-chip memory called *shared memory*, accessible to all cores of the SMX.[2] A 512KB L2 cache is shared by all SMXs, and the L2 cache, organized into four banks, is connected to 2GB DRAM memory called *global memory*. We refer to this global memory as *GPU memory* in this paper to differentiate it from CPU main memory. Access latency to registers, L1, shared memory, L2 and DRAM is 10, 80, 80, 210, and 340 cycles, respectively. The theoretically maximum bandwidth from L1, shared memory, L2 and DRAM have been reported as 190.7GB/s, 190.7GB/s, 512GB/s, and 192GB/s, respectively [21].

The term *kernel* is used to denote the function that executes on the GPU by a collection of threads in parallel. The programmer configures the kernel to be executed by a given number of GPU threads. These threads are grouped into *thread blocks* as configured by the programmer, with a maximum 1024 threads per thread block. Each thread block is assigned to a SMX by a hardware scheduler. Each SMX can host up to a maximum of 2048 running threads (e.g. two full-sized thread blocks) at a time. Threads of a thread block are further divided into groups of 32, called *warps*. The threads in a warp execute in lock-step because groups of 32 cores share the same instruction scheduler. This lock-step execution will lead to *thread divergence* if, on a conditional branch, threads within the same warp take different paths. Thread divergence can lead to serious performance degradations.

Memory requests issued by threads of a warp that fall within the same aligned 128-byte region are coalesced into one memory request by a hardware *coalescing unit* before being sent to memory, resulting in only one 128-byte memory transaction. Parallel memory accesses from a warp to data are defined as $n$-way coalesced if $n$ of the accesses fall within the same aligned 128-byte region.

[2] Each SMX also contains a texture cache and a constant cache, but they are not relevant for our objectives and hence not considered in this paper.

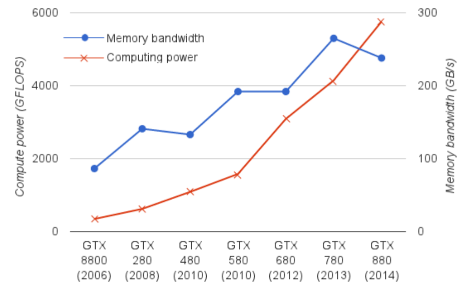


**Figure 3:** Compute power vs. memory bandwidth over time/GPU generations.

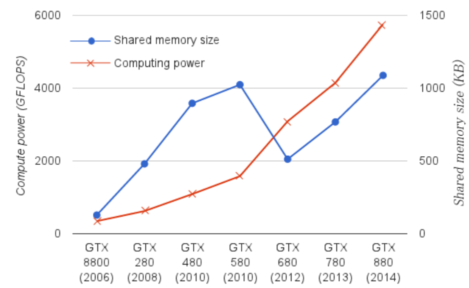


**Figure 4:** Compute power vs. size of L1 and shared memory over time/GPU generation.

### 2.2. Some Historical Trends

Figure 3 depicts how the aggregate compute power (in GFLOPS) and the memory bandwidth (in GB/s) of Nvidia GPUs have evolved from their earliest CUDA-enabled version to the current version. Compute power has been increasing steadily at a steep slope. Memory bandwidth has also been increasing, but not as quickly. As a result, memory bandwidth per FLOP has been decreasing from 250 bytes/KFLOP for the GTX 8800 to 41 bytes/KFLOP for the GTX 880.

If this trend continues then the GPU cores will become increasingly memory starved, considering that the number of registers are limited and the on-chip L1 cache is ineffective. Strategies to optimize GPU applications to make more efficient use of the memory hierarchy will likely become more important going forward.

Figure 4 depicts how the aggregate compute power and the total size of on-chip memory (L1 cache and shared memory) has evolved over time. The total amount of on-chip memory varies over time and at one point even decreases substantially from one generation to the next. It has clearly not kept up with the increase in compute power.

Given the fact that future GPU generations may have smaller on-chip memory sizes, as has happened in the past, GPU programmers cannot assume the availability of a specific shared memory size. As a result, the programmer will need to design GPU applications so that they configure the use of shared memory at run-time and possibly restrict the number of threads used by the application. Or use run-time libraries, such as the one we are presenting in this paper, that automatically adjust program behavior to the available hardware resources.

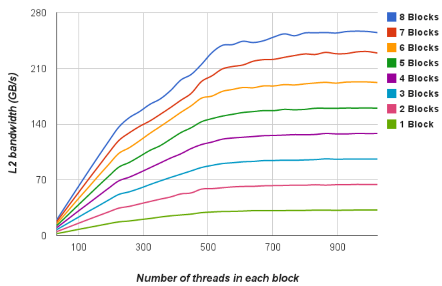


**Figure 5:** L2 memory throughput as a function of number of threads in a thread block. Each curve represents the throughput for a different number of thread blocks (1 to 8). thread-blocks

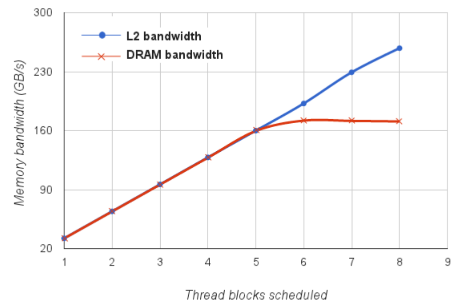


**Figure 6:** L2 and DRAM memory bandwidth as a function of number of thread blocks where each thread block is running 1,024 threads and the memory accesses are 4-way coalesced.

### 2.3. Behavior of GPU memory access performance

GPU vendors do not disclose much information on the micro-architecture of their GPUs. Hence, in order to optimize GPGPU programs so that they can more efficiently exploit hardware resources, it is often necessary to reverse engineer the performance behavior of the GPUs through experimentation. In this section, we present the results of some of the experiments we ran to gain more insight. All results we present here were obtained on an Nvidia GTX 680.

**2.3.1. Memory access throughput** In our first set of experiments, we used a micro-benchmark that has threads read disjoint subsets of data located in the L2 cache as quickly as they can. The benchmark is parameterized so that the degree of coalescing can be varied by controlling the addresses of the data being accessed by each thread. Figure 5 shows the maximum L2 memory bandwidth obtained, measured as number of bytes transferred over the network, when servicing 4-way coalesced accesses from the L2 cache as the number of threads running in each thread block is increased up to 1024.

Each curve represents a different number of thread blocks used, each block including the same number of threads. The thread blocks are assigned to SMXs in a round robin manner by the hardware. Focusing on the bottom curve, representing an experiment that has just one thread block running on one SMX, one can see that the memory throughput flattens out after about 512 threads at slightly less than 32 GB/s.[3]

We observe similar behavior for DRAM (not shown) when we adjusted the micro-benchmarks to only access data certain to not be in the L2 cache (i.e., resulting in L2 misses), except that the throughput flattens out earlier at about 480 threads, reaching a peak bandwidth of 170 GB/s with 8 blocks.

It is difficult to assess what causes the stagnation in L2 and DRAM throughput. However the near-linear scalability with the number of thread blocks indicates that the bottleneck is in the interconnect or in the SMX itself (e.g., coalescing units) rather than L2 or DRAM. This is shown in Figure 6 where we show the throughput as a function of the number of thread blocks with each thread block running 1,024 threads. Each point along the L2 bandwidth curve is equal to the end point (at 1,024 threads) of the corresponding curve of Figures 5. L2 throughput increases almost linearly, reaching close to 256 GB/s with 8 blocks (and close to 170 GB/s for DRAM).

All of the results presented above measured the amount of data transferred to the SMXs by the hardware. However, depending on the application, much of this data may not actually be used by the application. For example, for non-coalesced accesses, each 4-byte integer access will result in 32 byte transfers, of which only 4 are actual used.

The throughput limitations of L2 and DRAM, as well as the fact that only some of the data transferred is used by the application, indicates that experienced memory access latencies will be far larger than the theoretical access latencies presented in Section 2.1.

This implies that an SMX-local L1 cache, whether implemented in hardware or software, can dramatically reduce average application access latencies if implemented appropriately. In particular, in contrast to L2 and DRAM throughput, shared memory throughput within an SMX (not shown) does not flatten out and reaches 60GB/s (for an aggregate throughput of close to 500GB/s with 8 SMXs).

## 3. S-L1 Design and Implementation

### 3.1. Overview

S-L1 is a level 1 cache implemented entirely in software. It uses the space available in each SMX's shared memory, which has the same access latency as the hardware L1. We also considered using SMX's texture cache, but it is a hardware-managed, read-only cache and thus, does not suit S-L1 needs.

The design of S-L1 is based on three key design elements. First, the cache space is partitioned into thread-private cache segments, each containing one or more cache lines. The number of cache lines in each segment is determined dynamically at runtime based on the amount of available shared memory, the number of threads running in the SMX, and the size of the cache line. The decision to use thread-private cache segments is based on the fact that inter-thread data sharing is rare in the streaming applications we are targeting. Therefore, the threads mostly process data independently in disjoint lo-

[3] Our experiments show that varying the degree of coalescing does not completely remove the flattening out behavior. However, the smaller the coalescing degree (e.g. 1-way coalesced), the earlier the curve flattens out.

cations of memory. Allowing all threads to share the entire cache space would likely result in unnecessary collisions.

Secondly, we use relatively small cache lines. The optimal cache line size depends to a large extent on the applications' memory access patterns. Larger cache lines perform better for applications exhibiting high spacial locality, but they perform poorer for applications with low spacial locality due to (*i*) the extra overhead of loading the cache lines requiring multiple memory transactions and (*ii*) the increased cache thrashing because fewer cache lines are available.

We decided on using 16-byte cache lines after experimenting with different cache line sizes — see Section 4.6. This size works well because 16B is the widest load/store size available on modern GPUs, allowing the load/store of an entire line with one memory access. Moreover, the use of larger lines is constrained by the fact that the size of shared memory is limited — the most recent GPUs only have 48KB of shared memory per SMX for up to 2,048 threads.

Thirdly, we only cache a limited number of data structures.[4] The number of cache lines allocated to each thread (*CLN*) determines how many data structures we cache. *CLN* is calculated at runtime as

$$\lfloor (shMemSizePerSM/numThreadsPerSM)/cacheLineSize \rfloor$$

where *shMemSizePerSM* is the amount of shared memory available per SMX, *numThreadsPerSM* is the total number of threads allocated on each SMX, and *cacheLineSize* is the size of the cache line; i.e., 16 bytes in our current design.

The amount of shared memory available for the S-L1 cache depends on how much shared memory has previously been allocated by the application. The application can allocate shared memory statically or dynamically at run time. Hence, a mixed compile-time/runtime approach is required to identify how much shared memory remains available for S-L1. *NumThreadsPerSM* is calculated at runtime, in part by using the configuration the programmer specifies at kernel invocation and in part by calculating the maximum number of threads that can be allocated on each SMX which in turn depends on the resource usage of GPU threads (e.g. register usage) and available resources of SMX, which is extracted at compile-time and runtime, respectively.

Once the number of cache lines per thread – *CLN* – is determined, up to that many data structures are marked as S-L1 cacheable and a separate cache line is assigned to each.[5] Data structures that are not marked will not be cached and are accessed directly from memory. If the available size of shared memory per thread is less than *cacheLineSize* (i.e., too much shared memory has already been allocated by the application), no cache lines are assigned to threads (i.e. $CLN = 0$), so the software cache is effectively disabled.

To determine which data structures to cache, we evaluate the benefit of caching the data of each data structure using a short monitoring phase at runtime. In the monitoring phase, the core computation of the application is executed for a short period of time, during which a software cache for each data structure and thread is simulated to count the number of cache hits. When the monitoring phase terminates, the *CLN* data structures with the highest cache hit counts will be marked as cached. The code required for the monitoring phase is injected into existing applications using straightforward compiler transformations.

### 3.2. Code Transformations

The compiler transforms the main loop(s) of the GPU kernel into two loop slices. The first loop slice is used for the monitoring phase, where the computation is executed for a short period of time using the cache simulator. After the first loop slice terminates, the data structures are ranked based on their corresponding cache hit counts, and the top *CLN* data structures are selected to be cached in S-L1. The second loop slice then executes the remainder of the computation using S-L1 for the top *CLN* data structures.

The following example shows a sample transformation:

```
//Some initialization
for(int i = start; i < end; i ++) {
    char a = charInput[i];
    int b = intInput[i];

    int e = doComputation(a, b);
    intOutput[i] = e;
}
//Some final computation
```

is transformed into:

```
//Some initialization
cacheConfig_t cacheConfig;
int i = start;

//slice 1: monitoring phase
for(; (i < end) && (counter<THRESHOLD); i ++) {
    char a = charInput[i];
    simulateCache(&charInput[i], 0, &cacheConfig);
    int b = intInput[i];
    simulateCache(&intInput[i], 1, &cacheConfig);

    int e = doComputation(a, b);
    intOutput[i] = e;
    simulateCache(&intOutput[i], 2, &cacheConfig);
}
calculateWhatToCache(&cacheConfig, availNumCacheLines);
//slice 2: rest of the computation
for(; i < end; i ++)
{
    char a = *((char*) accessThroughCache(&charInput[i], 0,
                        &cacheConfig));
    int b = *((int*) accessThroughCache(&intInput[i], 1,
                        &cacheConfig));

    int e = doComputation(a, b);

    *((int*) accessThroughCache(&intOutput[i], 2,
                &cacheConfig)) = e;
}
flush(&cacheConfig);
//Some final computation
```

[4] In this context, each argument to the GPU kernel that points to data is referred to as a data structure. For example, matrix multiply might have three arguments a, b and c referring to three matrices; each is considered a data structure.

[5] In principle, multiple cache lines could be assigned to a data structure, but we found this does not benefit the streaming applications we are targeting.

**3.2.1. Monitoring phase** In the monitoring loop, a call to simulateCache() is inserted after each memory access. This function takes as argument the address of the memory being accessed, a *data structure identifier*, and a reference to the *cacheConfig* object, which stores all information collected during the monitoring phase. The *data structure identifier* is the identifier of the data structure accessed in the corresponding memory access, and is assigned to each data structure statically at compile time.

The pseudo code of simulateCache() is listed below. This function keeps track of which data is currently being cached in the cache line, assuming a single cache line is allocated for each thread and data structure, and it counts the number of cache hits and misses that occurred. To do this, cacheConfig contains, for each data structure and thread, an address variable identifying the memory address of the data that would currently be in the cache, and two counters that are incremented whenever a cache hit or miss occurs, respectively. On a cache miss, the address variable is updated with the memory address of the data that would be loaded into the cache line.

```
simulateCache(addr, accessId, cacheConfig) {
    addr /= CACHELINESIZE;

    if(addr == cacheConfig.cacheLine[accessId].addr)
        cacheConfig.cacheLine[accessId].hit ++;
    else {
        cacheConfig.cacheLine[accessId].miss ++;
        cacheConfig.cacheLine[accessId].addr = addr;
    }
}
```

The monitoring phase is run until sufficiently many memory accesses have been simulated so that the behavior of the cache can be reliably inferred. To do this, we simply count the number of times simulateCache() is called by each thread; once it reaches a predefined threshold for each thread, the monitoring phase is exited. This pre-defined threshold is set to 300 in our current implementation.[6]

**3.2.2. Determining what to cache** In the general case, we mark the *CLN* data structures with the highest cache hit counts to be cached in S-L1. However, there are two exceptions. First, we distinguish between read-only and read-write data structures. Read-write data structures incur more overhead, since dirty bits need to be maintained and dirty lines need to be written back to memory. Hence, we give higher priority to read-only data structures when selecting which structures to cache. Currently, we select a read-write data structure over a read-only data structure only if its cache count rate is twice that of the read-only data structure, because accesses to read-write cache lines involve the execution of twice as many instructions on average.

Secondly, in our current implementation, we only cache data structures if it has a cache hit rate above 50%. A hit rate of more than 50% means that, on average, the cache lines are reused at least once after loading the data due to a miss. We do this because otherwise the overhead of the software implementation will not be amortized by faster memory accesses.

**3.2.3. Computation phase** In the second loop slice, the compiler replaces all memory accesses with calls to accessThroughCache(). This function returns an address, which will either be the address of the data in the cache, or the address of the data in memory, depending on whether the accessed data structure is cached or not. A simplified version of the accessThroughCache() is as follows:

```
void* accessThroughCache(void* addr, int accessId,
              cacheConfig_t* cacheConfig)
{
    if(cacheConfig.isCached[accessId] == NOT_CACHED) {
        return addr;
    }
    else {
        //If already cached, then simply return the
        //address within the cache line
        if(alreadyCached(addr, cacheConfig.cacheLine[accessId])) {
            return &(cacheConfig.cachelines[accessId].
                    data[addr % 16]);
        }
        //requested data is not in the cache, so,
        //before caching it we need to evict current data.
        else {
            //If not dirty, simply overwrite. If dirty,
            //first dump the dirty data to memory

            if(cacheConfig.cachelines[accessId].dirty) {
                dumpToMemory(cacheConfig.cachelines[accessId]);
            }
            loadNewData(addr, cacheConfig.cachelines[accessId]);
            return &(cacheConfig.cachelines[accessId].
                    data[addr % 16]);
        }
    }
}
```

If the data of a cached data structure is not found in the cache line, then the current contents of the cache line is evicted before new data is loaded into the cache line from memory. When evicting a line, its dirty bit is checked if the corresponding data structure is read-write. If not set, the cache line is simply overwritten by the new data. If set, the modified data is written back to memory before loading the new data. We keep a bitmap (in registers) to identify which portions of the cache line are modified, so that only those need to be written back; this also guarantees that if two different threads cache the same line (in different S-L1 lines) and modify different potions of the cache line, they will not overwrite each others data. Atomic bitwise operations use the bitmap as a write mask to perform the data write-backs.

A call to flush() is inserted after the second loop slice to flush the modified cache lines to memory and invalidate all cache lines before the application terminates.

Extra overhead can be avoided if the pointers to data structures, provided as arguments to the GPU kernel, are not aliased. Programmers can indicate this is the case by including the restrict keyword with each kernel argument. If this keyword is not included then the caching layer will still work properly, albeit with extra overhead because it has to be assumed that

[6] This method of statically setting the duration of monitoring phase works well for regular GPU applications such as the ones we are targeting, but more sophisticated methods may be required for more complex, irregular GPU applications. Moreover, while we only run the monitoring phase once, it may be beneficial to enter into a monitoring phase multiple times during a long running kernel to adapt to potential changes in the caching behavior.

the pointers may be aliased, in which case the caching layer will need to perform data lookups in all cache lines assigned to the same thread for each memory access — even for memory accesses to uncached data structures.

### 3.3. S-L1 overheads

When evaluating S-L1, the following overheads need to be considered:

**Monitoring phase:** Our experiments show that the performance overhead of the monitoring phase is relatively low — an average of less than 1% was observed in the 10 applications we experimented with (see Section 4.5). The overhead is low because the monitoring phase only runs for a short period of time and because the code of simulateCache() is straightforward and typically does not incur additional memory accesses since all variables used in simulateCache() are located in statically allocated registers. In terms of register usage, the monitoring phase requires three registers per data structure/simulated cache line: one for the mapped address of the cache line in memory and two to keep the cache hit and miss counters. These registers are only required during the monitoring phase and will be reused after the phase terminates.

**calculateWhatToCache():** The performance overhead of calculateWhatToCache() is negligible since it only needs to identify which *CLN* data structures have the highest hit counts, and typically, the applications access only a few data structures.

**accessThroughCache():** Most of the overhead of the caching layer occurs in this function. For accesses to data structures that are not cached, the performance overhead entails the execution of four extra machine instructions. However, accesses to cached data structures incur more significant overhead since a large body of code needs to be executed in some cases; e.g., when evicting a cache line. Our experiments indicate that the caching layer increases the number of instructions issued by 25% on average over the course of the entire application (see Section 4.5). This overhead can indeed negatively impact the overall performance of an application if it is not amortized by the lower access times offered by S-L1, and the overhead is exacerbated if the application's throughput is already limited by instruction-issue bandwidth.

In terms of register usage, accessThroughCache() requires three additional registers per data structure and thread: one for the memory address of the data currently being cached, one for the write bitmap (which also serves as the dirty bit), and one for the data structure identifier. (If the data structure is not cached, the value of the last register will be -1). As an optimization, we do not allocate bitmap registers for read-only data structures. Additionally, since data structures that are not cached do not access the bitmap and address registers, the compiler might spill them to memory, without accessing them later, thus reducing the register usage of uncached data structures to 1. The recent GPU architectures (e.g. *Kepler*) have 65,535 registers per SMX and can support at most 2,048 threads, in which case the S-L1 caching layer would, in the worst case, use up to 6% and 9% of the total number of available registers for cached read-only and read-write data structures, respectively.

### 3.4. Coherence considerations

Since each thread has its own private cache lines, cached data will not be coherent across cache lines of different threads. Thus, if two threads write to the same data item cached separately, the correctness of the program might be compromised. Fortunately, the loose memory consistency model enforced by GPUs makes it easy to maintain the required level of consistency for cache accesses. We follow two simple rules to maintain the correctness of the program: (a) we flush the entire cache on *memory fence instructions* and (b) we evict the cache lines containing data targeted by atomic instructions before the instruction executes.

Executing a *memory fence instruction* enforces all memory writes that were performed before the instruction to be visible to all other GPU threads before the execution of the next instruction. GPGPU programmers are required to explicitly use these instructions if the application logic relies on a specific ordering of memory reads/writes. We implement this by inserting a call to flush() immediately before each memory fence instruction, which flushes the contents of the modified cache lines to memory and invalidates the cache lines by setting their associated mapped addresses to zero.

By executing an *atomic instruction*, a thread can read, modify, and write back a data in GPU memory atomically. We insert an overloaded version of flush() before each atomic operation. The overloaded version of flush() has two arguments: the address of the memory location accessed by the atomic operation and the associated data structure identifier. It flushes the cache line if (and only if) the cache line contains the data pointed to by address [8].

## 4. Experimental Evaluation

### 4.1. Experimental Setup

Unless otherwise noted, all GPU kernels used to evaluate S-L1 were executed on an Nvidia GeForce GTX 680 GPU connected to 2GB of GPU memory with a total of 1,536 computing cores running at 1020MHz. As described in Section 2.1, the GTX 680 is from the Kepler family and has 8 multiprocessors, each with 192 computing cores, and 64KB of on-chip memory (of which 48KB is assigned to shared memory).

All GPU-based applications were implemented in CUDA, using CUDA toolkit and GPU driver release 6.0.37 installed on a 64-bit Ubuntu 12.04 Linux with kernel 3.5.0-23. All applications are compiled with the corresponding version of the *nvcc* compiler using optimization level three.

For each experiment, we ran the target application using different thread configurations, and only considered the configuration with the best execution time for reporting and compar-

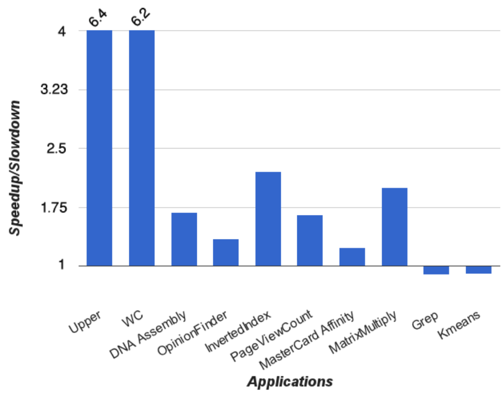


**Figure 7:** Speedup when using S-L1 relative to no L1 caching.

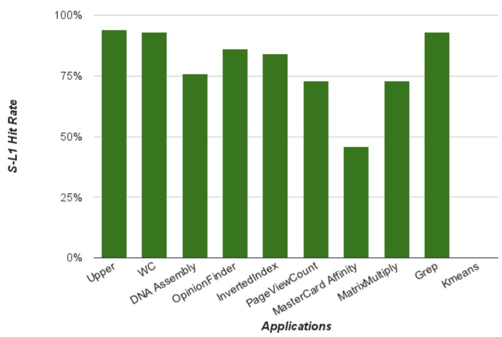


**Figure 8:** S-L1 hit rate.

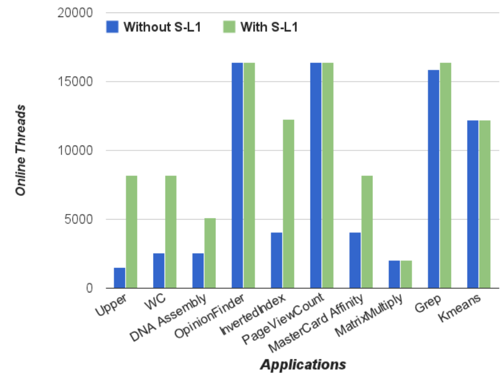


**Figure 9:** The optimal number of online threads (that leads to the best execution times) with and without S-L1.

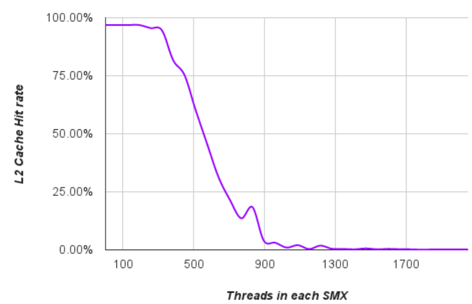


**Figure 10:** L2 hit rate for wc.

ison purposes.[7] Specifically, we tested each application using 512 different thread configurations, starting with 4 blocks of 128 threads (for a total of 512 threads) and increased the number of threads in 128 increments, up to 256 blocks of 1024 threads (for a total of 256K threads).

### 4.2. S-L1 performance evaluation for GPU-local applications

We applied S-L1 to the ten streaming applications listed in Table 1. There is no standard benchmark suite for GPU streaming applications, so we selected 6 representative applications, 2 simple scientific applications (MatrixMultiply and Kmeans) to see how well S-L1 works on them, and 2 extreme applications to stress test S-L1: wc, which has minimal computation (only counter increments) for each character access, and upper, which is similar to wc but may modify the characters. For each experiment, the data accessed by the applications was already located in GPU memory.

Figure 7 shows the performance of our 10 benchmark streaming applications when run with S-L1 relative to the performance of the same applications run without S-L1 (and also without the hardware L1). On average, these applications run 2.45X faster than their non-caching counterparts. Some (e.g., upper and wc) run multiple times faster, while others (e.g., grep and Kmeans) experience slight slowdowns.

The benefits obtained from S-L1 depends on a number of factors. First, the attained cache hit rate obviously has a large effect. Figure 8 depicts the S-L1 hit rate for all benchmarks. Overall, the hit rate is quite high, in part because most of the applications have high spatial locality (which is to be expected for streaming applications). As an extreme example, consider wc, where each thread accesses a sequence of adjacent characters, so each S-L1 miss is typically followed by 15 hits, given a 16 byte cache line. Kmeans is an exception: because the application allocates much of the shared memory for its own purposes, there is insufficient space for S-L1 cache lines, and hence the effective S-L1 hit rate is zero for this application.[8]

A second factor is the memory intensity of the applications; i.e., the ratio of memory access instructions to the total number of instructions executed. Some applications (e.g., upper and wc) are memory bound and hence benefit from S-L1. At the other extreme, grep performs worse despite having a high cache hit rate, because it is compute intensive with its recursive algorithm and because it has a large number of branches with significant thread divergence. The benefits of the caching layer is negated by the extra instructions that need to be executed because of the software implementation of S-L1.

A third factor is the degree to which S-L1 enables extra thread parallelism, thus improving the utilization of the GPU

[7] GPGPU programmers typically experimentally run their applications with different thread configurations to determine the optimal number of threads and from then on run that configuration.

[8] Because Kmeans allocates space in shared memory dynamically at run time, the compiler cannot know that there is not enough space for S-L1 — otherwise it potentially could have avoided adding the code required for S-L1.

| Application | Description | Used number of data structures |
|---|---|---|
| Upper | Converts all text in an input document from lowercase to uppercase. | 2 |
| WC | Counts the number of words and lines in an input document. | 1 |
| DNA Assembly | merges fragments of a DNA sequence to reconstruct a larger sequence [3]. | 3 |
| Opinion Finder | analyzes the sentiments of tweets associated with a given subject (i.e. a set of given keywords) [24] | 4 |
| Inverted Index | Builds reverse index from a series of HTML files. | 3 |
| Page View Count | Counts the number of hits of each URL in a web log. | 3 |
| MasterCard Affinity | finds all merchants that are frequently visited by customers of a target merchant X [13] | 3 |
| Matrix Multiply | Calculates the multiplication of two input matrices. This is a naive version and does not use shared memory. | 3 |
| Grep | Finds the string matching a given pattern and outputs the line containing that string. | 2 (1 in shared memory) |
| Kmeans | Partitions *n* particles into *k* clusters so that particles are assigned to the cluster with the nearest mean. | 2 (1 in shared memory) |

**Table 1:** Ten streaming applications used for experimental performance evaluation, their description, and the number of data structures they use in their main lc

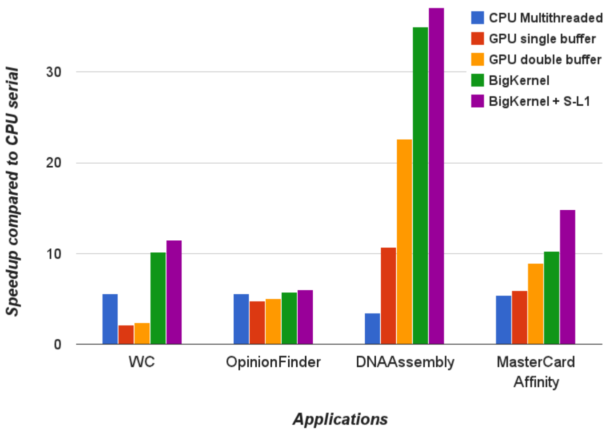


**Figure 11:** Speedup of various implementations over CPU serial version for four GPU applications processing large data sets located in CPU memory.

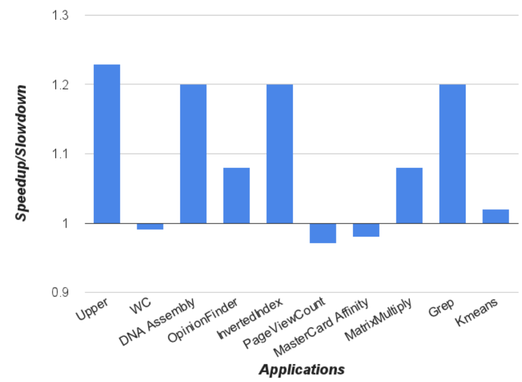


**Figure 12:** Speedup obtained when using the hardware L1 cache on the older GTX 560.

cores. Figure 9 shows the number of *online* threads[9] that result in the best performance for each application with and without S-L1. With S-L1, applications can run with more threads without having to worry about thrashing the S-L1. Without S-L1, applications typically need to limit the degree of parallelism to prevent L2 cache thrashing. For example, Figure 10 shows the L2 cache hit rate of WC as a function of the number of threads per SMX, where the cache hit rate drops to less than 1% at 1,024 threads.

### 4.3. Evaluation of S-L1 for data residing in CPU memory

For big data-style applications, the data will not fit in GPU memory because of the limited memory size. Hence, in this subsection, we consider the performance of four applications with data sets large enough to not fit in GPU memory. We ran these applications under five different scenarios:

1. CPU multithreaded when run on a 3.8GHz Intel Xeon Quad Core E5 connected to 16GB of quad-channel memory clocked at 1.8 GHz;
2. GPU using a single buffer to transfer data between CPU and GPU;
3. GPU using state-of-the-art double buffering to transfer data between CPU and GPU;
4. GPU using BigKernel [13]; and
5. GPU using BigKernel combined with S-L1.

We selected to combine S-L1 with BigKernel in particular, because BigKernel is currently the best performing system for data intensive GPU streaming applications [13].

Figure 11 shows the results. For all four applications, using BigKernel combined with S-L1 performs the best, and for all but one application the performance is an order of magnitude better than the multithreaded CPU version. Compared to BigKernel alone, BigKernel combined with S-L1 is 1.18X faster. The primary reason preventing BigKernel from performing better when combined with S-L1, we observed, is that BigKernel uses many registers and therefore, its parallelism is limited[10]. We are still working on optimizing this.

### 4.4. Comparing with hardware L1

An interesting question is how our benchmarks would perform if hardware L1 were available. To answer this question, we executed our applications on an older GPU, the Nvidia GTX 560.[11] Figure 12 shows the performance obtained with the L1 relative to running the applications with L1 disabled. Overall, the performance gains with the L1 are limited to under 25% and in some cases result in minor slowdowns. We would

[9] I.e., threads that run at the same time on all multiprocessors, the maximum of which can be 16K threads on our GPU.

[10] When a kernel uses high number of registers, an SMX will schedule less number of *online threads* to be able to provide them with the required number of registers.

[11] Because the L1 cache of GTX 680 is disabled for application data by the vendor, we ran our experiments described in this section on an Nvidia GTX 560, which only has 336 computing cores connected to 1GB of GPU memory. The GTX 560 is a Fermi GPU containing 7 multiprocessors, each with 48 computing cores and 64kB of on-chip memory. For the experiments, the L1 cache size was configured to be 48KB.

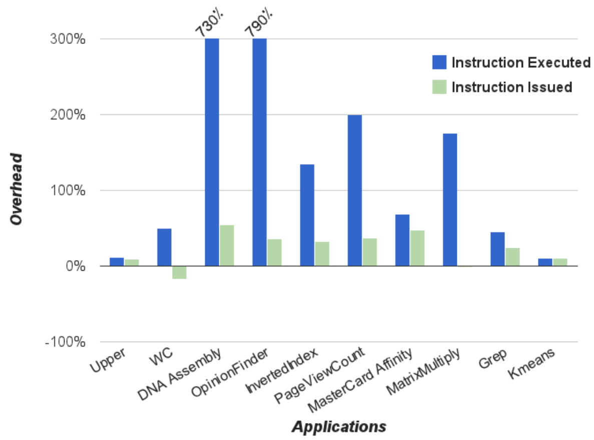


**Figure 13:** Extra instruction executed and issued (in %) due to S-L1.

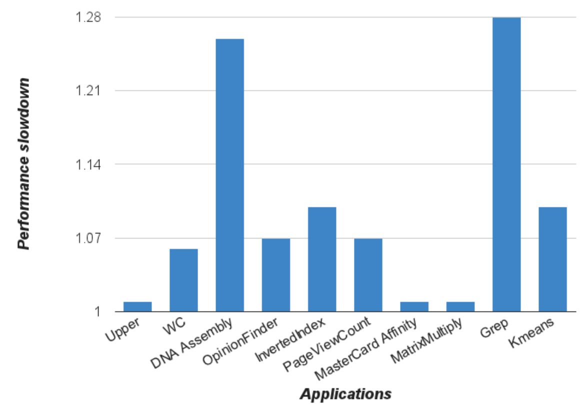


**Figure 14:** Overhead of S-L1 when S-L1 is enabled but not used to cache any data.

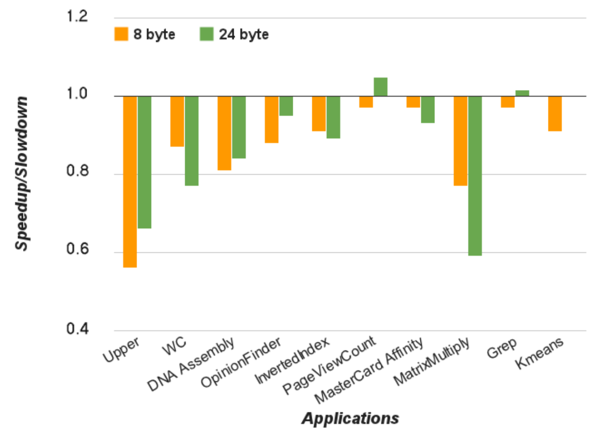


**Figure 15:** Slowdown/speedup when using 8B and 24B cache lines over using 16B cache lines.

expect that the performance with a hardware L1 cache for application data on the GTX 680 would be far worse, because it has many more cores (192 vs. 48) but the same L1 cache size, which would result in increase L1 cache thrashing.

Two further interesting observations are worth noting. First, unlike with S-L1, the hardware L1 cache increases the performance of grep reasonably well. We attribute this to the fact that the hardware L1 incurs no extra instruction overhead, which for grep is significant because it is already computationally intensive.

Secondly, wc, PageViewCount, and MasterCard Affinity exhibit slowdown when L1 is enabled. We attribute this to the added memory transfers due to L1's large 128 byte cache lines (which is 4 times larger than L2's cache line size). This phenomenon was originally observed by by Jia et al. [8].

### 4.5. S-L1 overheads

S-L1 has significant overhead because it is implemented in software and extra instructions need to be executed for each memory access: a minimum of 4 and potentially well over 100 extra instructions per memory access. Figure 13 depicts the increase in the number of instructions, both executed and issued, under S-L1. Executed instructions are the total number of instructions completed, while issued instructions also count the times an instruction is "replayed" because it encountered a long latency event such as a memory load.

The increase in the number of executed instructions is significant: 220% on average. The reason is obvious: each memory access instruction is transformed to additionally call a function that needs to be executed. On the other hand, the increase in the number of issued instructions is more reasonable: 25% on average. (For wc and MatrixMultiply the number of issued instructions actually decreases.) The reason issued instructions increase less than executed instructions is that S-L1 provides for improved memory performance, which reduces the number of required instruction replays.

To evaluate the overhead S-L1 introduces for data structures that are not cached, we ran our benchmarks with S-L1, but with all data structures marked as non-cacheable. Figure 14 shows the overhead incurred in this case: 8% on average. Based on our experiments, the monitoring phase accounts for less than 1% of this overhead. The rest of the overhead is attributed to executing the memory access function that is called for each memory access. As suggested in Section 4.2, one potential way to avoid this overhead is have the compiler not transform memory accesses to data structures that are found not worthy of caching – e.g. a data structure that is statically known not exhibit any caching benefit.

### 4.6. Effect of S-L1 cache line size

Figure 15 compares the overall performance of applications when using different variations of S-L1 using different cache line sizes. Specifically, we show the performance improvement/loss for 8-byte and 24-byte cache lines over 16-byte cache lines.[12]

In most cases, 16-byte cache lines seems to be best choice. As we described in Section 3.1, we believe this is mainly because 16-bytes is the widest available load/store size on

[12] We did not choose 32 as a potential cache line size since it is more than the size of shared memory that will be assigned to a threads, if SMXs are fully occupied (i.e. 2048 threads), given the maximum size of shared memory (i.e. 48KB).

GPU ISA and hence, the entire cache line can be read/written with one memory access.

Decreasing the cache line size to 8-bytes impacts performance negatively in every case, since the cache then typically needs to execute the inserted memory access function twice as often for a fixed amount of streaming data to be processed by the application. Note that our benchmarks primarily consist of streaming applications that have high spatial locality and that consume most of the data in a cache lines.

Increasing the cache line size to 24-bytes also reduces performance in all but two cases, mainly because 24 bytes do not provide much additional benefit over 16 bytes, yet require two memory accesses to fill a cache line instead of one. For example to process 48 characters accessed sequentially, a 16B cache line results in 3 misses and thus 3 L2/DRAM accesses, whereas a 24B cache line results in 2 misses and thus 4 L2/DRAM accesses.

## 5. Related work

A large body of work focuses on using shared memory to increase the performance of applications in an application centeric way [4, 18, 17, 22, 15, 11, 10]. For instance, Nukada et al. propose an efficient 3D FFT that uses shared memory to exchange data efficiently between threads [14].

Other work studies more general approaches to harness the benefits of shared memory, typically by providing libraries or compile time systems that use shared memory as a scratch-pad to optimize the memory performance of applications [1, 26, 20, 9, 12, 7]. For instance CudaDMA provides a library targeting scientific applications that allows the programmer to stage data in shared memory and use the data from there [2]. A producer consumer approach is proposed where some warps only load the data in shared memory (producer) and others only do the computation (consumer). What we achieve with S-L1 can also be achieved with CudaDMA, however in CudaDMA the programmer is responsible to use the API manually, set the number of producer and consumer threads, and assign the proper size of shared memory to different threads.

Yang et al. propose a series of compiler optimizations, including vectorization and data prefetching, to improve the bandwidth of GPU memory [26]. In particular, they provide a technique in which uncoalesced memory accesses are transformed to coalesced ones using shared memory for staging.

Others have also studied the characteristics of GPU memory and GPU caches [25, 5]. Jia et al. characterize GPU L1 cache locality in Nvidia GPUs and provide a taxonomy for reasoning about different types of access patterns and how they might benefit from L1 caches [8]. Tore et al. provides insights into how to tune the configuration of GPU threads to achieve higher cache hit rates and also offers an observation on how the L1 impacts a handful of simple kernels [19].

Finally, some studied the potential architectural changes that could improve the GPU caching behavior, including a recent study that analyzes potential coherent models for GPU L1 caches [16].

## 6. Concluding Remarks

By reverse-engineering the Nvidia GTX 680 through a series of experiments, we characterized the behavior of the memory hierarchy of modern GPUs. We showed that the bandwidth between off-chip memory and GPU SMXs is capped so that the latency of L2/DRAM accesses increases substantially the more memory intensive the application. We also showed that raw GPU compute power has been increasing more rapidly than the size of on-chip caches. As a result, there is a growing need for effective, on-chip caching to reduce the number of memory accesses that need to leave each multiprocessor.

In this paper, we proposed S-L1, a GPU level 1 cache which is implemented entirely in software using SMX shared memory. S-L1 determines, at run time, the proper size of cache, samples the effectiveness of caching the data of different data structures, and based on that information, decides what data to cache. Although the software implementation adds on average 8% overhead to the applications we tested, our experimental results show that this overhead is amortized by faster average memory access latencies for most of these applications. Specifically, S-L1 achieves speedups of between 0.89 and 6.4 (2.45 avg) on ten GPU-local streaming applications. Combining S-L1 with BigKernel, the fastest known technique accelerating GPU applications processing large data sets located in CPU memory, leads to speedups between 1.07 and 1.45 (1.18 avg.) over BigKernel alone, and speedups bewteen 1.07 and 11.24 (4.32 avg.) over the fastest CPU multicore implementations.

While it is understandable that GPU designers need to prioritize optimizations for graphical processing and maintain commodity pricing, we believe that our work provides some indications of how the GPU designers could enhance current designs to make GPU designs more effective for data intensive GPGPU applications. The most straightforward enhancement is to significantly increase the size of the L2 — with its current size it only supports 1-2 cache lines per thread when applications run with the maximum number of online threads allowed. We also believe that a design similar to our S-L1 design could be implemented in hardware in a reasonably straightforward way, which would almost entirely eliminate the overhead that our software entails. Finally, the size of on-chip caches need to be substantially increased, especially as the number of cores continues to increase. And perhaps L1 caches line sizes should be configurable so that smaller sizes can be accommodated.

For future work, we intend to reduce the overhead of S-L1 by relying more on the compile-time technology. Using compiler technology, we can avoid transforming memory accesses to data structures that are statically known to exhibit poor caching behavior. Moreover, if accesses to all data structures can be statically analyzed, the monitoring phase might also become unnecessary.

## References


[1] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS '08)*, pages 225–234, 2008.

[2] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU Memory Bandwidth Via Warp Specialization. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 12, 2011.

[3] J.A. Chapman, I. Ho, S. Sunkara, S. Luo, G.P. Schroth, and D.S. Rokhsar. Meraculous: De Novo Genome Assembly with Short Paired-End Reads. *PLoS ONE*, page e23501, 2011.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization (IISWC 2009)*, pages 44–54, 2009.

[5] Jayesh Gaur, Raghuram Srinivasan, Sreenivas Subramoney, and Mainak Chaudhuri. Efficient Management of Last-level Caches in Graphics Processors for 3D Scene Rendering Workloads. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 46)*, pages 395–407, 2013.

[6] C. Gregg and K. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Proceedings of IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 134–144, 2011.

[7] Feng Ji and Xiaosong Ma. Using Shared Memory to Accelerate MapReduce on Graphics Processing Units. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2011)*, pages 805–816, 2011.

[8] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing (ICS 26)*, pages 15–24, 2012.

[9] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. A Script-based Autotuning Compiler System to Generate High-performance CUDA Code. *ACM Transactions on Architecture and Code Optimization (TACO)*, pages 31:1–31:25, 2013.

[10] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*, pages 239–252, 2014.

[11] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and J-M Shyu. Accelerating String Matching Using Multi-threaded Algorithm on GPU. In *Global Telecommunications Conference (GLOBECOM 2010)*, pages 1–5, 2010.

[12] Maryam Moazeni, Alex Bui, and Majid Sarrafzadeh. A Memory Optimization Technique for Software-managed Scratchpad Memory in GPUs. In *IEEE 7th Symposium on Application Specific Processors (SASP'09)*, pages 43–49, 2009.

[13] Reza Mokhtari and Michael Stumm. BigKernel – High Performance CPU-GPU Communication Pipelining for Big Data-Style Applications. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*, pages 819–828, 2014.

[14] A Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2008)*, pages 1–11, 2008.

[15] Christopher I Rodrigues, David J Hardy, John E Stone, Klaus Schulten, and Wen-Mei W Hwu. GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications. In *Proceedings of the 5th conference on Computing frontiers*, pages 273–282, 2008.

[16] Inderpreet Singh, Arrvindh Shriraman, Wilson Fung, Mike O'Connor, and Tor Aamodt. Cache Coherence for GPU Architectures. 2013.

[17] Marc A Suchard, Quanli Wang, Cliburn Chan, Jacob Frelinger, Andrew Cron, and Mike West. Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures. *Journal of Computational and Graphical Statistics*, pages 419–438, 2010.

[18] Jonas Tölke and Manfred Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, pages 443–456, 2008.

[19] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R Llanos. Understanding the Impact of CUDA Tuning Techniques for Fermi. In *2011 International Conference on High Performance Computing and Simulation (HPCS)*, pages 631–639, 2011.

[20] Sain-Zee Ueng, Melvin Lathara, Sara S Baghsorkhi, and W Hwu Wen-mei. CUDA-lite: Reducing GPU Programming Complexity. In *Languages and Compilers for Parallel Computing*, pages 1–15. 2008.

[21] Manuel Ujaldon. Inside kepler. http://gpu.cs.uct.ac.za/Slides/Kepler.pdf, 2013.

[22] John Paul Walters, Vidyananth Balu, Suryaprakash Kompalli, and Vipin Chaudhary. Evaluating the Use of GPUs in Liver Image Segmentation and HMMER Database Searches. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009)*, pages 1–12, 2009.

[23] Wikipedia. Geforce 700 series — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/GeForce_700_series, 2013.

[24] T. Wilson, P. Hoffmann, S. Somasundaran, J. Kessler, J. Wiebe, Y. Choi, C. Cardie, E. Riloff, and S. Patwardhan. OpinionFinder: a System for Subjectivity Analysis. In *Proceedings of HLT/EMNLP on Interactive Demonstrations*, pages 34–35, 2005.

[25] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A Moshovos. Demystifying GPU Microarchitecture Through Microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 235–246, 2010.

[26] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, pages 86–97, 2010.