Warp-Consolidation: A Novel Execution Model for GPUs

Ang Li
Pacific Northwest National Lab, USA
Ang.Li@pnnl.gov

Weifeng Liu
Norwegian University of Science and Technology, Norway
weifeng.liu@ntnu.no

Linnan Wang
Brown University, USA
linnan_wang@brown.edu

Kevin Barker
Pacific Northwest National Lab, USA
kevin.barker@pnnl.gov

Shuaiwen Leon Song
Pacific Northwest National Lab, USA
College of William and Mary, USA
Shuaiwen.Song@pnnl.gov

ABSTRACT

With the unprecedented development of compute capability and extension of memory bandwidth on modern GPUs, parallel communication and synchronization soon becomes a major concern for continuous performance scaling. This is especially the case for emerging big-data applications. Instead of relying on a few heavily-loaded CTAs that may expose opportunities for intra-CTA data reuse, current technology and design trends suggest the performance potential of allocating more lightweighted CTAs for processing individual tasks more independently, as the overheads from synchronization, communication and cooperation may greatly outweigh the benefits from exploiting limited data reuse in heavily-loaded CTAs. This paper proceeds this trend and proposes a novel execution model for modern GPUs that hides the CTA execution hierarchy from the classic GPU execution model; meanwhile exposes the originally hidden warp-level execution. Specifically, it relies on individual warps to undertake the original CTAs' tasks. The major observation is that by replacing traditional inter-warp communication (e.g., via shared memory), cooperation (e.g., via bar primitives) and synchronizations (e.g., via CTA barriers), with more efficient intra-warp communication (e.g., via register shuffling), cooperation (e.g., via warp voting) and synchronizations (naturally lockstep execution) across the SIMD-lanes within a warp, significant performance gain can be achieved. We analyze the pros and cons for this design and propose corresponding solutions to counter potential negative effects. Experimental results on a diverse group of thirty-two representative applications show that our proposed Warp-Consolidation execution model can achieve an average speedup of *1.7x, 2.3x, 1.5x and 1.2x* (up to *6.3x, 31x, 6.4x and 3.8x*) on NVIDIA Kepler (Tesla-K80), Maxwell (Tesla-M40), Pascal (Tesla-P100) and Volta (Tesla-V100) GPUs, respectively, demonstrating its applicability and portability. Our approach can be directly employed to either transform legacy codes or write new algorithms on modern commodity GPUs.

CCS CONCEPTS

•Computer systems organization → Single instruction, multiple data; •Software and its engineering → Synchronization; •Theory of computation → *Parallel computing models;*

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

ICS '18, Beijing, China

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

978-1-4503-5783-8/18/06…$15.00
DOI: 10.1145/3205289.3205294

1 INTRODUCTION

Classic GPU programming models such as CUDA organize thread-level-parallelism in two levels: *thread blocks* and *threads* [1]. However, from hardware perspective, there are three execution hierarchies: *Cooperative-Thread-Array (CTA)*, *warp* and *SIMD-lane*[1]. When a kernel is launched, a thread block is mapped to a CTA while a thread is mapped to a SIMD-lane. Consequently, the warp-level execution is transparent to programmers unless warp primitives such as voting are specifically invoked. For example, programmers often have to manually extract *warp-id* from a kernel at runtime.

Moreover, it is common in parallel execution that workload is unbalanced due to the disparity arising from various aspects, e.g., application code, input data, shared resource contention, runtime scheduling, etc. [2]. Delay from such unbalanced workload can be largely mitigated or even eliminated when multiple tasks are allocated, executed and released freely and independently. However, when data-sharing and/or control-sharing occurs, including *S*ynchronization, *C*ommunication and *C*ooperation (*SCC* for short), such delay can be amplified, resulting in system resource under-utilization and performance degradation. This is especially the case for GPU, as its fundamental design principle is to rely on massive parallelism to hide the long latency from expensive off-chip memory access. GPU workload imbalance can occur among both threads and thread blocks. At thread-block level (*inter-CTA*), this is nonissue since CUDA and OpenCL enforce completely independent thread blocks. This ensures a thread block can be fetched, dispatched and retired on an arbitrary available SM without interfering with others. At thread level (*intra-CTA*), however, sharing may incur significant overhead, as threads in a thread block can provoke *control-sharing* and *data-sharing*.

Intra-CTA thread sharing can be further categorized into two types: sharing among threads within the same warp and sharing among threads from different warps. The former, also known as *warp-divergence*, is extensively discussed [3–8]. The latter, however, has been largely ignored since conventional wisdom assumes that warps executed in parallel can effectively hide the latency from data and control sharing.

In this paper, we demonstrate that such overhead from inter-warp sharing within the scope of a CTA can severely limit application performance that may be otherwise deliverable from GPU's large-scale SIMT. For example, Figure 1 profiles the percentage of the stalls that are caused by warp synchronization (one type of inter-warp SCC sharing) across 80 common applications on a NVIDIA

[1]In this paper, we use NVIDIA terminology, as it matches the evaluation platforms. Meanwhile, we use thread and lane, CTA and thread block interchangeably.

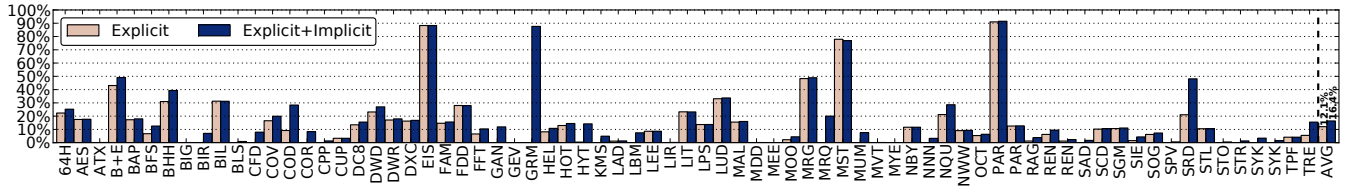

Figure 1: Percentage of stalls due to CTA synchronization. "Explicit" implies user-defined "`__syncthreads()`" barrier. "Implicit" describes the implicit internal barrier upon kernel termination.

Tesla-P100 GPU. As demonstrated by the pale pink bars, stalls from inter-warp synchronization that explicitly uses `__syncthreads()` accounts for an average of *12%* (up to *90%*) of the total execution stalls on GPU SMs. These *explicit* synchronizations can significantly impact performance since they prevent GPU warps from overlapping execution and hiding latency. Additionally, extra delay may also incur at kernel termination as warps cannot be allocated and released individually (i.e., allocate and release occur at thread block level). To assess such *implicit* synchronization overhead, we manually insert a CTA barrier at the end of each kernel in our profiling. Illustrated by the dark blue bars in Figure 1, the percentage of the stalls due to explicit+implicit synchronization grows up to 16.4% on average; for certain applications such as `GRM` and `SRD`, such implicit stalls appear to be quite significant.

This work focuses on optimizing applications with strong inter-warp SCC sharing, e.g., applications composed of repeated warp synchronization calls, demonstrating great warp-level workload imbalance, and/or performing intensive inter-warp communication via shared memory. We propose a novel warp-based execution model called **Warp-Consolidation** for boosting the overall kernel performance by employing one warp to complete the entire workload of an original CTA. This essentially transforms warp-level SCC to lane-level SCC, leading to significant SCC overhead reduction. Meanwhile, on-chip resource allocation and release can now be subject to warps instead of CTAs, mitigating the resource fragmentation issue [9, 6].

Our Warp-Consolidation execution model boosts performance by significantly reducing the overhead from (i) inter-warp synchronization (i.e., eliminated), (ii) inter-warp communication and (iii) false waiting due to workload imbalance. For (i), we convert explicit synchronization across warps to synchronization within a warp, which is naturally established by SIMT lockstep execution. For (ii), we replace the inter-warp communication via shared memory by highly-efficient register shuffling at SIMD-lane level. For (iii), we propose warp delegation to allocate and release CTA jobs independently and instantaneously, while maintaining high occupancy. Evaluation on a wide spectrum of representative applications demonstrates that our proposed method can achieve an average speedup of *1.7x, 2.3x, 1.5x and 1.2x* (up to *6.3x, 31x, 6.4x and 3.8x*) on NVIDIA Kepler, Maxwell, Pascal and Volta GPUs, respectively. We also discuss the compatibility of our approach to the current technology trend and suggestions on how to write new algorithms under our execution model. In summary, this paper makes the following contributions:

- We propose the concept, methodology and design of Warp-Consolidation execution model (Section 3).
- For explicit synchronization among warps, we propose *warp aggregation* to convert them into synchronization across SIMD-lanes, which is naturally guaranteed by GPU's SIMT lockstep execution (Section 3.1).
- For the communication among warps via shared memory, we propose *register remapping* to convert it into highly-efficient inter-lane shuffling, which is achieved via native register operations (Section 3.2).
- For the implicit synchronization upon the termination of a GPU kernel, resource fragmentation and potential occupancy degradation issues, we propose *warp delegation* to eliminate unnecessary waiting and enhance occupancy (Section 3.3).
- We validate our Warp-Consolidation model on all the recent generations of NVIDIA GPU architectures and results demonstrate that it can substantially improve overall performance by drastically reducing SCC overheads and resource fragmentation (Section 4).

2 GPU THREE LEVEL PARALLELISM

Shown in Table 1, we summarize the pros and cons of synchronization, cooperation and communication (SCC) at three GPU execution levels: *CTA*, *warp*, and *SIMD-lane*.

2.1 CTA-Level SCC

CTAs are fully independent. From the software perspective, GPU programming models such as CUDA ensure that thread blocks (or CTAs) are independent [1]. From hardware perspective, CTAs can be issued, dispatched, executed and released freely without interfering with each other, regardless of whether they are on the same SM or not. In other words, *there is no control-sharing among CTAs*. Additionally, SM resources such as warp-slots, registers, and shared memory, are allocated and released according to CTAs. Traditionally, there is no synchronization or cooperation mechanism among CTAs natively enabled on GPUs. Although smart circumventing approaches have been proposed [11, 12], they essentially convert CTA synchronization to warp or thread synchronization in global memory (e.g., global memory atomic); and deadlocks have to be carefully avoided [13]. However, this situation is recently changed due to the introduction of *Cooperative Groups* [14] under the latest CUDA runtime together with Pascal or Volta architectures, where the whole kernel grid can synchronize with a barrier. Finally, for communication among CTAs, if by any means Read-After-Write (RAW) race condition is resolved, it is through global memory, which can take hundreds of cycles (see Table 1).

2.2 Warp-Level SCC

Warps in a CTA are semi-independent. They can be fetched and executed freely. This is why branching across warps, if not producing workload imbalance, is not harmful to performance [15]. However, a warp cannot be allocated and released individually — all warps in a CTA must be terminated at the same time. In other words, *warps in a CTA have control-sharing*. Moreover, an implicit barrier is always enabled at kernel termination to ensure the arrival of all warps before the CTA can exit as a whole. Furthermore, the SM resources are

Table 1: GPU three execution hierarchies and their SCC features.

Hierarchy	Independence	Allocate/Destroy	Independent Execution	Synchronization	Cooperation	Communication	Latency(cycle)	Bandwidth
CTA	Complete	Yes	Yes	No	No	global memory	$O(100)$	$\sim$ 1x[10]
Warp	Semi	No (resp. CTA)	Yes	__syncthreads	bar.	shared memory	$O(10)$	$\sim$ 7.6x[10]
Lane	No (except Volta)	No (resp. CTA)	No (except Volta)	lockstep	warp-vote	register shuffling	$O(1)$	$\sim$ 47.3x[10]

allocated and retreated at the granularity of a CTA instead of a warp. Thus, when warp workload imbalance occurs, some executing resources such as registers are wasted. Several hardware-based approaches have been proposed to allocate resources at warp level [9, 6, 16].

Warp-level **synchronization** is achieved through the native primitive "`__syncthreads()`". It operates as a barrier for warps rather than threads [17]. Warp barrier is used to coordinate warps and ensure shared and global memory consistency. The overhead of barrier comes from three sources: (I) raw execution overhead (tens of cycles [17]), (II) memory fence (i.e., `__threadfence_block()`) to ensure memory access consistency, and (III) warp workload imbalance. The imbalance induced overhead can be extremely significant when it is used to synchronize short segments of code that only contain a few memory access inside loops. Warp **cooperation**, proposed in [18] and [15], rely on PTX instruction "`bar.arrive`" and "`bar.sync`" [19] to perform producer-consumer patterns. Although divergence among warps may not necessarily lead to workload imbalance, warps following entirely distinct execution path (e.g., in warp cooperation) may introduce large amount of instruction cache misses which may severely hurt performance [15]. Inter-warp **communication** or data sharing, is often through shared memory. However, to avoid RAW hazard and maintain memory access consistency (note warps are independent for execution), synchronization or atomic access is often required [1]. This adds extra delay for communication. For example, shared memory communication in GPU typically takes tens of cycles (Table 1).

2.3 Thread-Level SCC

Threads in a warp are traditionally not independent. For architectures before Volta, warp is the basic unit for instruction fetching and execution; a thread, if executed separately, is within a *warp divergence* region. For warp divergence, a warp will traverse the divergent path of each branch sequentially, incurring significant delay [20]. The latest Volta architecture changes the picture by maintaining execution state for each thread rather than shared by the entire warp, thus allowing independent thread scheduling. Nevertheless, the execution model is still SIMT [14], although the divergence and reconvergence of threads in a warp are more flexible and efficient.

Since threads in a warp are executed in a lockstep manner, thread **synchronization** is implicitly integrated in the instruction. Thread **cooperation** can be achieved via warp voting instructions, which allow lanes in a warp to evaluate a predicate register and broadcast the result to all the lanes. There are three voting instruction variants: "`__all()`" and "`__any()`" merge the results of the 32 lanes into a single *bool* and then broadcast it. "`__ballot()`" conjuncts a 32-bit mask, with each bit representing the predicate of each respective lane [1]. For older GPU architectures such as Fermi and Kepler, it is also possible to cooperate threads via the shared memory lock-bits [13]. Furthermore, thread **communication** can be accomplished through inter-lane register shuffle. Shuffle is the instruction to exchange data among lanes in a warp, i.e., a lane can obtain the register value of another lane. There are four shuffle instruction variants: "`__shfl()`" is indexed as any to any transfer; "`__shfl_up()`" and "`__shfl_down()`" are to shift left and right to the *n*th neighbor; and "`__shfl_xor()`" represents butterfly (*XOR*) exchange [1]. All these primitives are implemented by native hardware operations without shared memory involved. Note that each shuffle transaction can only exchange 4 bytes of data. For data types other than 4 bytes, one can refer to the SHFL library [21]. Shuffle instructions are widely supported since the Kepler architecture. Additionally, inter-lane communication is also feasible through shared memory. However, the shared memory partition then must be declared as *volatile* to ensure memory consistency.

2.4 Summary

The three GPU execution hierarchies and their SCC features are summarized in Table 1. With finer granularity, the degree of independence declines but the efficiency of SCC sharing enhances. In conclusion, a coarser execution granularity seems to be better for hiding latency and amortizing control overhead within a CTA, but suffers from less efficient control sharing and data sharing (synchronization, cooperation and communication).

3 WARP-CONSOLIDATION EXECUTION MODEL

Since managing SCC among fine-grained SIMD-lanes are much more efficient than across warps, we are considering the possibility of transforming warp-level SCC to lane-level SCC, which particularly benefits applications that are SCC-heavy (Figure 1). With a finer granularity, it may also benefit applications having resource fragmentation issues. To reach this goal, we propose a new GPU execution model named **Warp-Consolidation model** which restricts a CTA to have only one warp inside. In other words, it applies one warp to complete the entire workload of a CTA. Comparing with the conventional thread-block/thread two-level execution model from CUDA, Warp-Consolidation model exhibits the following advantages:

- **Synchronization:** No synchronization (i.e., `__syncthre-ads()`) is required at all as lanes in a warp are always executed in lockstep. Under this model, there is no divergence across warps, thus no stalls come from warp workload imbalance.
- **Communication:** Data communication is now conducted via register shuffling. This is much more efficient than through shared memory which demands an extra register read & write, an extra shared memory read & write, and a warp-level synchronization.
- **Cooperation:** Cooperation is implemented through the warp voting instructions, which is much more efficient than using warp-level barrier-based primitives (e.g., `bar.sync`–`bar.arrive` pair chain [15, 19]).
- **Resource Allocation:** Under this model, all the on-chip resources are allocated subject to warp granularity. In this way, it is feasible to precisely control the number of warps dispatched to an SM so that when register and shared memory are not constraints, maximum occupancy could be achieved.
- **Resource Release:** A warp now can retire and release all its resources (registers, shared memory, warp slots, etc) immediately

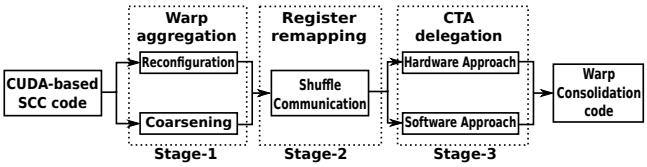

Figure 2: Warp-Consolidation model guided optimization workflow.

upon job completion without waiting for other inflight warps to exit together [2].

Additionally, from our previous work [22, 23], we found that that the "`clock()`" instruction can be leveraged to signal warp context switch: once we add the clock instruction into an empty waiting loop, a spinning-warp is able to convey the execution to another warp which it is busy-waiting for; otherwise an infinite waiting will occur. Such a side-effect is probably because the `clock` instruction is often used for counting ticks in the waiting loop. Latest practice also suggests the benefit of using efficient atomic operations and reduction at warp level [24]. Throughout the paper we will demonstrate that using our model to convert SCC type of GPU programs can significantly improve their overall performance, even under reduced occupancy[2] (Section 4-5) .

In summary, our proposed Warp-Consolidation model describes a two-level execution model: **warps** and **threads**. We view CTA as **transparent**, similar to the hidden role of warps in the CUDA model. A warp is granted the ability for independently fetching, dispatching, resource allocation, scheduling, executing and releasing. Synchronization is naturally ensured. Cooperation is done through warp voting while Communication is accomplished via register shuffling. Shared memory is viewed as thread-private space for relieving register pressure rather than a shared space across warps, since data sharing is already handled by the more efficient register shuffling. The optimization workflow for transforming programs that are written via traditional CUDA execution model to ours is illustrated in Figure 2. It comprises three main stages: (a) *warp aggregation*, (b) *register remapping* and (c) *warp delegation*.

3.1 Warp Aggregation: Eliminating Synchronization

We propose a two-step approach to aggregate the number of warps in a CTA into one.

Step 1: Adjusting CTA size. If the CTA size (e.g., number of threads contained) in a kernel configuration can be simply adjusted without affecting application correctness, it is ideal to change it to a single warp size, which is 32 threads for all existing NVIDIA GPUs. Common CUDA programming guide suggests the CTA or thread block size should be a minimum of 64 threads (i.e., 2 warps) [25]. However, under our Warp-Consolidation model, we aim to adjust the CTA size to 32 threads (only one warp) which implies an expanded number of CTAs with smaller individual CTA size. This intends to increase the number of CTAs launched on an SM in order to increase CTA-level parallelism to compensate the reduced intra-CTA level parallelism. Listing 1-2 show an example on how to adjust CTA size for `BFS` kernel described in Table 3. The number of warps per CTA is reduced from 16 to 1 (i.e., 512 to 32 threads in Line 8), while `__syncthreads()` and `__threadfence_block()` (Line 4 and 5 in Listing 1) are removed. The reduction of CTA size often comes with

[2]Occupancy or warp-slots utilization is the ratio of the number of active warps per SM to the total number of hardware warp-slots per SM.

```
1  __global__ void Kernel(...){
2    __shared__ sh[N];
3    for(...){
4      __syncthreads();
5      __threadfence_block();
6      ...}
7  }
8  num_thds = 512;//16 warps
9  num_blks = (int)ceil(num_nodes/
10           (double)num_thds;
11 dim3 grid(num_blks,1,1);//grid config
12 dim3 thds(num_thds,1,1);//blk config
13 do{
14   Kernel<<<grid,thds>>>(...)
15 while(stop);
```

Listing 1: `BFS` original kernel

```
1  __global__ void Kernel(...){
2    __shared__ volatile sh[N];//volatile smem
3    for(...){
4      //sync is avoided
5      //blockwise memory fence is avoided
6      ...}
7  }
8  num_thds = 32;//reduce to one warp
9  num_blks = (int)ceil(num_nodes/
10           (double)num_thds;//num of blks x16
11 dim3 grid(num_blks,1,1);//grid config
12 dim3 thds(num_thds,1,1);//blk config
13 do{
14   Kernel<<<grid,thds>>>(...)
15 while(stop);
```

Listing 2: `BFS` after warp aggregation

increment in grid size (Line 11). We also declare the shared memory as *volatile* for inter-lane data exchanging (Line 2 in Listing 2).

Step 2: Warp Coarsening. If the underlying algorithm hinders the CTA size from being adjusted to warp size, aggressive warp coarsening is needed. We describe it as *aggressive* because the coarsening factor is always the maximum – condensing the number of warps per CTA to one. We want to emphasize that warp aggregation is different from GPU thread coarsening [26–28] in two major ways: (1) Thread coarsening [27, 28] fuses two or more threads in order to increase the amount of work performed per thread but reduces the total thread number. It tries to increase the instruction-level-parallelism (ILP) [26], reduce the number of memory-access instructions [27] and dismiss auxiliary instructions [29]. However, the objective for warp-aggregation is to reduce the number of warps per CTA to one for avoiding significant SCC overheads across warps. (2) For thread coarsening, choosing an appropriate coarsening factor is crucial for performance delivery [27, 28]. But for warp aggregation there is no coarsening factor tuning; we always coarsen CTA size to one warp. Note that since warp coarsening costs extra registers and possibly larger shared memory usage which may lead to occupancy degradation or register spilling, we prefer to avoid warp coarsening if adjusting CTA size (Step 1) is adequate. When the original CTA size is too large (e.g., 32 warps/CTA), in practice we first perform Step 1 to adjust CTA size to the smallest allowable value before conducting coarsening. In general, we do not expect direct performance benefit from warp coarsening itself. We will further discuss warp coarsening and its related concerns in Section 5.

Listing 3-4 depict an example of conducting warp coarsening for the `SOG` kernel shown in Table 3. Since the original kernel fixes its CTA size to 64 threads (i.e., 2 warps) based on algorithm's grid pattern, we are unable to only perform Step 1 to reduce its CTA size to 1 warp. Thus we resort to warp coarsening whose process is described as follows. Since the thread id (i.e., `threadIdx`) is solely used for distinguishing threads in a CTA, in this example we only need to replicate each instruction that depends on `threadIdx.x` along the `x` dimension. As can be seen, variables *i0, g, X, mask* are all replicated once (Line 7-13 in Listing 4). For ***"if"*** statement, we either need to replicate it (Line 24-25 in Listing 3 to Line 19-22 in Listing 4) or convert it to a "*for*" statement with the stride being the warp size (Line 8-10 in Listing 3 to Line 5-6 in Listing 4). Regarding ***"for"*** or ***"while"*** statement, we replicate the whole loop block first and then decide whether to fuse them based on compiler-level dependency check, register pressure and data locality pattern within each loop. For example, the two loops in Line 25-34 in Listing 4 (from Line 30-34 in Listing 3) cannot be fused, while the transformed loops from Line 19-23 in Listing 3 can be fused (Line 14-18 in Listing 4). Warp coarsening performed here can

```
__global__ void sobolGPU_kernel(...){
//smem converted to volatile smem
__shared__ unsigned v[n_direction];
//only threadIdx.x need to be affined
d_directions+=n_directions*blockIdx.y;
d_output+=n_vectors*blockIdx.y;
//"if" statement coverted to "for"
if (threadIdx.x<n_directions){
 v[threadIdx.x]=d_directions[threadIdx.x];
}
//avoid synchronization
__syncthreads();
//var resp. to threadIdx.x to be affined
int iO=threadIdx.x+blockIdx.x*blockDim.x;
int stride=gridDim.x*blockDim.x;
unsigned int g=iO^(iO>>1);
unsigned int X=0;
unsigned int mask;
for(unsigned k=0;k<__ffs(stride)-1;k++){
  mask= - (g & 1);//replicate
  X ^= mask & v[k];//replicate
  g = g >> 1;//replicate
}
if(iO<n_vectors) //replicate or use "for"
 d_output[iO]=(float)X*k_2powneg32;
unsigned v_log2stridem1=v[__ffs(stride)-2];
unsigned v_stridemask=stride-1;
//"for" statement need to replicated,
//can be fused later if possible
for(int i=iO+stride;i<n_vectors;i+=stride){
 X^=v_log2stridem1^v[__ffs(~((i-stride)|
     v_stridemask))-1];
 d_output[i]=(float)X*k_2powneg32;
}}
sobolGPU_kernel<<<dimGrid,64>>>(...);
```

Listing 3: `SOG` **original kernel**

```
__global__ void sobolGPU_kernel(...){
__shared__ volatile unsigned v[n_directions];
d_directions+=n_directions*blockIdx.y;
d_output+=n_vectors*blockIdx.y;
for(int j=threadIdx.x;j<n_direction;j+=32)
  v[j]=d_directions[j];
int iO_0=threadIdx.x+blockIdx.x*64;
int iO_1=threadIdx.x+32+blockIdx.x*64;
int stride=gridDim.x*64;
unsigned g_0=iO_0^(iO_0>>1);
unsigned g_1=iO_1^(iO_1>>1);
unsigned X_0=0; unsigned X_1=0;
unsigned mask_0, mask_1;
for(unsigned k=0;k<__ffs(stride)-1;k++){
   mask_0=-(g_0&1); mask_1=-(g_1&1);
   X_0^=mask_0&v[k]; X_1^=mask_1&v[k];
   g_0=g_0>>1; g_1=g_1>>1;
}
if(iO_0<n_vectors)
   d_output[iO_0]=(float)X_0*k_2powneg32;
if(iO_1<n_vectors)
   d_output[iO_1]=(float)X_1*k_2powneg32;
unsigned v_log2stridem1=v[__ffs(stride)-2];
unsigned v_stridemask=stride-1;
for(int i=iO_0+stride;i<n_vectors;i+=stride){
    X_0^=v_log2stridem1^v[__ffs(~((i-stride)|
       v_stridemask))-1];
    d_output[i]=(float)X_0*k_2powneg32;
}
for(int i=i1_0+stride;i<n_vectors;i+=stride){
    X_1^=v_log2stridem1^v[__ffs(~((i-stride)|
      v_stridemask))-1];
    d_output[i+32]=(float)X_1*k_2powneg32;
}}
sobolGPU_kernel<<<dimGrid,32>>>(...);
```

Listing 4: `SOG` **after aggregation**

be viewed as conducting warp-level vectorization on CTA-level already vectorized code; special caution is required when handling boundary and epilogue. Note that we currently manually perform the code transformation for Warp-coarsening. But it is feasible to implement this process as an automatic code transformation handled by compiler [27].

3.2 Register Remapping: Accelerating Communication

The objective of this technique is to reduce communication overhead under the new execution model. Once a single warp can accomplish the entire workload of a CTA in the original kernel via warp aggregation, inter-warp communication is then converted to inter-lane communication. The basic idea here is to leverage register shuffle to replace data exchange through shared memory in order to enhance communication efficiency. As previously discussed, communication across warps using shared memory requires two register access, two shared memory access and a synchronization. Even for intra-warp communication via shared memory, it requires two register access and two shared memory access as GPU ISA only allows one operand in shared memory space. Additionally, accessing shared memory incurs much higher latency than accessing registers, since the requests need to flow through the memory access pipeline via load-store units.

In comparison, register shuffle has many advantages: (i) it is hardware supported; (ii) no synchronization or memory fence is required; (iii) no intermediate exchange buffer is needed; and (iv) register read and write are faster than shared memory while two register access is sufficient for communication. Experience on early versions of NVCC has suggested that 90% of the performance improvement can be achieved by keeping things in registers [30] and only register-to-register instruction is able to to reach the peak instruction throughput [26, 10]. Thus applying shuffle instructions to replace shared memory instructions may significantly accelerate applications with communication.

However, converting shared memory communication to register shuffle is nontrivial. **First**, shared memory is a unified space, having

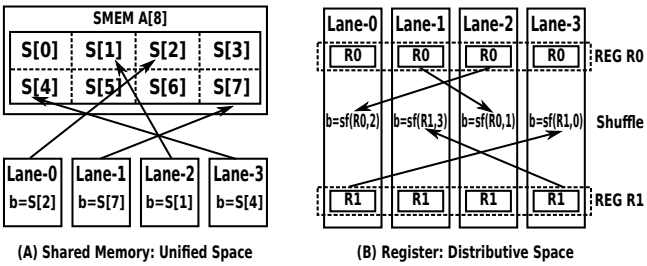

Figure 3: Communication via shared memory to communication via register shuffle. The unified, collective shared memory space is now mapping a separated, distributive local register space per lane, and data is exchanged by shuffling a register, such as R0 or R1.

consistent view from all-lanes and all-warps in the CTA. But register is a lane-private space so lanes cannot directly see each other's stored value. Upon a register access, all the SIMD-lanes in a warp need to have the same reference tag (e.g., `R1`) due to lockstep execution. Thus converting shared memory communication to register shuffle is similar to mapping a collective shared memory access pattern to a distributive memory access pattern, as shown in Figure 3. **Second**, shared memory can be referenced using an address that is calculated at runtime. However, to access a register, the tag (e.g., `R0`, `R1`) must be decided at launch time or even at compile time. Therefore, no runtime address calculation is allowed for allocating variable array in registers; otherwise, local memory which allocated in off-chip global memory space, is employed and performance suffers severely. In other words, kernels with their communication patterns defined at runtime do not gain performance benefit from such conversion. **Third**, the shuffle instructions are executed passively, meaning that it is the object lane that "feeds" its own register value to the subject lane, not the subject lane "fetches" from the object lane's register space. As a result, when the object lane does not participate in the shuffling process (e.g., in a divergent branch), an incorrect value can be transferred. **Finally**, space allocation on shared memory is very flexible – any number of entries can be allocated before hitting the capacity. But for registers, the number must be a factor of 32 since all the lanes must keep the same declaration. So an improper mapping may waste a lot of registers. If the number of shared memory entries is not a factor of 32, register remapping can be complicated and error-prone.

Listing 5-6 show a code transformation example for a stencil kernel (`STL` in Table 3). As can be seen, each element needs to access its surrounding neighbors. For its left and right neighbors, data sharing is originally done via shared memory (i.e., *sh_A0* array). We first apply warp aggregation to remove the synchronization across warps (Line 6, 15, 29, 32 in Listing 5). Based on the shared memory array *sh_A0* size (64 entries), we declare two additional registers per SIMD-lane (i.e., *r0_A0* and *r1_A0* for each of the 32 lanes) for data communication via register shuffle.

When accessing across register spaces (e.g., *r0_A0* to *r1_A0*), an additional register is employed. For example, in Line 25 of the left listing, lane-31 needs to access *sh_A0[sh_id+1]*, which is not in the present register space of *r0_A0* after register remapping but in *r0_A1* of lane-0. Since shuffle instructions executed by a warp must have the same source and destination register (lockstep execution), and rotation shuffle is currently not supported, we allocate another register *a_other* (Line 20 in Listing 6) to hold the value of register *r0_A1* of lane-0.

```
__global__ void block2D_hybrid_coarsen_x(...){
...
//Shared Memeory: [64]
extern __shared__ float sh_A0[];
sh_A0[sh_id]=0.0f; sh_A0[sh_id2]=0.0f;
__syncthreads();
...
float bottom=0.0f,bottom2=0.0f;
float top=0.0f,top2=0.0f;
if((i<nx) &&(j<ny)){
  bottom=A0[Index3D(nx,ny,i,j,0)];
  sh_A0[sh_id]=A0[Index3D (nx,ny,i,j,1)];
}
...
__syncthreads();
for(int k=1;k<nz-1;k++){
 float a_left_right,a_up,a_down;
 if((i<nx)&&(j<ny))
   top=A0[Index3D(nx,ny,i,j,k+1)];
 if(w_region){
  ...
  a_left_right=x_l_bound?A0[Index3D(
      nx,ny,i-1,j,k)]:sh_A0[sh_id-1];
  Anext[Index3D(nx,ny,i,j,k)]=(top+bottom
     +a_up+a_down+sh_A0[sh_id+1]
     +a_left_right)*c1-sh_A0[sh_id]*c0;
 }
 ...
  __syncthreads();
  bottom=sh_A0[sh_id];
  sh_A0[sh_id]=top;
  __syncthreads();
  }
}
dim3 block(32, 4, 1);
dim3 grid(nx+63)/64, (ny+3)/4, 1);
block2D_hybrid_coarsen_x<<<grid,block>>>(...);
```

Listing 5: STL original kernel

```
__global__ void block2D_hybrid_coarsen_x(...){
...
//Register Space: [32]*2
float r0_A0=0.0f; float r1_A0=0.0f;
...
float bottom=0.0f,bottom2=0.0f;
float top=0.0f,top2=0.0f;
if((i<nx) &&(j<ny)){
  bottom=A0[Index3D(nx,ny,i,j,0)];
  r0_A0=A0[Index3D (nx,ny,i,j,1)];
}
...//remove sync
for(int k=1;k<nz-1;k++){
float a_left,a_right,a_up,a_down,a_other;
if((i<nx)&&(j<ny))
  top=A0[Index3D(nx,ny,i,j,k+1)];
//shuffle is outside branch
a_left=__shfl_up(r0_A0,1);//left neighbor
a_right=__shfl_down(r0_A0,1);//right neighbor
a_other=__shfl(r1_A0,0);//2nd reg space
if(w_region){
 ...
 a_left=x_l_bound?A0[Index3D(nx,ny,i-1,j,k)]
       :a_left;
a_right=(threadIdx.x==31)?a_other:a_right;
Anext[Index3D(nx,ny,i,j,k)]=(top+bottom
   +a_up+a_down+a_left
   +a_right)*c1-r0_A0*c0;
}
...
 bottom=r0_A0;
 r0_A0=top;
 }
}
dim3 block(32, 1, 1);
dim3 grid(nx+63)/64, ny, 1);
block2D_hybrid_coarsen_x<<<grid,block>>>(...);
```

Listing 6: STL after reg remapping

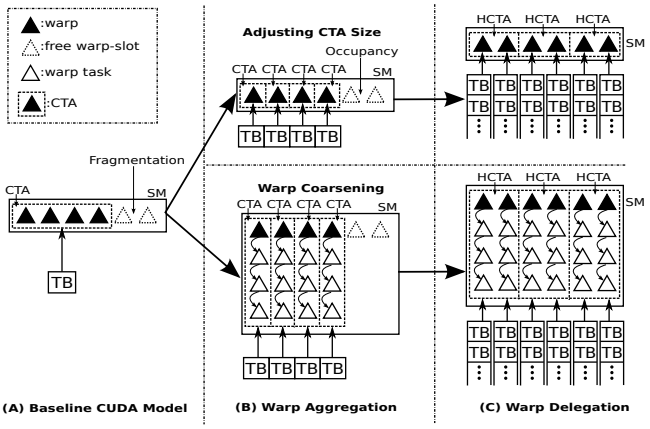

Figure 4: Execution model transformation from CUDA-baseline to warp-aggregation to warp-delegation.

Also note that shuffling (e.g., Line 18-19 in Listing 6) must be put outside of the *if* branch (Line 21 in Listing 6). Otherwise, the branch effect can cause certain lane(s) receive incorrect data value. In Listing 6, if lane-0 is branched out, it falls out of the *w_region* and lane-1 could receive an undefined value when shuffling from left to right (*shfl_down* in Line 19). The same condition may also occur for *shfl_up*.

3.3 Warp Delegation: Enhancing Occupancy

Warp delegation provides a method to tackle the occupancy concern from applying warp aggregation (Section 3.1) which restricts only one warp per CTA. Since the Kepler architecture, each SM supports 64 hardware warp-slots. However, each SM can only accommodate at most 16 CTAs for Kepler, and 32 CTAs for Maxwell and Pascal (Table 2). Consequently, the theoretic occupancy[3] after warp aggregation is at best 25% for Kepler and 50% for Maxwell and Pascal. For instance, suppose in Figure 4-(A) an SM has 6 hardware warp-slots and each CTA contains four warps in the original kernel, two warp-slots are then wasted, known as the *resource fragmentation* issue. After warp-aggregation (i.e., adjusting CTA size or warp-coarsening) shown in Figure 4-(B), each CTA contains only one warp. If an SM here can accommodate at maximum 4 CTAs (i.e., 4 CTA-slots), two warp-slots will be left idle, leading to a lower occupancy. Warp delegation aims to provide users a way to improve occupancy through fine tuning to achieve desired performance. Note that for some applications that are bounded by register usage and ILP [26, 10], increasing occupancy may reduce the overall performance. So leaving it as it is after warp aggregation and register remapping works better for them, shown as the *Hardware Approach* in Stage-3 in Figure 2. We will further discuss this in Section 4 and Section 5. This subsection focuses on the applications that require improved occupancy to achieve better performance after warp aggregation.

The basic idea is to circumvent the restriction from less CTA-slots than warp-slots per SM on contemporary GPUs. Inspired by [31], we propose a software-based warp delegation method to relax this restriction. Our warp delegation approach is different from [31] in several aspects: (a) warp delegation aims to improve occupancy for our execution model rather than clustering CTAs with inter-CTA locality on SMs; (b) warp delegation instructs each **warp** to fetch and execute CTA jobs after warp aggregation, while CTA-clustering applies **CTA**-agents to fetch and execute CTA jobs; (c) warp delegation uses a different indexing method: $warpid * SM + sm\_id$ which addresses the issue when CTA number cannot evenly divide SM number; (d) CTA-clustering requires global memory atomic operations to distinguish among CTAs in an SM, but warp delegation simply relies on hardware warp-slot id (fetched directly from a constant register) to identify the warp-agents in an SM.

In warp delegation, we define the term *Hyper-CTA* or *HCTA*, as shown in Figure 4-(C). Unlike the traditional CTA definition in the CUDA model, HCTA is a super CTA that comprises warps that are processed by warp aggregation stage. These warps are completely

[3]Occupancy is the ratio of the number of active warps per SM to the number of warp-slots per SM, or warp-slots utilization. The theoretic occupancy is calculated theoretically from kernel configuration; the achieved occupancy is measured by real hardware performance counters at runtime.

```
//================== Warp_Consolidation.cuh ====================
#if __CUDA_ARCH__ < 500 //Kepler GPU
#define MAX_CTAS_PER_SM 16 //Each SM host at most 16 CTAs
#define WARPS_PER_CTA 4 //16x4 to saturate all 64 warpslots
#else //Other GPUs
#define MAX_CTAS_PER_SM 32 //Each SM host at most 32 CTAs
#define WARPS_PER_CTA 2 //32x2 to saturate all 64 warpslots
#endif //We want to keep CTAs as small as possible to reduce possible unbalancing
//Modification to the parameter list
#define PARAM const int _ctas, const int _ox, const int _oy
#define CALL(X) ((X).x*(X).y*(X).z), (X).x, (X).y
//Calculate the new CTA coordinate
#define X_PARTITION unsigned bx=vid/_oy; unsigned by=vid%_oy;
#define Y_PARTITION unsigned by=vid/_ox; unsigned bx=vid%_ox;
//Main delegation loop
#define WARP_DELEGATION unsigned sm_id;\
asm("mov.u32 %0,%%smid;":"=r"(sm_id));\//fetch sm-id
unsigned warp_id;asm("mov.u32 %0,%%warpid";:"=r"(warpid));\//fetch hardware warp id
if(warpid>=MAX_CTAS_PER_SM*WARPS_PER_CTA) return;\ //for warp throttling
unsigned laneid;asm("mov.u32 %0,%%laneid";:"=r"(laneid));\//fetch lane id
const unsigned agentid=warpid*_SM+sm_id;\//we use warps as CTA agents
const unsigned nagents=MAX_CTAS_PER_SM*WARPS_PER_CTA*_SM;\//maximum persistent warps
for(unsigned vid=agentid;vid<_ctas;vid+=nagents)//job loop for each warp agent
```

Listing 7: Warp delegation header file

```
__global__ void
Kernel(int* A){
 Kernel_Body;
}
//kernel after warp
//aggregation and
//register remapping
kernel<<<grid,32>>>(...);
```

Listing 8: Original kernel

```
#include "Warp_Delegation.cuh"
__global__ void Kernel(int* A, PARAM){
WARP_DELEGATION{
   X_PARTITION;//or Y_PARTITION
   Kernel_Body;//use bx, by to replace blockIdx.x & y
} //and _ox, _oy to replace gridDim.x & y
} kernel<<<MAX_CTAS_PER_SM*_SM,
          WARPS_PER_CTA*32>>>(A, CALL(grid));
```

Listing 9: Transformed kernel by warp delegation

Table 2: GPU SM resource configuration. "Reg" refers to the number of 4B register entries.

GPU	Arch/CC	Runtime	SMs	CTAs/SM	Warps/SM	Warps/CTA	Reg/SM	Reg/CTA	Reg/Thd	Shared/SM	Shared/CTA
Tesla-K80	Kepler-3.7	7.5	15	16	64	32	128K	64K	255	112KB	48KB
Tesla-M40	Maxwell-5.2	8.0	24	32	64	32	64K	64K	255	96KB	48KB
Tesla-P100	Pascal-6.0	8.0	56	32	64	32	64K	64K	255	64KB	48KB
Tesla-V100	Volta-7.0	9.0	80	32	64	32	64K	64K	255	96KB	96KB

Table 3: Benchmark Characteristics. "WPs" stands for warps per CTA. "CTAs" means the default number of CTAs per SM in the baseline. "Regs" represents the number of registers per thread in the baseline. "SMem" is the shared memory requirement (in bytes) per CTA in the baseline. "Sync" represents whether inter-warp synchronization (i.e., __syncthreads()) exists in the kernel. "Comm" indicates whether inter-warp communication appears in the kernel. "Ref" refers to the source benchmark-suit.

Application	Description	abbr.	Kernel Name	CTAs	WPs	Regs	SMem	Sync	Comm	Ref
64H	64 and 256 bin histogram calculation	64H	*mergeHistogram256Kernel()*	256	8	15	1000B	Y	Y	[32]
streamcluster	Assigning points to nearest centers	STR	*kernel_compute_cost()*	128	16	25	0B	N	N	[33]
atax	Matrix transpose and vector multiply	ATX	*atax_kernel2()*	256	8	18	0B	N	N	[34]
bh	Gravitational forces via Barnes-Hut method	BHH	*BoundingBoxKernel()*	168	32	32	24000B	Y	Y	[35]
mersenne	Mersenne Twister random generator	MEE	*BoxMullerGPU()*	32	4	18	0B	N	N	[32]
binomial	Option call price via binomial model	BIL	*binomialOptionsKernel()*	1024	8	17	2007B	Y	Y	[32]
BlackScholes	Option call price via Black-Scholes model	BLS	*BlackScholesGPU()*	480	4	21	0B	N	N	[32]
corr	Correlation computation	COR	*corr_kernel()*	8	8	22	0B	N	N	[34]
cutcp	Compute Coulombic potential for 3D grid	CPP	*cenergy()*	512	4	22	0B	N	N	[36]
fft	Fast Fourier transform	FFT	*IFFT512_device()*	32768	2	56	4500B	Y	Y	[37]
stencil	Jacobi stencil operation on regular 3D grid	STL	*block2D_hybrid_coarsen_x()*	64	4	29	1000B	Y	Y	[36]
hotspot	Estimate processor temperature	HOT	*calculate_temp()*	7396	8	38	3000B	Y	Y	[33]
lps	3D Laplace solver	LPS	*GPU_laplace3d()*	5000	4	16	2390B	Y	Y	[38]
matrixMul	Matrix multiplication	MAL	*matrixMulCUDA()*	16384	32	27	8000B	Y	Y	[32]
mum	Pairwise local sequence alignment for DNA	MUM	*mummergpuKernel()*	196	8	23	0B	N	N	[38]
scalarProd	Scalar products of input vector pairs	SCD	*scalarProdGPU()*	4096	8	24	4000B	Y	Y	[32]
sgemm	Single precision general matrix multiply	SGM	*mysgemmNT()*	496	4	46	512B	Y	Y	[36]
recursiveGaussian	Recursive Gaussian filter	REN	*d_transpose()*	1024	8	10	1062B	Y	Y	[32]
nbody	All-pairs gravitational N-body simulation	NBY	*integrateBodies()*	224	8	36	4000B	Y	Y	[32]
pathfinder	Dynamically finding a path in 2D grid	PAR	*dynproc_kernel()*	3334	1	13	256B	Y	Y	[33]
MonteCarlo	Option call price via Monte-Carlo method	MOO	*MonteCarloReduce()*	1024	8	29	4000B	Y	Y	[32]
bfs	Breadth first search	BFS	*Kernel()*	1954	16	16	0B	Y	N	[33]
b+tree	B+tree Operation	B+E	*findK()*	10000	8	27	0B	Y	N	[33]
SobolQRNG	Sobel edge detection filter for images	SOG	*sobolGPU_kernel()*	25600	2	19	128B	Y	Y	[32]
dct8x8	Discrete cosine transform for 8x8 block	DC8	*CUDAkernel1DCT()*	4096	2	18	500B	Y	Y	[32]
srad	Speckle reducing anisotropic diffusion	SRD	*reduce()*	2048	16	25	4000B	Y	Y	[33]
backprop	Perceptron back propagation	BAP	*bpnn_layerforward_CUDA()*	32768	8	18	1062B	Y	Y	[33]
gesummv	Scalar vector and matrix multiplication	GEV	*gesummv_kernel()*	128	8	23	0B	N	N	[34]
gaussian	Solving variables in a linear system	GAN	*Fan1()*	2	16	12	0B	N	N	[33]
single	Monte Carlo single Asian option	SIE	*initRNG()*	782	4	32	0B	Y	Y	[32]
syr2k	Symmetric rank-2k operations	SY2	*syr2k_kernel()*	16384	8	19	0B	N	N	[34]
syrk	Symmetric rank-k ooperations	SYK	*syrk_kernel()*	4096	8	28	0B	N	N	[34]

isolated, free to allocate, execute and free an original CTA's job without interfering with each other. At SM level, HCTAs stay persistently on an SM and iteratively fetch from the original CTAs' task set. In this way, warp delegation can successfully circumvent the restriction from the limited hardware CTA slots to improve occupancy. Meanwhile, this method can also resolve the resource fragmentation issue of the original kernel. For instance, Figure 4-(C) shows that all the hardware warp-slots can be filled under warp delegation. Note that in our proposed Warp-Consolidation model, **HCTAs are hidden** from users: as long as the desired occupancy can be achieved, a different CTA formation can be applied.

Listing 7 shows the implementation of warp delegation. Users can simply incorporate this header file into their code. In Line 20-21, each warp fetches its hardware warp-slot id to calculate its unique warp-agent id for this kernel context. Then all the warp agents (*nagents* in Line 22) are employed to traverse the CTA task set (Line 23). Listing 8-9 show how to integrate the header for warp delegation. *X_PARTITION* or *Y_PARTITION* still needs to be specified to identify the HCTAs' x and y coordinates in the new kernel grid.

Although warp delegation can effectively increase occupancy for our Warp-Consolidation model and tackle the resource fragmentation problem [6], a high occupancy may not necessarily lead to higher performance due to potentially extra congestion in cache, NoC and off-chip bus [26, 10]. To better accompany warp delegation and enable finer-tuning in Warp-Consolidation model, we apply *warp-throttling* to control the number of warps being active (Line 19 in Figure 7) to reach optimal performance. A more dynamic runtime throttling targeting cache congestion can be implemented via [39].

Although in this work we perform such code transformation manually due to the fact that the vendors do not release compiler-level details, the major goal here is to demonstrate the feasibility and potential benefit of the idea. Algorithm 1 summarizes the major procedure of kernel transformation for warp-consolidation. By following it, the three stages can be integrated into an optimization compiler to realize auto code transformation, similar to [27, 40, 41].

Algorithm 1: Warp Consolidation

```
Input  : Original Kernel Code
Output : Warp-Consolidation based Optimized Kernel Code
Stage-1 Warp Aggregation(kernel code):
    if CTA-size can be adjusted to 32 threads then
        Adjust CTA-size to 32 threads;
    else
        Perform aggressive warp coarsening;
        for each code block separated by __syncthreads() do
            Replicate each var reused across the code block;
            Replicate statements depending on threadIdx.x;
            Convert "if" statements to "for" statements;
            Replicate "for" and "while" block;
            Perform loop fusion if possible;
        end
    end
    Remove all sync statements;
    return optimized code after warp-aggregation;
Stage-2 Register Remapping(kernel code):
    if Fixed s-mem access pattern and no addr calculation then
        Remove all shared memory allocation;
        Partition smem space into warp-based chunks;
        Allocate these chunks in different registers;
        Replace smem references by shuffle instructions;
    end
    return optimized code after register-remapping;
Stage-3 Warp Delegation(kernel code):
    Include the Warp_Delegation.cuh header file;
    Pack kernel body into the WARP_DELEGATION macro;
    Set partition strategy;
    Convert kernel invokation;
    return optimized code after warp-delegation;
```

4 EVALUATION

4.1 Evaluation Methodology

We evaluate the proposed Warp-Consolidation execution model and its techniques including *warp aggregation*, *register remapping* and *warp delegation* on four NVIDIA high-end Tesla GPUs, covering all the recent generations of NVIDIA GPU architectures: *Kepler*, *Maxwell*, *Pascal* and *Volta*, as listed in Table 2. For generality, we select 32 representative applications from popular GPU benchmark suits, covering various science domains and kernel features as listed in Table 3. The baseline here is the original code provided by the benchmark packages without any modification. We use large inputs for all the application runs to better reflect real-world workloads. In addition to the three stage optimization discussed in Section 3, we also evaluate *warp throttling* to further fine-tune optimal performance for Warp-Consolidation model, which is discussed in Section 3.3. Results shown in this section are the average of multiple runs and their outputs have been compared with the original kernels' outputs to guarantee correctness.

4.2 Overall Results Analysis

Figure 5, 6, 7 8 show the overall performance improvement by applying each stage of warp-consolidation optimization incrementally, marked as "*Baseline*", "*Aggregate*", "*Shuffle*" and "*Delegation*". "Throttle" is also included last to provide additional fine-tuning for performance. The cyan lines based on the right axis show the *achieved occupancy* measured by CUDA profiler, which imply the actual levels of TLP at runtime. Note that when profiling achieved occupancy for `COR, SY2, GEV` on Tesla-K80, the profiler reports performance counter overflow due to their lower bit-width design on Fermi and Kepler, so the achieved occupancy for these three applications on K80 appears as zero. L1/Tex cache is turned on for these experiments.

Together with warp-throttling, our Warp-Consolidation execution model achieves **1.7x** (up to 6.3x), **2.3x** (up to 31.1x), **1.5x** (up to 6.4x) and **1.2x** (up to 3.8x) on Kepler-K80, Maxwell-M40, Pascal-P100 and Votal-V100 GPUs, respectively. For applications containing synchronization and shared memory communication (e.g., `64H`, `BHH`, `STL`, `HOT`, `PAR`, `MOO`, `SOG`, `DC8`, `SRD`, `BAP`, `SIE`, etc. See "*Sync*" and "*Comm*" columns in Table 3), both warp-aggregation and register-remapping can lead to significant performance gain by avoiding cross-warp synchronization and less-efficient shared memory communication. For applications without synchronization and communication (e.g., `STR`, `ATX`, `COR`, `CPP`, `MUM`, `GEV`, `SY2`, `SYK`, etc.), warp-aggregation (*Aggregate* and *Shuffle* show the same performance here as no register-remapping is applied) degrades performance, suggesting that when no SCC types of overheads need to be avoided, occupancy degradation may lead to performance decrease. On the other hand, warp delegation has demonstrated its effectiveness for improving their performance because it resolves the resource fragmentation issue for better occupancy. The performance of applications such as `GEV` and `SYK` is bounded by congestion in cache, NoC or off-chip bus, thus warp throttling can effectively mitigate such contention. For `MAL` and `REN`, register-remapping (Shuffle) degrades the performance significantly. This is because the shared memory addresses used for communication are only known at runtime for these applications, so when applying register-remapping, local memory rather than registers is used, causing high overheads from accessing global memory (Section 3.2). We validate this by observing high local memory transaction increase in the profiler.

Comparing across the four architectures, we can observe that our approach achieves slightly better performance on Kepler than other architectures (particularly for `CPP`, `MAL`, `PAR`, `MOO`, `DC8`, `SRD` and `GEV`). This is because an SM on Kepler has more scalar cores (192 cores/SM for Kepler vs.128 cores/SM for Maxwell and 64 cores/SM for Pascal and Volta), so it is easier to explore optimization opportunities via Warp-Consolidation techniques on Kepler since the SCC degree (e.g., higher synchronization and communication intensity) and memory hierarchy contention are relatively higher. Resolving these bottlenecks on Kepler via warp-aggregation (e.g., `MAL`, `PAR`, `GEV`) and warp-throttling (e.g., `MOO`, `DC8`, `BAP`) result in good performance gain. Moreover, from the occupancy curve, we can observe that warp-delegation technique can effectively address the occupancy degradation caused by warp aggregation (e.g., `STR`, `ATX`, `HOT`, `LPS`, `SCD`, etc). On the other hand, the best performance delivered by warp-throttling for applications such as `BIL`, `PAR`, `MOO`, `SY2` suggest that occupancy improvement does not necessarily lead to better performance. Finally, for Volta, the benefit is not as significant as the others. This is mainly due to the baseline performance improvement for applications such as `SCD`, `SGM`, `SGM`, `SY2` and `SYK`. More specifically, there are three main reasons result in such improvement: (i) Volta supports independent thread scheduling (see Section 2.3), which significantly reduces warp control divergence overhead, consequently mitigating the probability and degree of inter-warp unbalancing; (ii) In Volta, FP32 and Int32 instructions can be executed simultaneously, which reduces the overhead for address calculation in divergent memory access, lowering the possibility of false-waiting among warps; (iii) the L1 cache speed, capacity and bandwidth has been dramatically enhanced in Volta, which also contributes to the lower divergence overhead as well as inter-warp imbalance.

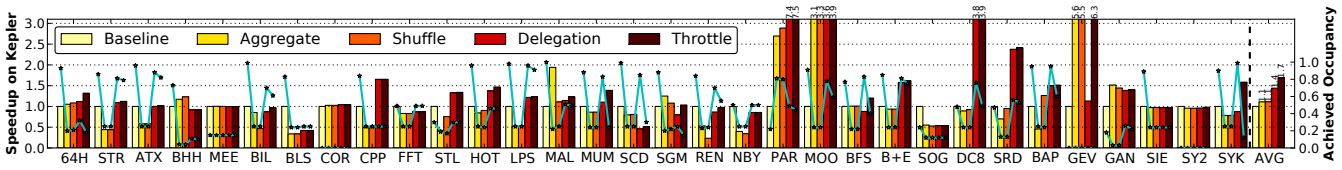

Figure 5: Speedup and Achieved Occupancy on Kepler-based NVIDIA Tesla-K80 GPU. The cyan star-line indicates achieved occupancy. "AVG" is the average.

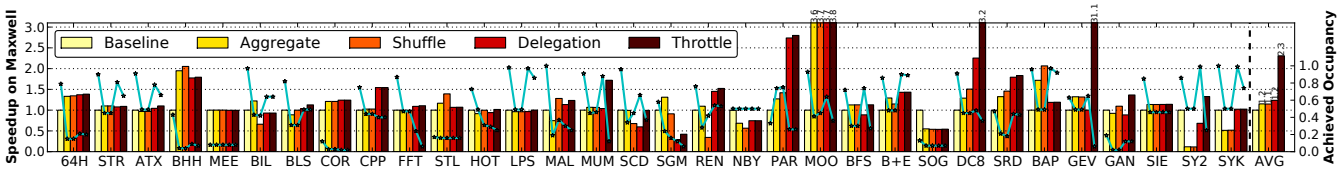

Figure 6: Speedup and Achieved Occupancy on Maxwell-based NVIDIA Tesla-M40 GPU. The cyan star-line indicates achieved occupancy. "AVG" is the average.

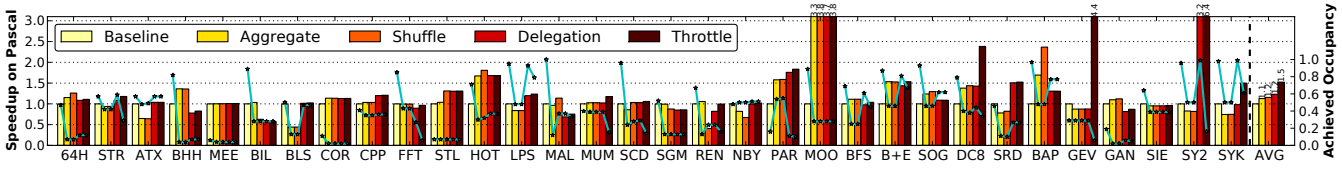

Figure 7: Speedup and Achieved Occupancy on Pascal-based NVIDIA Tesla-P100 GPU. The cyan star-line indicates achieved occupancy. "AVG" is the average.

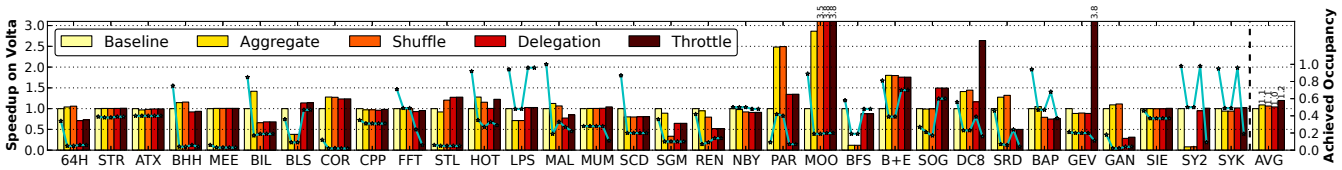

Figure 8: Speedup and Achieved Occupancy on Volta-based NVIDIA Tesla-V100 GPU. The cyan star-line indicates achieved occupancy. "AVG" is the average.

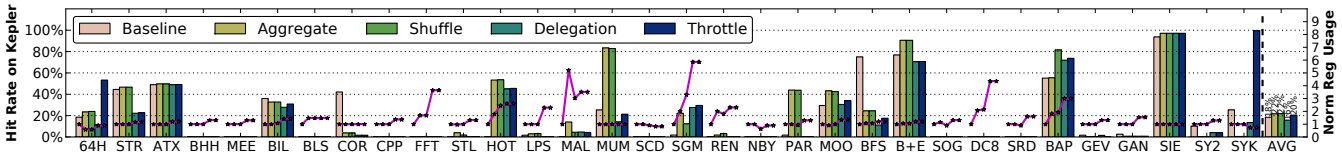

Figure 9: L1 Cache hit-rate for global memory access on Kepler Tesla-K80 GPU. The purple star-line indicates the normalized register usage per thread.

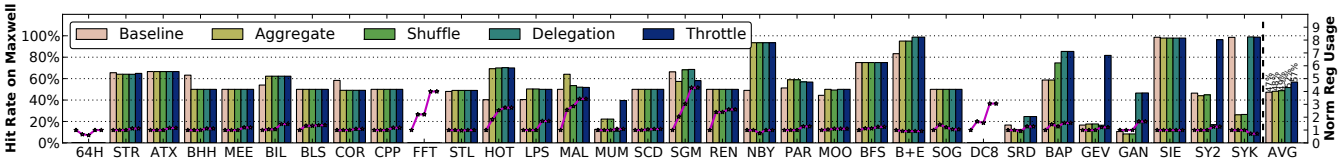

Figure 10: L1 Cache hit-rate for global memory access on Maxwell Tesla-M40 GPU. The purple star-line represents the normalized register usage per thread.

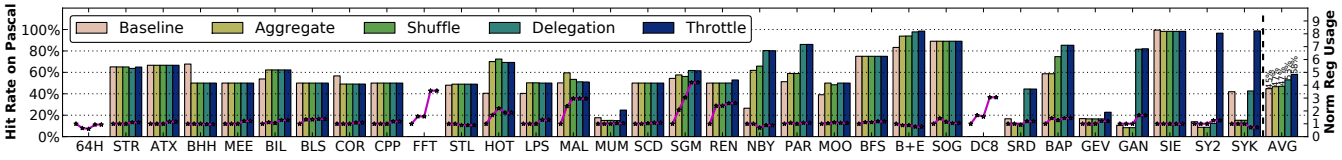

Figure 11: L1 Cache hit-rate for global memory access on Pascal Tesla-P100 GPU. The purple star-line represents the normalized register usage per thread.

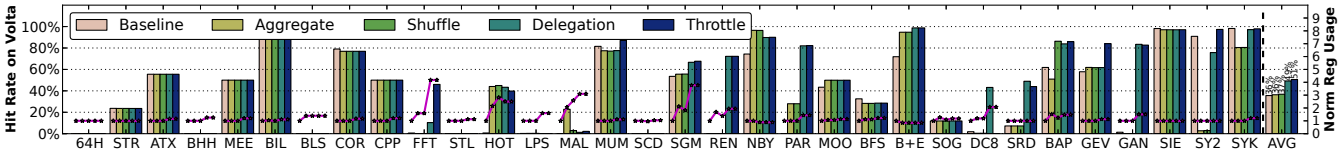

Figure 12: L1 Cache hit-rate for global memory access on Volta Tesla-V100 GPU. The purple star-line represents the normalized register usage per thread.

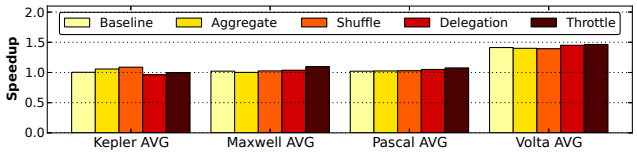

Figure 13: The average performance speedup across all 32 applications when turning off L1/Tex cache on three GPU architectures.

4.3 Impact from Cache and Registers

To further explore the impact from cache and register files, we profile the L1 cache hit rate for global access as well as the normalized register usage per thread (purple lines based on the right axis) through *nvprof* [42], shown in Figure 9, 10, 11 and 12 for Kepler, Maxwell, Pascal and Volta, respectively. Note that Kepler architecture uses both L1 and texture cache [43] for global memory access. Thus some applications such as BHH, MEE, MUM, NBY, etc., show a different L1 cache behavior in Figure 9 from that on Maxwell and Pascal. We have several observations. (1) Much higher L1 cache hit rates for GEV, SY2 and SYK are observed after applying warp-throttling on Maxwell and Pascal, indicating that these applications are primarily contention-bound at L1. The significantly increased hit rate for these three applications on Volta explain the degraded speedup from warp delegation in Figure 8, supporting our arguments in Section 4.2. (2) Regarding warp-aggregation, sometimes the cache efficiency increases (e.g., BIL, HOT, MUM, MOO, B+E) due to better and more random exploitation of the inter-CTA locality, while sometimes hit rate drops (e.g., BHH, COR, BFS) due to less intra-CTA reuse. Overall, the primary performance gain from our Warp-Consolidation execution does not directly come from better utilization of cache (i.e., only slightly higher average hit rate) but rather from reducing SCC related stalls and resource fragmentation issues. Figure 13 shows the average performance gain for each scheme and architecture by enabling L1/Tex cache across the 32 applications. As can be seen, although Volta shows great performance speedup by enabling L1/Tex cache amongst architectures, the speedups among schemes for a particular architecture are very consistent, implying that they benefit similarly from the cache. Therefore, the performance gain achieved by Warp-Consolidation execution is essentially from reducing SCC overheads through more efficient data communication and better utilization of the on-chip resources (registers, shared memory, hardware warp slots, NoC bandwidth, etc) rather than cache. For register impact, Figure 9-11 show that the register usage per thread does not increase much with warp aggregation, much smaller than the aggregation factor.

5 EXTENDED DISCUSSION AND REFLECTION

By summarizing the observations from Figure 5 to Figure 13, we extend our discussion here including additional observations, solutions to concerns, compatibility to technology trend, and suggestions for writing new algorithms.

(I) Parallelism and Occupancy. There is a design trend with GPU vendors such as NVIDIA that the number of sustainable CTAs per SM keeps increasing with architecture evolution, e.g., from 8 in Fermi, to 16 in Kepler, to 32 in Maxwell and Pascal; but the hardware warp-slot number per SM remains largely unchanged, e.g., 48 for Fermi, 64 for the rest. This trend suggests that GPU hardware design is enhancing its ability to support more light-weighted CTAs. More importantly, our software-based warp-delegation technique

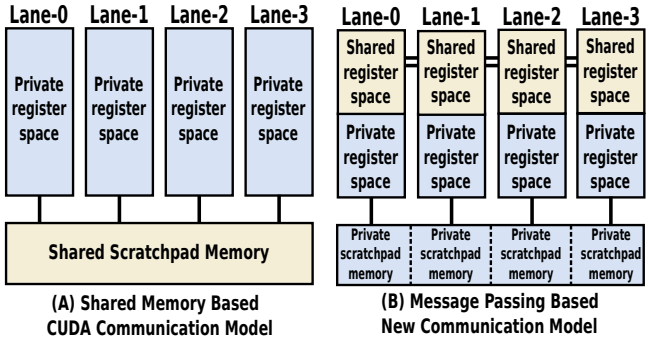

Figure 14: Traditional CUDA communication model vs. Warp-Consolidation communication model.

can effectively address the occupancy issue from mismatched CTA-slot number and warp-slot number, demonstrated by significant performance improvement from applications such as STR, BIL, CPP, LPS, B+E, DC8, and SRD. Additionally, we want to emphasize that our Warp-Consolidation execution model is a forward-looking technique that future GPU applications with a larger problem size or CTA number requirement can expect further enhanced performance under it. For example, from Table 2 we can calculate that, as current Tesla-V100 has 80 SMs, it requires $80 \times 64 = 5120$ warps to fully utilize all the available hardware warp-slots after Warp-Consolidation. For Tesla-K80, M40 and P100, this number is 960, 1536 and 3584. However, Table 3 shows that today's applications often do not meet this requirement for max hardware utilization. This observation indicates that our model could be a very promising tool for driving future big-data algorithm or application design on GPUs.

(II) Register Usage. There are some concerns on the potential of excessive usage of registers after warp coarsening. Our experiments (Figure 5-7) as well as other studies have shown that a good usage of more registers could actually bring significant speedup, greatly surpass the possible negative effects on occupancy. Figure 9-11 also show that Warp-Consolidation only expands register usage for some applications (e.g., HOT, FFT, MAL, SGM and DC8); but for the majority of them, our method does not consume much additional registers, especially on Maxwell, Pascal Volta. Furthermore, as the largest on-chip SRAM structure on GPUs, register files are often not fully utilized. Previous studies [6, 44] have indicated that nearly 30% of the GPU registers are not used at all during program execution. Thus extra usage of registers in Warp-Consolidation execution may not necessarily lead to occupancy reduction. This could become less an issue in future since the volume of registers on GPUs is continuously increasing across generations. For certain cases that register consumption really becomes a problem under our execution model, there are two possible solutions: *spilling into shared memory* and *register packing* [45, 46]. For the former, Figure 14 demonstrates the traditional shared memory based CUDA communication model (A) and a new communication model under our Warp-Consolidation (B). Under this setup, using shared memory as thread private space for spilling is more efficient than as a shared space. This is because when spilling occurs, all the lanes will spin at the same time with the same register tag, making these spilling access always aligned and coalesced. This ensures full speed of shared memory as no bank conflicts will occur. Refer to [47–49] for specific implementation examples. For the latter, several related techniques have been proposed previously [45, 46]. With recent GPU support for short-bitwidth data-types, e.g., Int8, Int16 and FP16, register packing may

become very attractive for certain applications. Finally, to comprehensively analyzing the interaction among various factors such as parallelism, data size, register usage and cache effect, one may refer to a visualizable performance model [50–52] or GPU profiling tools [53, 54].

(III) Shared Memory. Warp aggregation, especially warp coarsening, may also lead to extra shared memory usage. Based on our experimental observations, after the transformation stage of register-remapping, shared memory is not likely used under our model. But for the cases that shared memory usage does become a limiting factor, we can adopt the shared memory multiplexing technique [55]. As shown in Listing 7, since there are multiple warp-agents per CTA and shared memory is often exploited in one execution phase, multiplexing among these warp-agents is feasible after warp delegation.

(IV) Developing New Algorithms. In addition to applying Warp-Consolidation model to transform legacy codes, future new algorithms can also be implemented directly through it. We have several suggestions to assist this design effort. (1) Since a warp is granted the same ability as a traditional CTA under our model, a similar but more fine-grained partition strategy for CUDA can be utilized. (2) Bigger problem and more CTAs are desired to saturate all the hardware warp-slots on modern GPUs which tend to have a lot more SMs. (3) Data exchanging in our model is achieved via register shuffling, which is a message-passing communication model. An efficient implementation should follow a distributed design pattern and seek to finalize the communication pattern at compile time rather than computing addresses at runtime. (4) The local memory should now be allocated into warp-private partition in the shared memory space (Figure 14) rather than off-chip global memory. This can be achieved through either manual modification or compiler.

6 RELATED WORK

In the traditional CUDA model [1], task mapping contains two levels: first level is from thread blocks to CTAs and the second level is from threads to SIMD-lanes. However, warp is considered transparent. To address irregular applications, *warp-centric* [56] and *warp-specialization* [15] techniques were proposed. *Warp-centric* enables only a single SIMD-lane for the sequential phase (SISD) but activates the entire warp in the parallel phase (SIMD). Warp-specialization, on the other hand, initiates multiple warps to correspondingly handle the different portions of a task and cooperatively generate the results, similar to MISD. The two approaches, albeit different, both rely on the key observation that warps can be executed independently. Divergence across warps, unlike lane-level divergence, does not necessarily incur performance degradation. Unlike CUDA task mapping, theirs at first level is from thread block to CTA, but at the second level is from threads to warps rather than threads to SIMD-lanes warp lane. Their basic observation is that, by branching across warps, expensive SIMD divergence overhead caused by SIMD-lane divergence within a warp can be avoided. This can be seen as hiding SIMD-lanes but exposing warps for execution. Although parallelism degree is reduced, benefits from eliminating SIMD-lane divergence often outweighs degraded parallelism, especially for irregular applications. Our proposed model maps from thread blocks to warps as the first level, and threads to lanes as the second level. The expected benefit from our approach is not from eliminating warp divergence, but the SCC overheads across warps. Our method works for both regular and irregular applications.

Regarding warp-aggregation and synchronization, several hardware-based approaches [20, 57, 16, 6, 58, 2] have been proposed to increase resource utilization and enhance performance via execution granularity manipulation and better warp scheduling. For example, Narasiman et al. [20] proposed a hardware approach to dynamically partition large warps into smaller subwarps for better utilization of the compute resources and reducing warp-divergence overhead. Orr et al. [57] proposed a GPU architecture that could dynamically aggregate asynchronously produced fine-grained thread/warp tasks into coarse-grained CTA tasks for execution. Wang et al. [16] observed that for irregular applications, dynamically spawned tasks could be launched in a granularity of CTAs, rather than heavy-weighted kernel grid. Xiang et al. [6] focused on the resource fragmentation issue, and proposed hardware solutions to allocate and release resources based on warps. When resources underutilization occurs, they dispatched a partial CTA to exploit the unused resources. They reported difficulties when applications contain shared memory usage and synchronization. Lee et al. [58] targeted on the execution time disparity by different warps, and proposed hardware modification to schedule critical warps first and more often. Liu et al.[2] concentrated on reducing the synchronization overhead across warps, and also proposed warp-scheduler modification to prioritize critical warps. All these approaches require hardware modifications. Our design is purely software-based and can be directly deployed on commodity GPUs.

Regarding register shuffling, both [59] and [60] proposed to combine registers from multiple lanes to form a pool which could be employed as a user-managed last-level-cache (LLC). The limitation, however, is that only threads in a warp can access such pool; inter-warp sharing is not possible. Also, the pool is only beneficial when the data stored in the pool can be reused. Managing the pool is also difficult and requires significant user effort. In comparison, our approach uses registers as channels for inter-lane communication, rather than a cache. Also, all the communication under our model occur within a warp.

For warp-delegation, Xiao et al. [11] allocated a guard CTA persistently staying on an SM to coordinate inter-CTA synchronization. Pai et al. [61] proposed to initiate an elastic kernels staying on SMs to serve CTA tasks from a series of application kernels in a task queue, in order to make a better on-chip resource utilization. Li et al.[31] observed the possibility of exploiting inter-CTA locality and proposed an approach to cluster CTAs with good inter-CTA locality on the same SMs. However, all these approaches work on the conventional CUDA execution model and use persistent CTAs to fetch and execute CTA tasks. On contrary, our work applies persistent warps after warp aggregation to execute CTA task set. The execution model is very different. In addition, our major optimization goal is to reduce SCC and fragmentation overheads.

7 CONCLUSION

In this paper, we focus on optimizing the overheads from inter-warp SCC (synchronization, cooperation and communication) and propose a novel execution model that hides the CTA hierarchy but exposes independent warp-level execution. The basic idea is to replace more expensive inter-warp SCC with much cheaper intra-warp SCC in order to achieve significant performance gains. We propose three optimization stages: *warp aggregation, register remapping and warp delegation*, to drastically reduce SCC and fragmentation overheads under controllable occupancy. Evaluation on all the recent

generations of NVIDIA GPU architectures have demonstrated the effectiveness, applicability and portability of our approach. Through a thorough discussion on solutions to possible concerns, compatibility to technology trend and development of new algorithms, we believe our novel execution model is a forward-looking technique that can effectively address future big-data GPU applications.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments. This research is supported by U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under the "CENATE" project (award No. 66150), and the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie "TICOH" project (grant No. 752321). The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

REFERENCES

[1] NVIDIA. CUDA Programming Guide, 2017.
[2] Yuxi Liu, Zhibin Yu, Lieven Eeckhout, Vijay Janapa Reddi, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Chengzhong Xu. Barrier-Aware Warp Scheduling for Throughput Processors. In *ICS-16*. ACM.
[3] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained GPU performance. In *ISCA-12*. IEEE.
[4] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. SIMD re-convergence at thread frontiers. In *MICRO-11*. ACM.
[5] Wilson WL Fung and Tor M Aamodt. Thread block compaction for efficient SIMT control flow. In *HPCA-11*. IEEE.
[6] Ping Xiang, Yi Yang, and Huiyang Zhou. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *HPCA-14*. IEEE.
[7] Rachata Ausavarungnirun, Saugata Ghose, Onur Kayiran, Gabriel H Loh, Chita R Das, Mahmut T Kandemir, and Onur Mutlu. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *PACT-15*. IEEE.
[8] Ang Li, Wenfeng Zhao, and Shuaiwen Leon Song. BVF: enabling significant on-chip power savings via bit-value-favor for throughput processors. In *MICRO*. ACM, 2017.
[9] David Tarjan and Kevin Skadron. On demand register allocation and deallocation for a multithreaded processor, December 29 2009. US Patent App. 12/649,238.
[10] Vasily Volkov. Better performance at lower occupancy. In *GTC-10*.
[11] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. In *IPDPS-10*. IEEE.
[12] Jeff A Stuart and John D Owens. Efficient synchronization primitives for GPUs. *arXiv preprint arXiv:1110.4623*, 2011.
[13] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. Fine-grained synchronizations and dataflow programming on GPUs. In *SC-15*. ACM.
[14] NVIDIA. Volta Architecture White Paper, 2018.
[15] Michael Bauer, Sean Treichler, and Alex Aiken. Singe: leveraging warp specialization for high performance on GPUs. In *PPoPP-14*. ACM.
[16] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs. *ISCA-16*.
[17] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS-10*. IEEE.
[18] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *SC-11*. ACM.
[19] NVIDIA. Parallel Thread Execution ISA, 2017.
[20] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving GPU performance via large warps and two-level warp scheduling. In *MICRO-11*. ACM.
[21] Bryan Catanzaro. LDG and SHFL Intrinsics for arbitrary data types, 2014.
[22] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *EuroPar-16*. Springer.
[23] Weifeng Liu, Ang Li, Jonathan D Hogg, Iain S Duff, and Brian Vinter. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. *Concurrency and Computation: Practice and Experience*, 2017.
[24] Justin Luitjens. Faster Parallel Reductions on Kepler, 2014.
[25] NVIDIA. CUDA C Best Practice Guide, 2017.
[26] Vasily Volkov and James W Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC-08*. IEEE.
[27] Yi Yang, Ping Xiang, Jingfei Kong, Mike Mantor, and Huiyang Zhou. A unified optimizing compiler framework for different GPGPU architectures. *TACO*, 2012.
[28] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *PACT-14*. IEEE.
[29] Alberto Magni, Christophe Dubach, and Michael FP O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *SC-13*. IEEE.
[30] Mike Murphy. NVIDIAs Experience with Open64. In *Open64 Workshop at CGO*, 2008.
[31] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. Locality-Aware CTA Clustering for Modern GPUs. In *ASPLOS-17*. ACM.
[32] NVIDIA. CUDA SDK Code Samples, 2015.
[33] 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 *IISWC-09*. IEEE.
[34] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *InPar-12*. IEEE.
[35] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS-09*. IEEE.
[36] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.
[37] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU-10*. ACM.
[38] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS-09*. IEEE.
[39] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. Adaptive and transparent cache bypassing for GPUs. In *SC-15*. IEEE.
[40] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *TACO*, 2013.
[41] Jun Shirako, Akihiro Hayashi, and Vivek Sarkar. Optimized two-level parallelization for GPU accelerators using the polyhedral model. In *CC-17*. ACM.
[42] NVIDIA. CUDA Profiler User's Guide, 2018.
[43] NVIDIA. Kepler GK110 Whitepaper, 2013.
[44] Mohammad Abdel-Majeed and Murali Annavaram. Warped register file: A power efficient register file for GPGPUs. In *HPCA-13*. IEEE.
[45] Andrew Davidson and John D Owens. Register packing for cyclic reduction: A case study. In *GPGPU-11*. ACM.
[46] Andrew Davidson, David Tarjan, Michael Garland, and John D Owens. Efficient parallel merge sort for fixed and variable length keys. In *InPar-12*. IEEE.
[47] Ari B Hayes and Eddy Z Zhang. Unified on-chip memory allocation for SIMT architecture. In *SC-14*. ACM.
[48] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *MICRO-15*. ACM.
[49] Ang Li, Shuaiwen Leon Song, Akash Kumar, Eddy Z Zhang, Daniel Chavarría-Miranda, and Henk Corporaal. Critical points based register-concurrency autotuning for GPUs. In *DATE-16*.
[50] Ang Li, YC Tay, Akash Kumar, and Henk Corporaal. Transit: A visual analytical model for multithreaded machines. In *HPDC-15*. ACM.
[51] Ang Li, Shuaiwen Leon Song, Eric Brugel, Akash Kumar, Daniel Chavarria-Miranda, and Henk Corporaal. X: A comprehensive analytic model for parallel machines. In *IPDPS-16*. IEEE.
[52] Ang Li, Weifeng Liu, Mads RB Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels. In *SC-17*. ACM.
[53] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R Johnson, David Nellans, Mike O'Connor, and Stephen W Keckler. Flexible software profiling of GPU architectures. In *SC-15*. ACM.
[54] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. Cudaadvisor: Llvm-based runtime profiling for modern gpus. In *CGO-18*. ACM.
[55] Yi Yang, Ping Xiang, Mike Mantor, Norm Rubin, and Huiyang Zhou. Shared memory multiplexing: a novel way to improve GPGPU throughput. In *PACT-12*. ACM.
[56] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *PPoPP-11*. ACM.
[57] Marc S Orr, Bradford M Beckmann, Steven K Reinhardt, and David A Wood. Fine-grain task aggregation and coordination on GPUs. In *ISCA-14*. ACM.
[58] Shin-Ying Lee and Carole-Jean Wu. CAWS: criticality-aware warp scheduling for GPGPU workloads. In *PACT-14*. ACM.
[59] Thomas L Falch and Anne C Elster. Register caching for stencil computations on GPUs. In *SYNASC-14*. IEEE.
[60] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. Fast multiplication in binary fields on gpus via register cache. In *SC-16*. ACM.
[61] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. Improving GPGPU concurrency with elastic kernels. In *ASPLOS-13*. ACM.