Exploiting SPMD Horizontal Locality

Chunyang Gou (c.gou@tudelft.nl) and Georgi N. Gaydadjiev (g.n.gaydadjiev@tudelft.nl)
Delft University of Technology

Abstract—In this paper, we analyze a particular spatial locality case (called *horizontal locality*) inherent to manycore accelerator architectures employing barrel execution of SPMD kernels, such as GPUs. We then propose an adaptive memory access granularity framework to exploit and enforce the horizontal locality in order to reduce the interferences among accelerator cores memory accesses and hence improve DRAM efficiency. With the proposed technique, DRAM efficiency grows by 1.42X on average, resulting in 12.3% overall performance gain, for a set of representative memory intensive GPGPU applications.

1 Introduction

The bulk synchronous programming model[14] has been widely adopted in programming languages targeting manycore accelerator architectures, e.g., CUDA[6] and OpenCL[10]. In such languages, the parallelism of the application's compute intensive kernels is explicitly expressed in a *single program multiple data* (SPMD) manner. *Explicitly-parallel, bulk-synchronous* SPMD program execution on manycores, such as GPUs, often employs *barrel processing*[13] due to its low pipeline implementation overhead.

Off-chip memory bandwidth is becoming a precious resource in current and future manycore processors due to chip pin count limitations. Particularly, the bandwidth can be a severe bottleneck for the manycore accelerator architectures for a growing number of data/memory intensive applications. In addition, DRAM access streams from different cores can easily incur destructive interferences among them, in the following forms: **Hot DRAM Channels:** when multiple cores access single or few DRAM channels, leaving the others idle and basically wasting their bandwidth. **DRAM bank conflicts:** when memory accesses from multiple cores compete reading or writing to different rows of the same DRAM bank causing frequent opening and closing row operations, known for their high penalty. **Bus read/write transition cost:** Shifting between read and write in the same DRAM channel causes latency and bandwidth losses due to the data bus turn-around time which is necessary for the shared input/output data bus design adopted in most contemporary DRAM chips. The above penalties are often noneligible for data intensive applications.

Manycore accelerators using SPMD barrel execution have specific characteristics. In order to better address the off-chip memory bandwidth inefficiencies, the unique characteristics of both the programming and execution models of these manycores should be exploited. In this paper, we leverage a spatial locality typical for SPMD barrel processing (*horizontal locality*) to improve external memory access efficiency. We propose a holistic DRAM bandwidth optimization framework for manycore accelerators with combined compile-time, run-time, and architectural efforts. Our technique utilizes statically detected memory instruction access pattern information, and runtime access granularity scheduling based on the available horizontal locality determined by the access pattern. With the support of co-designed hardware, adaptive memory access granularity is achieved and DRAM efficiency and the overall performance are improved.

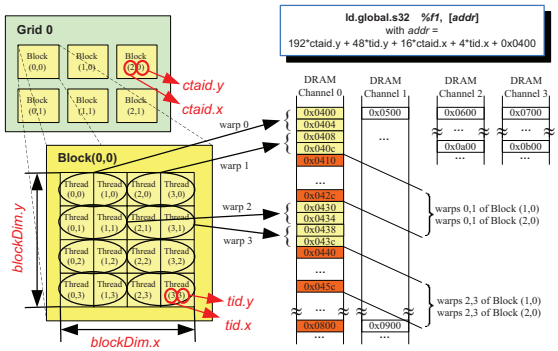

Fig. 1. Worker threads hierarchy and memory accesses

In the remaining sections, we analyze the horizontal locality (Sec. 2), exploit it using our adaptive memory access granularity scheme (Sec. 3), and evaluate the memory efficiency improvement using cycle level full system simulation (Sec. 4). Experiments show that for a set of memory intensive GPGPU applications our technique improves DRAM efficiency by 1.42X and overall performance by 12.3%.

2 SPMD Barrel Execution Analysis

Programming Model Properties: In *explicitly-parallel, bulk-synchronous* SPMD programming models, the programmer extracts the application data-parallel section, identifies the basic working unit (e.g., element in the problem domain), and explicitly expresses the same sequence of operations on each working unit in a *kernel*. Multiple kernel instances (CUDA *threads*) run independently on the accelerator cores.

In CUDA, parallel threads are organized in 2-level hierarchy, in which a kernel (called *grid*) consists of parallel CTAs (*Cooperating Thread Array*, or *block*), with CTAs composed by parallel threads, as shown in Fig. 1[1]. Explicit, localized synchronization and on-chip data sharing mechanisms (e.g., CUDA shared memory) are supported inside each CTA.

Baseline Manycore Barrel Processing Architecture: Fig. 2 shows our baseline architecture. On the right the high-level system organization is shown. The system consists of an accelerator node with K cores and a memory subsystem with

1. CUDA, with warp size reduced from 32 to 2 for simplification

Manuscript submitted: 28-Feb-2011. Manuscript accepted: 19-Mar-2011. Final manuscript received: 28-Mar-2011

Posted to the IEEE & CSDL on 4/15/2011
DOI 10.1109/L-CA.2011.5 1556-6056/10/$26.00 © 2011 Published by the IEEE Computer Society

L DRAM channels, connected by the on-chip interconnect[2]. The host processor offloads compute intensive kernels to the accelerator cores. The kernel code and parameters are transferred using the host interface, and the workloads are dispatched at the grain of independent CTAs/blocks.

The left part of Fig. 2 illustrates a single accelerator core. During execution, a batch of threads from the same CTA are grouped into a *warp*, the smallest unit for the pipeline front-end processing. Each core maintains a set of on-chip hardware execution contexts and switches among them at the warp granularity. The context switching, also called *warp scheduling*, is done in an interleaved manner, also known as *barrel processing*[13]. Warps are executed by the core pipelines in a SIMD fashion to improve pipeline front-end efficiency. Warps can access two memory types: on-chip shared and off-chip memory. When there is an off-chip memory access, the execution is taken care of by a miss status holding register (MSHR), shown in Fig. 2. The memory access information is logged by the allocated MSHR entry, and warp execution is put into inactive status.

Horizontal Locality: Within barrel execution of SPMD kernels, memory access behavior of manycore accelerators is determined not only by a *single* thread/warp, but also by concurrent warps execution. Fig. 1 shows a typical CUDA kernel 2D address pattern. Please note that, the access pattern not only guides each worker thread to its working field (data memory address), but also binds the relationship among threads memory accesses. For example, memory addresses of warps 0 and 1 are always contiguous, for the given access pattern. Therefore, spatial locality can be exploited. Different from the spatial locality in general purpose processors, this locality type has two distinct characteristics: 1) it is *inter-thread/warp locality* among multiple independent warps; 2) it can only benefit the neighbor threads/warps but not the memory access initiator. We call such inter-thread/warp spatial locality *horizontal locality*.

Above we have assumed that all warps are executing the same memory instruction using the same access pattern. In fact, this originates from the combination of the SPMD programming model and the barrel execution model used by contemporary manycore accelerators, e.g., GPGPUs. Within *strict barrel execution*, an instruction from each warp context is launched at each clock cycle in an *interleaved* manner. Moreover, all warps are executing the same SPMD kernel code. In this way, the execution of concurrent warps in a core is *highly correlated*, and thus the in-flight instructions are similar. Even with *relaxed barrel execution*, where consecutive instructions from the same warp are allowed to issue into the SIMD pipeline, since the average size of *independent instruction blocks*[3] is small (1∼3 instructions in our benchmarks), the horizontal locality can still be captured in time[4] before the requested data arrives.

As stated in Sec. 1, concurrent memory accesses of different cores can interfere in manycores. For example, even if accesses from warps 0 and 1 of block (0,0) are issued back-to-back to memory, they can be *separated* on the way to DRAM, e.g., by on-chip interconnect or memory controllers. As a result, their horizontal locality is broken, leading to extra

2. Accelerator nodes may be separate chips (e.g., GPUs), or together with the host CPU(s) be on the same die (e.g., [1])

3. A sub-basic block with mutually independent instructions

4. 32 warps will issue a 3 instructions block in 384 cycles, shorter than average main memory access delay (> 500 cycles in our case)

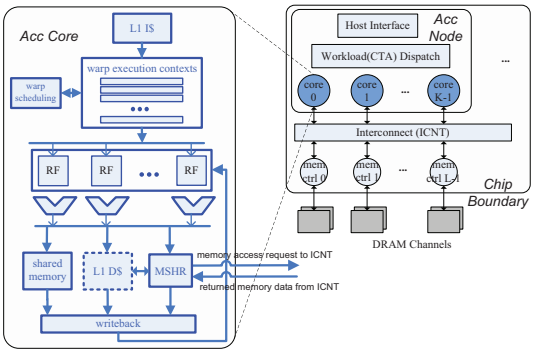

Fig. 2. Baseline barrel processing accelerator architecture

DRAM bank conflicts and memory bus transition penalties. In the following, we propose a novel way to *exploit* and *enforce* the horizontal locality in manycore accelerators and improve external memory *bandwidth efficiency*.

3 Improving Memory Efficiency

Compiler and Runtime Access Pattern Analyzer: In an explicitly-parallel, bulk-synchronous SPMD program, in order for the worker thread to identify its working set, a mapping between the thread id and its working set is designated in the code: $addr = \Phi(tid.z, ctaid.y, tid.y, ctaid.x, tid.x)$, see Fig. 1. Ideally Φ can have arbitrary form, however, the number of patterns used in programmers practice is rather small, and address generation complexity is often limited. Leveraging this observation, we have prototyped a framework able to detect and exploit the most common memory access patterns for CUDA kernels. Our framework employs static control- and dataflow analysis at compilation to detect the access *skeleton* type, and build the corresponding parameters expressions. A skeleton is defined as a *parameterized* address mapping function, which is able to generate *a class of* memory access patterns. Our runtime library will evaluate the parameters expressions provided by compiler analysis, based on the CUDA kernel dimensions and input parameters available at launch time. We implemented our prototype as additional NVCC compiler pass and library extensions to CUDA runtime environment. Currently, our framework supports 2 most common *skeletons – 1D/2D contiguous/strided/block-strided access:*

$$\eta(a,b,c,d,e,f)=(a,b,c,d,e,f)\cdot(tid.z,ctaid.y,tid.y,ctaid.x,tid.x,1)$$
$$=a\cdot tid.z+b\cdot ctadid.y+c\cdot tid.y+d\cdot ctaid.x+e\cdot tid.x+f$$

with parameters $a,b,c,d,e,f\in\mathbb{N}$ (used in regular memory access patterns); and *skewed access:*

$$y=\eta(a,b,c,d,e,f)$$
$$\varphi(h1,l1,s1,h0,l0,s0,\alpha,\beta,y)=(y[h1{:}l1]\ll s1 \mid y[h0{:}l0]\ll s0)\cdot\alpha+\beta$$

where, $h1,l1,s1,h0,l0,s0,\alpha,\beta\in\mathbb{N}$ (used in irregular applications). Skewed access skeleton is constructed using 1D/2D access (y), and has 2 bits sections: higher, taken from $h1$ to $l1$ of y shifted left by $s1$ bits; and lower section generated similarly.

The parameters of the above two equations are internally represented as trees. For now we support only 2 skeletons with rather complete representations to investigate our approach's true benefits under a constrained case scenario.

Adaptive Memory Access Granularity Scheduling: To reduce bank conflicts and bus turn-around times, *adaptive memory access granularity* (Alg. 1) is created to determine the DRAM access granularity, based on access pattern information generated by the memory pattern analyzer.

Alg. 1 Adaptive memory access granularity scheduling

1: $max_gran \leftarrow$DRAM channel interleaving granularity
2: $min_gran \leftarrow$DRAM minimal access granularity
3: $data_size \leftarrow$data size of memory access instruction
4: $blockDim.x \leftarrow$block size in dimension x
5: $d, e, is_skewed_access, l1, s1, h0 \leftarrow$access pattern parameters
6: **if** $e! = data\_size$ or $blockDim.x \times e! = d$ or (is_skewed_access and ($l1 \leq h0$ or $s1 \leq h0$ or $2^{h0} \leq max\_gran$)) **then**
7: **return** min_gran
8: **else if** $blockDim.x \times e > max\_gran$ **then**
9: **return** max_gran
10: **else if** $blockDim.x \times e < min\_gran$ **then**
11: **return** min_gran
12: **else**
13: **return** $blockDim.x \times e$
14: **end if**

The essential idea of the adaptive granularity scheduling is as follows. Since the address pattern determines the locality among neighbor warps' memory accesses, we can choose the maximal access granularity *allowed* by the horizontal locality, for a given memory instruction. When a warp memory instruction execution misses, it initiates the external memory access with the *large granularity*. Due to the horizontal locality, the extra requested data will be used by neighbor warps in most cases, hence, improving memory efficiency, *without the penalty of wasted memory bandwidth*.

The key in the adaptive memory access granularity is to correctly identify the available horizontal locality. Line 6 of Alg. 1 examines if a memory instruction has the appropriate horizontal locality. Currently, only instructions accessing contiguous memory addresses among neighbor threads in a block x-axis (Fig. 1) are considered. Lines 8 to 13 determine the proper access granularity under three constraints: 1) min access granularity, set by the DRAM interface[5]; 2) max granularity, set by DRAM channel interleaving granularity; and 3) the *block boundary* constraint ($e \times blockDim.x$), because a warp should not fetch data outside block boundary.

Example: For the global memory load instruction in Fig. 1, our compiler pass identified *2D access* skeleton, with parameters $(a,b,c,d,e,f)=(0,192,48,16,4,0)$ calculated by the runtime by evaluating their tree representations. At kernel launch time, first the warp memory access contiguity is determined, by checking if neighbor warps accesses fail to cover the contiguous address space ($e \neq data\_size$ or $blockDim.x \times e \neq d$, false in this case). At this point, the access pattern is assured to have *strong* horizontal locality where contiguous $blockDim.x \times e$ (16 in this case) bytes data will be *fully* used by the load instruction warps execution in the block. Therefore, an MSHR entry is allocated, and 16B external memory request is fired into the interconnect when warp 0 is run. In case of no HW data cache, warp 0 has also to book MSHR space for the fired load request, *only for its own requested data* (8B data segment X). Later, when warp 1 is invoked for the load execution, it checks the MSHR tag array to find an ongoing load which *covers* its request (8B data segment Y). At this point, it allocates an MSHR entry and reserves the data space for Y. Afterwards, it becomes inactive, awaiting data Y to return from DRAM. The HW mechanism supporting this process is called *elastic MSHR with deferred reservation*.

5. 32B for GDDR3 in our study

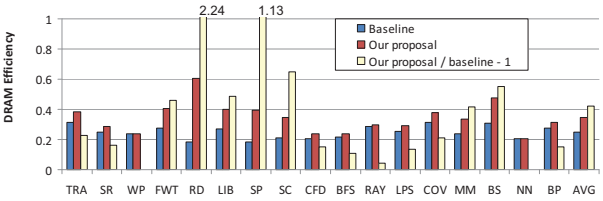

Fig. 3. DRAM efficiency improvement

4 Experimental Evaluation

Experimental Setup: We use a modified version of GPGPU-Sim[2], a cycle level full system simulator implementing PTX ISA[7]. The detailed configuration of the modeled accelerator is shown in Tbl. 1. The prototyped access pattern analyzer and the adaptive access granularity scheduling run on the host CPU. When a kernel is to be launched, the scheduled optimal access granularity values for memory instructions are transferred to the accelerator. At the accelerator node, the baseline and our proposal execution differ mainly in memory access granularity: 32B[6] *vs* value adapted to the memory instruction. The adaptivity, exploiting both intra- and inter-warp locality, is *determined* by the scheduler at launch time, and *realized* by the elastic MSHR (Sec. 3) implemented in our architecture. In our experiments only adaptive memory **load** granularity is evaluated. We used 17 memory intensive benchmarks from CUDA SDK[8], Rodinia[3], and [2], common in GPU architecture research.

DRAM Access Granularity Distribution: In our experiments, no particular pattern in the granularity type distribution is observed across all benchmarks – their optimal granularity spread among all four valid categories of 32/64/128/256B in our configuration. Moreover, 10 out of the 17 benchmarks require at least two access granularity types. This suggests that in general a *single* optimal access granularity for all memory accesses is not feasible, and, confirms the need for scheduling memory instructions separately, according to their access patterns.

Improved DRAM Efficiency: DRAM efficiency is defined as the ratio between DRAM data bus actual transfer cycles and the number of cycles with pending DRAM access: $E_{tot}=\frac{\sum_{k=0}^{\#kernels-1}\sum_{i=0}^{\#channels-1}\#bus\_transactions_{k,i}}{2\cdot\sum_{k=0}^{\#kernels-1}\sum_{i=0}^{\#channels-1}\#active\_cycles_{k,i}}$, where $\#bus\_transactions_{k,i}$ is the number of accomplished DRAM bus transactions in channel *i* during kernel *k* execution, and $\#active\_cycles_{k,i}$ is the number of DRAM bus cycles during which channel i is not *completely idle*[7]. The factor *2* in the equation is due to two bus transactions per cycle for *dual-data-rate* memories.

Fig. 3 shows the overall DRAM efficiency of our adaptive access granularity scheme for all benchmarks. Each group has three bars, left and middle for the DRAM efficiency with the baseline and our proposal and the right bar for the net gain. The RD and SP see the largest DRAM efficiency improvement due to very few memory access instructions with simple access patterns (high horizontal locality). Meanwhile, we have observed that most fetched data are efficiently utilized by neighbor warps in such kernels, even at the largest access granularity of 256B. Other

6. Appropriate for the simulated 8-way SIMD pipeline

7. The DRAM channel status with no pending access in its request queue and any ongoing memory access

TABLE 1
Baseline Accelerator Configuration†

Number of cores	16 @1296.0 Mhz
Core Configuration	8-wide SIMD execution pipeline, 24 pipeline stages 32 threads/warp, 1024 threads/core, 8 CTAs/core, 16384 registers/core warp scheduling policy: Round-robin, execution model: strict barrel processing (Sec. 2)
On-chip Memories	16KB software managed cache (i.e., shared memory)/core, 8 banks, 1 access per core cycle per bank 64 MSHRs/core, with 32B data field per MSHR (no hardware cache)
DRAM	4 GDDR3 memory channels, 2 DRAM chips per channel, 2KB page per DRAM chip, 8 banks per DRAM chip 8 Bytes/channel/transmission, 68.2 GB/s aggregate bandwidth @1066 Mhz bus freq GDDR3 memory timing: t_{CL}=12, t_{RP}=12, t_{RC}=41, t_{RAS}=29, t_{RCD}=14, t_{RRD}=10 memory controller policy: out-of-order (FR-FCFS)[11], 32 DRAM request buffer entries
Interconnect Network	crossbar (2-ary 5-fly butterfly[4]) @602.0 Mhz, 16-Byte flit size, 2 Virtual Channels, 8 buffers per Virtual Channel

† a *downsized* NVIDIA GeForce GTX 280 version, with 16 vs 30 processor cores and half of its aggregated DRAM bandwidth (68.2 instead of 141.7 GB/s)

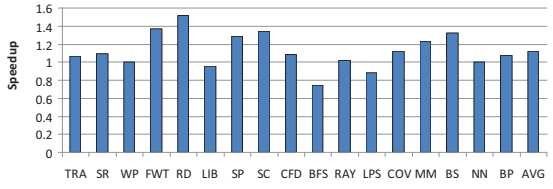

Fig. 4. Speedup over the baseline

benchmarks also show varied efficiency increment, with the only exceptions being WP and NN. The reason is that the *identified* access patterns do not have exploitable horizontal locality, while for the rest instructions the analyzer failed to extract the precise access patterns as they are *control flow dependent*. As a result WP and NN are executed with the same access granularity as the baseline. Nonetheless, our scheme improves the average DRAM efficiency by 1.42X.

Overall Performance Improvement: Fig. 4 shows the overall performance improvement with our adaptive memory granularity scheme. Only kernel execution times on the accelerator are counted in calculating the speedup (our lightweight runtime library (Sec. 3) has little CPU overhead). On average, performance is improved by 12.3%. Some benchmarks, e.g., BFS and LPS, show performance degradation. Detailed analysis reveals that severe *inter-warp control flow divergence* occurs during kernel execution. In this case, neighbor warps execute along different control flow paths, rendering the extra data fetched at large granularity being wasted. As a result, memory bandwidth is wasted and system performance is degraded, especially for BFS (heavily memory bound). This requires further investigation.

5 Related Work

GPGPU memory performance optimizations have been addressed at different architectural levels. At the **processor core** level, there have been studies in applying *prefetching* techniques for GPGPUs[12], however only for data *inside* one thread. A recent GPGPU prefetching proposal *Inter-Thread Prefetching (IP)*[5], is quite similar to our work spirit, since recognizes the GPU specific locality among parallel threads. *IP* focused on *latency reduction* using speculation, while our work emphasize on *external memory access efficiency*, using *accurate access pattern* information. Moreover, our work is novel in analyzing the horizontal locality in SPMD barrel execution embodied by GPGPUs. *Memory coalescing*[9] is a HW mechanism in NVIDIA GPUs to buffer and merge *intra-warp* memory accesses. It assumes half warp or single warp scope, due to lacking future warp access information. In contrast, our proposal takes advantage of the high level access pattern information, and captures horizontal locality both inside and among warps even when **not** issued back-to-back (Sec. 2). We observed 28.4% performance gain for RD and 10+% for FWT and BS, while comparing our proposal with half-warp coalescing. On the other hand, coalescing tracks dynamic warp memory access behavior, being *complementary* to our proposal in capturing the control flow dependent horizontal locality (not captured by static analysis, such as BFS and LPS).

At the **interconnect** level, work in [15] addresses *memory access streams interleaving* problems in GPU interconnect, using a customized control flow policy. At the **memory controller** side, sophisticated out-of-order DRAM scheduling schemes, e.g., FR-FCFS[11], use queues to reorder and optimize DRAM accesses, albeit limited by the queue size. In contrast, we utilize high-level access pattern information, and adaptively adjust memory access granularity to prevent locality from being broken.

6 Conclusion

In this paper, we analyzed horizontal locality inherent in manycore accelerators with SPMD barrel execution. We proposed an adaptive memory access granularity scheme to exploit the horizontal locality and reduce memory access interferences among cores to improve DRAM efficiency. Our proposal improves DRAM efficiency by 1.42X on average, and the overall performance by 12.3%, for a set of representative memory intensive GPGPU applications.

References

[1] AMD, http://sites.amd.com/us/fusion/apu/pages/fusion.aspx
[2] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS 2009*.
[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC 2009*.
[4] W. J. Dally, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.
[5] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for gpgpu applications," in *MICRO 2010*.
[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, March 2008.
[7] NVIDIA, "The CUDA compiler driver NVCC," 2009.
[8] NVIDIA, http://developer.nvidia.com/page/tools.html
[9] NVIDIA, "CUDA best practice guide, edition 3.0,"
[10] "OpenCL home page," http://www.khronos.org/opencl/
[11] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory access scheduling," in *ISCA 2000*.
[12] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded GPU," in *CGO 2008*.
[13] M. R. Thistle and B. J. Smith, "A processor architecture for horizon," in *Supercomputing 1988*.
[14] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, August 1990.
[15] G. Yuan, A. Bakhoda, and T. Aamodt, "Complexity effective memory access scheduling for many-core accelerator architectures," in *MICRO 2009*.