Resurrecting L1: A Software Cache for GPUs

Abstract

In this paper we propose a new method of caching for GPUs.

1. Design and Implementation

The design of our software cache is based on three key ideas to minimize cache thrashing:

1. Thread-private cache lines.
2. Smaller cache lines.
3. Selective caching.

The cache space is partitioned into thread-private cache segments, each containing one or more cache lines. The number of cache lines in each cache segment is determined dynamically at runtime based on the available size of shared memory per thread and the size of the cache line. The decision to have thread-private cache segments was made because there is little or no inter-thread data sharing in streaming applications. In these applications, threads process data records independently and hence typically access disjoint locations of memory. Therefore, allowing all threads to share the entire space of cache in such applications only results in high collision misses, quickly exhausting the cache space.

The use of smaller cache lines allows for more cache lines to be squeezed in the limited cache space. Choosing a proper size of cache line in our software cache follows a trade-off, where a smaller cache line diminishes the benefit of caching aimed for spatial locality and a larger cache line increases the risk of wasting memory bandwidth (due to potentially more memory transactions required to load/store each cache line) and also increases the chance of thrashing the cache (due to reduced number of available cache lines). We performed several experiments to determine a proper cache line size and found 16-bytes to be more effective, mainly because 16-bytes is the widest load/store size available on modern GPU architectures – which allows us to load/store the entire cache line in one memory access – and also because it falls within the typical range of shared memory size assigned to each thread.[1] Section **??** shows the results of an experiment that compares the effectiveness of various cache line sizes.

The maximum number of cache lines assigned to each thread, denoted by *CLN*, is determined at runtime, and is calculated as $[(shMemSizePerSM/numThreadsPerSM)/cacheLineSize]$, where *shMemSizePerSM* is the available size of shared memory per multiprocessor, *numThreadsPerSM* is the number of threads that are allocated on each multiprocessor, and *cacheLineSize* is the size of the cache line, i.e. 16 bytes in our current design. The available size of shared memory is calculated at runtime by probing the hardware to determine how much shared memory is available on the running GPU and subtracting from it the portion that is already in use by the application.[2] *NumThreadsPerSM* is also calculated at runtime, in part using the configuration the programmer specifies at kernel invocation. Once the number of cache lines per thread – *CLN* – is determined, up to that number of data structures are marked and a separate cache line is assigned to each. 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* (perhaps because the shared memory is extensively in use by the application code), no cache lines are assigned to threads (i.e. $CLN = 0$) and the caching layer becomes disabled.

To determine which data structures to select, the benefit of caching the data of each data structure is evaluated using a short monitoring phase at runtime and only the top *CLN* data structures that showed the highest caching benefit will be selected. In the monitoring phase, the core computation of an application is executed for a short time while simulating giving a cache line to each data structure, and measuring how well the accesses to that data structure will benefit from being cached. To measure the caching benefit, each cache line is associated with a hit counter that counts the number of cache hits to that cache line. When the monitoring phase terminates, cache lines with higher number of cache hits will be evaluated as more effective and their corresponding data structures are marked to be cached. The code required for monitoring phase is injected into existing applications using straightforward compiler transformations.

1.1. Code Transformations

Compiler takes the main loop(s) of an application as its core computation, and breaks it up into two loop slices. The first loop slice, performing the monitoring phase, performs the computation for a short time while simulating the cache hits. After the first loop slice terminates, the data structures are ranked based on their associated cache hits. The second loop slice then performs the rest of the computation while using *CLN* separate cache lines for the memory accesses of the data

[1]The most recent GPU hardware has 48KB of shared memory per multiprocessor. Given that this GPU can schedule from 1-2048 threads on each multiprocessor, each thread could eventually be assigned from 48KB-24B of shared memory.

[2]The portion of shared memory in use by the application must be determined using a mixed compile-time and runtime approach, as programmers can allocate shared memory both statically (which could be determined at compile-time) and dynamically (which is only known at runtime).

structures that exhibited the highest cache hits.

The following example shows how the transformations are performed.

```
//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
```

Will be 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);

}
calculateWhatDataToCache(&cacheConfig, availNumCacheSegments);

//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
```

Monitoring phase: in the monitoring phase, a call to simulateCache() is inserted after each memory access. This function takes as argument the address of the memory access, a *data structure identifier*, and the reference to the *cacheConfig* object, which stores all information regarding the monitoring phase, including the cache hit counts and also the information regarding what data structures to cache. The *data structure identifier* is the identifier of the data structure accessed in the corresponding memory access. This identifier is assigned to each data structure statically at compile time; the compiler identifies individual data structures by extracting the pointers from kernel arguments.

The pseudo code of simulateCache() is listed below. This function calculates cache hit counts when assigning a single cache line per thread to each data structure. To do that, the cacheConfig object keeps a separate address variable for each data structure and thread which holds the memory address that an imaginary cache line is mapped to. A cache hit is recorded whenever a memory access falls within the *cacheLineSize* of the corresponding address. Otherwise, the address variable is update with the address of the new memory access – simulating a cache miss.

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

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

Monitoring phase duration: we run the monitoring phase until sufficiently many memory accesses are simulated so that the behavior of the cache can be reliably inferred. To do this, we augmented the simulateCache() function to also count the number of times it is called by each thread. Once this number reaches to a predefined threshold, a flag is set by the simulateCache() and the corresponding thread exits the monitoring loop. This pre-defined threshold is set to 300 in our current design.

While this method of statically setting the duration of monitoring phase works well for most of the existing, regular GPU applications, we believe that more sophisticated methods will be required for more complex and irregular GPU applications. Moreover, while we only run the monitoring phase once in the beginning of the kernel execution, a more sophisticated method might require running this phase multiple times during a long running kernel to adapt to potential changes in caching behavior that might result from changes in the runtime data.

Finally, a call to flush() is inserted right after the loop. This function simply flushes the modified cache lines to memory and sets the mapping address associated with all cache line to zero as a way to invalidate all the cache lines.

Determining what to cache

Writable cache lines: we distinguish between read-only and read-write data structures when deciding which data structures to cache. In general, caching the data of read-only data structures are easier and brings more performance gain, since the cache lines do not get modified and therefore, can be discarded when evicted, whereas the cache lines assigned to read-write data structures might get modified and need to be written back to memory. Therefore, the caching layer is designed to give a higher priority to read-only data structures when it comes to selecting which ones to cache. Currently, we select a read-write data structure over a read-only data structure only if its cache hit counts is two times higher than the cache hit counts of the read-only data structure. This is mainly because the cache management of a writable cache line requires two times more instruction execution, on average.

Caching the marked data structures: In the second loop slice, the compiler replaces all memory accesses with calls to accessThroughCache(). This function returns an address, which might be the address of the data in the cache, or the address of it in the memory. For memory accesses to data structures that are not marked, the accessThroughCache() returns the corresponding address in memory, causing the memory access to be performed as if no caching layer exists. However, for memory accesses to data structures that are marked to be cached, the address of the data in cache will be returned. Note that, what is described here is the C-level representation of the transformations. In the actual code transformations, the compiler in-lines calls to accessThroughCache(). Therefore, instead of first calling accessThroughCache() and then using the addresses it returns, the data is directly accessed.

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

Pointer aliasing: to increase the performance of the caching layer, we need the pointers in GPU kernel arguments to not be aliased – so that no two pointers point to the same data structure. Therefore, we require programmers to provide a restrict keyword with each kernel argument that points to a different data structure. Note that, if this keyword is not provided, the caching layer still works, but its performance might be lowered. The reason for a lowered performance is that, with possible pointer aliasing, our caching layer needs to perform data lookups in all assigned cache lines for all memory accesses – even memory accesses to data structures that are not cached – which has an overhead on the performance. However, when kernel arguments are guaranteed to not be aliased, memory accesses to a cached data structure are looked up only in the corresponding cache line – and memory accesses to un-cached data structures are not looked up in any cache line.

Cache-line eviction: if, while processing a call to accessThroughCache(), the data is not found in the corresponding cache line, the existing data of the cache line is evicted and new data, containing the requested data, is loaded into the cache line. During the eviction, if a data structure is not read-only, its dirty bit is checked. If the dirty bit is not set, the cache line will be simply overwritten by the new data. If the dirty bit is set, however, the modified data is written back to memory before loading the new data. Additionally, we keep a bitmap to identify which parts of the cache line are modified, so that we only write-back the modified parts of the cache lines to memory. This guarantees that, if two different threads modify different parts of their respective cache lines that mapped to the same memory location, no one of them overwrites the data of the other thread. To do this, atomic bitwise operations use the bitmap as a write mask and perform the data write-backs.

1.1.1. The overheads of the caching layer

- **simulateCache():** our experiments show that the performance overhead of the monitoring phase is relatively low; an average of less than 1% overhead on the entire execution time was observed in our 10 experimental applications (see Section **??**). This is mainly because the code of simulateCache() is straightforward and typically does not incur additional memory accesses – since all variables used in simulateCache() can be stored in statically allocated registers. Furthermore, the monitoring phase is run only for a short time.
  In terms of register usage, the monitoring phase requires two registers per data structure/simulated cache line, one to hold the mapped address of the cache line in memory and another to keep the cache hit counter. These registers are only required during the monitoring phase and will be reused after the phase terminates.
- **calculateWhatDataToCache():** the performance overhead of calculateWhatDataToCache() is negligible since it only sorts the data structures (of which there are typically only a few) based on their cache hit counts and marks the top *CLN* ones to be cached.
- **accessThroughCache():** the heavy lifting of the caching

layer is done in the $\mathrm{accessThroughCache}()$ function. For memory accesses to data structures that are not cached, the performance overhead of this function is negligible since only 3 additional machine instructions are executed [FACTCHECK]. However, for memory accesses to cached data structures, often a large body of code must be executed to manage the cached data (e.g. for cache line eviction). Our experiments show that this accounts for an average of 25% increase in the instruction issue rate in the applications we experimented with. Indeed, this can negatively impact the overall performance of an application if the application's throughput is already limited by instruction-issue bandwidth.

In terms of register usage, the $\mathrm{accessThroughCache}()$ requires three additional registers per data structure. One register holds the corresponding cache line's mapped address in memory, another keeps the write bitmap (the bitmap register also serves as the dirty bit), and last one holds the cache line ID associated with the data structure (if the data structure is not cached, the value of this register will be -1). As an optimization, we do not allocate the bitmap register for read-only data structures. Additionally, since data structures that are not cached do not need to access bitmap and address registers, compiler might be able to spill them to memory, without accessing them later, thus reducing the register usage of un-cached data structures to 1.

Considering a recent GPU architecture (e.g. *Kepler* architecture with 65,535 registers per multiprocessor), in the worst case that the maximum number of threads are resident in a multiprocessor (i.e. 2048 threads), caching layer requires up to 6% and 9% of the total number of available registers in the multiprocessors per read-only and read-write data structures, respectively. For example, an application that has 2 read-only and 2 read-write data structures, needs to put aside up to 31% of registers available to it for the caching layer.

1.1.2. Maintaining the correctness of application Since each thread has its own private cache lines, cached data will not be coherent across cache lines of different threads. While this is fine for read-only data structures, for data structures that also are written to, the correctness of the program might be broken if different threads read and write to memory locations that fall within the same cache line.

We follow two simple rules to maintain the correctness of the program in our caching layer: (a) flushing the entire cache on *memory fence instructions* and (b) not caching the memory locations accessed by atomic instructions.

Executing a *memory fence instruction* enforces all memory writes that were performed before the instruction to be visible by all other GPU threads before the execution of the next instruction continues. Programmers are required to explicitly use these instructions if the application logic relies on a specific ordering of memory reads/writes.

By executing an *atomic instruction* a thread can read, modify, and write back a location in GPU memory in a single step without allowing other threads to either read or modify the specified location.

To do (a), a call to $\mathrm{flush}()$ is inserted before each memory fence instruction. As described before, $\mathrm{flush}()$ function flushes the contents of the modified cache lines to memory and invalidates cache lines by setting their associated mapped addresses to zero. To do (b), a call to an overloaded version of $\mathrm{flush}()$, that also takes an $\mathrm{address}$ and a data structure id as arguments[3], is inserted before each atomic operation with the address of the memory location accessed by the atomic operation and its corresponding data structure identifier as the input. This version of $\mathrm{flush}()$ only flushes the cache line that contains the data pointed to by $\mathrm{address}$. This function does not perform anything if the corresponding data structure is not cached or if the corresponding cache line does not contain the data pointed to by the $\mathrm{address}$[1].

References

[1] 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*, pages 15–24. ACM, 2012.

[3]i.e. $\mathrm{flush(cacheConfig\_t{*}\ cacheConfig,\ void{*}\ address,\ int\ id)}$