Eliminating Intra-warp Load Imbalance in Irregular Nested Patterns via Collaborative Task Engagement

Farzad Khorasani Bryan Rowe Rajiv Gupta Laxmi N. Bhuyan
Computer Science and Engineering Department
University of California Riverside, CA, USA
{*fkhor001, roweb, gupta, bhuyan*}*@cs.ucr.edu*

***Abstract*—Nested patterns are one of the most frequently occurring algorithmic themes in GPU applications where coarse-grained tasks are constituted from a number of fine-grained ones. However, efficient execution of irregular nested patterns, with coarse-grained tasks that substantially vary in size, has remained an open problem for the GPUs SIMT architecture. Existing methods rely on static task decomposition where one or a fixed number of threads inside the SIMD grouping (warp) carry out the fine-grained tasks. These approaches fail to provide portable performance across diversity of irregular inputs. Moreover, due to intra-warp load imbalance, they incur warp underutilization. In this paper, we introduce a novel software technique called Collaborative Task Engagement (CTE) that, unlike previous methods, achieves sustained high warp execution efficiencies across irregular inputs and provides portable performance. CTE assigns a group of coarse-grained tasks to the warp and allows threads inside the warp to carry out the expanded list of fine-grained tasks collaboratively. In multiple rounds, all the warp threads perform mapping portion of fine-grained tasks and participate in a reduction phase with appropriate lanes to reduce calculated values. This scheme avoids over-subscription or under-subscription of threads while preserving the benefits of parallel reduction. We prepared a CUDA C++ device-side template library for developers to easily express nested patterns in GPU kernels using our technique. Our experiments show that CTE delivers up to 37% warp execution efficiency improvement and gives up to 1.51x speedup over sub-warp decomposition with the best sub-warp width.**

***Keywords*-GPU; GPGPU; warp execution; divergence; load imbalance; irregularity; nested parallelism**

I. Introduction

The abundance of execution units, accompanied with a high memory bandwidth, have made GPUs the primary candidates for accelerating algorithms containing data parallelism. To increase the energy efficiency, GPU threads are grouped into *warps*[1] (on current Nvidia devices, 32 threads are grouped into one warp). While the instruction fetch and decode are performed once for all the threads inside the warp, threads map into different GPU execution units (cores) to process different data. Hence, the underlying design enforces the whole warp to contain only one active PC (Program Counter) at a time. However, GPU's SIMT architecture design allows threads inside the warp to take different execution paths by masking off inactive threads. This feature has made GPU programming easier since the developers need not worry about handling diverging threads executing unwanted pieces of code. However, this comfort can jeopardize the performance. One such frequent scenario is of nested patterns that contain imbalanced loads, more specifically a pattern where *a set of coarse-grained tasks hold a number of fine-grained tasks with different sizes*.

Early work for handling such nested patterns on GPUs employed 1D decomposition by assigning one thread to every coarse-grained task. The thread then iterates over the fine-grained tasks and carries them out. This approach has appeared in many GPU applications including graph processing [12], Sparse Matrix Vector Multiplication (SpMV) [3], and the analysis framework in [20] where it is known as 1D mapping. While being concise and easy to reason about, 1D decomposition is highly prone to underutilization in presence of imbalanced loads since all the threads inside the warp have to wait for the thread that has been assigned the largest number of fine-grained tasks. To cope with the warp underutilization issue in irregular nested workloads, researchers suggested assigning fixed-sized sub-warps to coarse-grained tasks [14], [34]. Therefore, threads in a sub-warp carry out its assigned fine-grained tasks iteratively. We refer to this approach as sub-warp decomposition. Although providing better warp utilization compared to 1D decomposition, sub-warp decomposition lacks portable performance since every application and input combination exhibits the best performance at a specific sub-warp width. Most importantly, the same issue that hurts 1D decomposition performance still exists in sub-warp decomposition. Here, the whole warp has to wait for the sub-warp with the largest assigned task size.

In this paper, we present Collaborative Task Engagement (CTE), a novel software technique that greatly enhances the warp utilization of irregular nested tasks compared to previous approaches. It provides a portable performance across inputs and applications. Unlike aforementioned static task-to-thread assignment methods, CTE delivers dynamic decomposition via the expansion of coarse-grained tasks. In multiple rounds, each thread inside the warp gets assigned the work of mapping portion of a fine-grained task regardless

[1]We use terms employed in CUDA platform to describe GPU specific architecture and programming model throughout the paper.

of the coarse-grained task it belongs to. Later, the thread determines the coarse-grained task to which the fine-grained task belongs. This is achieved efficiently via a binary search of the buffer containing prefix sum of task sizes – the buffer is held inside the shared memory. Therefore, it can participate in the reduction for the coarse-grained task's final value, if necessary. CTE does not over-subscribe or under-subscribe warp threads in the mapping stage and yet performs parallel reduction in minimum number of steps between the fine-grained tasks of a coarse-grained task in every round. To facilitate the employment of our technique, we have prepared a CUDA C++ device-side template library. The template library abstracts away the complications of the implementation of nested pattern with CTE, allowing developers to focus on the program's algorithm and to quickly obtain the desired functionality. In addition, the template library is built with the program hence providing ultimate portability across various systems.

The major contributions of this paper are:

- We propose *Collaborative Task Engagement* (*CTE*), a novel task decomposition technique to efficiently process irregular nested parallel patterns in GPUs. Unlike previous methods, warp threads in CTE pass over the expanded list of fine-grained tasks, making it resilient against input irregularities.
- We developed a CUDA C++ device-side template library for easy-expression of nested patterns with CTE.
- We measured and analyzed the performance of our CTE in comparison with other static decomposition methods across different class of applications. CTE improves the warp execution efficiency of CUDA kernels by up to 37% and provides 1.51x speedup compared to the sub-warp decomposition with the best sub-warp width.

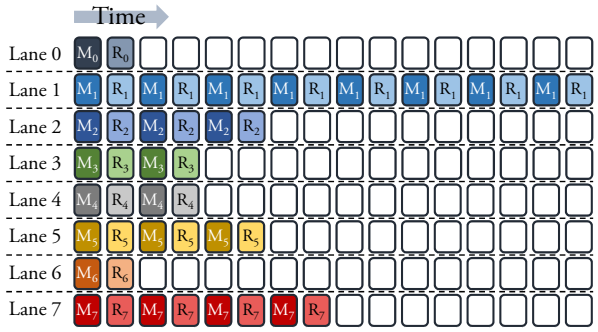
The rest of this paper is organized as follows. Section II explains the drawbacks of available static task-to-thread assignment approaches and motivates the necessity of a solution that copes with irregularities in the input. Section III presents our solution that dynamically assigns tasks to SIMD threads and is robust against load imbalance. Section IV gives experimental evaluation results. Section V discusses the related work and Section VI concludes the paper.

II. Motivation: Inefficiency of Static Task Decomposition Methods

GPU's innovative SIMT architecture enables the implementation of conditional device code in which threads belonging to a warp take different execution paths. The underlying hardware keeps track of active and inactive warp threads in diverging paths and masks off irrelevant threads. While this scheme speeds up GPU software development, it can easily make GPU kernels prone to resource underutilization. If only a few threads take a divergent path, all other threads inside the warp will have to wait for those threads before they can continue. In other words, execution

```
1  template<typename valT, typename idxT>
2  __global__ void spmv_CSR_1D_mapping( const valT* mat,
3   const idxT* nnzRowScan, const valT* inVec,
4   const idxT* colIdx, valT* outVec ) {
5     int rowID = threadIdx.x + blockIdx.x * blockDim.x;
6     valT sum = 0;
7     const idxT startPos = nnzRowScan[ rowID ];
8     const idxT endPos = nnzRowScan[ rowID + 1 ];
9     for( idxT i = startPos; i < endPos; ++i ) {
10       valT mapped =
11        mat[ i ] * inVec[ colIdx[ i ] ];  // MAP.
12       sum += mapped;  // REDUCE.
13    } outVec[ rowID ] = sum; }
```

(a) The SpMV CUDA C++ kernel with 1D decomposition.



(b) The visualization of a possible warp execution of the kernel in Figure 1(a). Warp size is assumed 8.

Figure 1. An example — Sparse Matrix-Vector Multiplication (SpMV) CUDA kernel with a CSR matrix using 1D decomposition. Intra-warp load imbalance induces warp inefficiency and performance loss.

units are reserved for inactive threads and they perform no operations.

Aforementioned issue intensifies in GPU kernels with irregular nested patterns where there are a number of coarse-grained tasks each of which contains a different number of fine-grained tasks. Therefore, threads inside the same warp may have to iterate different number of times over the code to fully carry out their task. Figure 1(a) illustrates the above problem using an example of nested pattern that appears in the SpMV kernel with CSR (scalar) [4] decomposition. Since all the threads reconverge at the end of the loop, different amounts of load for different warp threads results in partial warp utilization. In other words, threads that finish early stay inactive until the thread with the longest number of iterations finishes. In Figure 1(a), inside the loop, threads first compute the intermediate value (*map*) and then *reduce* it with the thread's private variable. Figure 1(b) visualizes the utilization of warp threads executing this loop. Note that in this work, we focus on intra-warp task assignment strategies.

This task assignment strategy, being very intuitive, is frequently seen in widely-used GPU applications involving nested parallel patterns; especially when the algorithm contains a set of coarse-grained tasks each of which containing a number of fine-grained tasks. Sparse Matrix-Vector Multiplication (SpMV) with Compressed Sparse-Row (CSR) format in [3] uses this task assignment strategy, and is algorithmically identical to the kernel in Figure 1(a). This method is

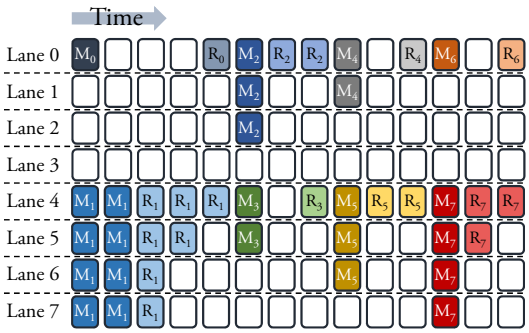


Figure 2. Warp execution visualization in sub-warp decomposition (with width 4) for the example in Figure 1. Sub-warp decomposition attempts to exploit parallelism inside coarse-grained tasks.

also employed by the analysis framework in [20] where it is called *1D mapping*. We use the term 1D decomposition throughout this paper.

To tackle the inefficiencies of 1D decomposition, CUDA provided *dynamic parallelism* to let threads spawn thread blocks for ease of expressing nested patterns; it has been shown in [34], [31] that dynamic parallelism imposes overheads such as parent thread blockage and communication via relatively slow global memory between the parent and children threads. These overheads have hindered the adoption of dynamic parallelism in GPU applications.

Other task assignment approaches aim to exploit parallelism inside a coarse-grained task— which is untouched by the 1D decomposition. The most notable among them, groups the threads belonging to the same warp and assigns the resulting coarse-grained tasks to the warps [3]. Later works improve upon this strategy by dividing the warp into smaller sub-warps and assigning each sub-warp to process a coarse-grained task. Sub-warps have fixed width – one of 2, 4, 8, 16, or 32 – throughout the kernel computation. Threads within a sub-warp participate in carrying-out the fine-grained tasks of the coarse-grained task. As an instance of usage of this scheme, CUSP library [6] assigns a sub-warp to process a row of the CSR matrix in SpMV computation. Threads inside the sub-warp execute the mapping function for a section of the row and reduce the outcomes in parallel. This procedure is performed iteratively on all the sections of the row. CUDA-NP [34] expressed a similar approach in form of a primary thread and a few subordinate threads for nested parallel patterns. Here we refer to this approach as the *sub-warp decomposition* (such approaches have also been called *warp-based mapping* [20]). Figure 2 visualizes the warp execution for the example in Figure 1 when sub-warp decomposition with width 4 is employed.

Although sub-warp decomposition provides improved SIMD utilization, it suffers from a constraint. In order to guesstimate the best sub-warp width, for every specific GPU kernel, the developer needs to know the characteristics of the algorithm and must analyze the input in a preprocessing step. This constraint of sub-warp decomposition makes application portability unachievable. A sub-warp width that works well for one input, may not deliver a good performance for another input. In addition, sub-warp decomposition suffers from the same issue as 1D decomposition. It leaves a great portion of the warp underutilized when some coarse-grained tasks contain a large number of fine-grained tasks while others have only a few. For example, CUSP's heuristic to determine the best sub-warp width is to choose the closest equal or higher power of 2 to the average of the non-zeros per row (for averages bigger than the warp size it chooses the warp size). Relying only on the average of the coarse-grained task distribution, this method basically ignores their variance. Some rows of the input matrix may have much larger number of non-zero elements than other rows but this method assigns the same processing power to each and every row. Thus, the entire warp must wait for the sub-warp with the largest amount of fine-grained tasks.

Program	Input	1D	VW2	VW4	VW8	VW16	VW32
SpMV	Wbedu	16.6%	36.8%	41.1%	49.8%	50.4%	28.9%
	Delau	57.0%	63.4%	66.5%	61.3%	52.7%	38.7%
FMM	nEquProb	20.9%	26.8%	39.7%	33.2%	37.8%	39.4%
	EquProb	42.2%	44.3%	44.7%	44.8%	36.7%	29.1%

Table I
KERNEL WARP EXECUTION EFFICIENCY OF CUDA APPLICATIONS EXPOSED TO DIFFERENT INPUTS WITH 1D AND SUB-WARP DECOMPOSITION METHODS. THE EFFICIENCY OF KERNELS NOT ONLY VARIES FROM ONE SUB-WARP WIDTH TO ANOTHER (THE BEST IN EACH ROW IS UNDERLINED), IT IS ALSO WELL BELOW 100%.

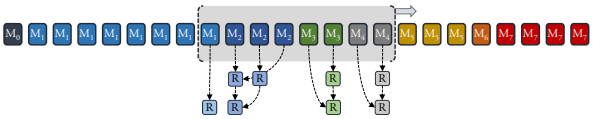
Essentially, **in both 1D and sub-warp decomposition methods, the static thread-to-task assignment not only lacks portable performance across different inputs, but also makes the kernel highly susceptible to the warp execution inefficiency due to load imbalance between irregular coarse-grained tasks**. Table I confirms this observation by showing profiled warp execution efficiency[2] of different benchmarks (FMM stands for Fast Multiple Method [19] for n-body approximation) and inputs for 1D decomposition and sub-warp decomposition with different sub-warp widths. We can see that different applications with different inputs demonstrate the best execution efficiency at various sub-warp widths.

Above observation motivates the need for an approach that, regardless of the input task size variance, effectively maps the irregular tasks to threads for efficient execution for nested patterns on the GPU architecture.

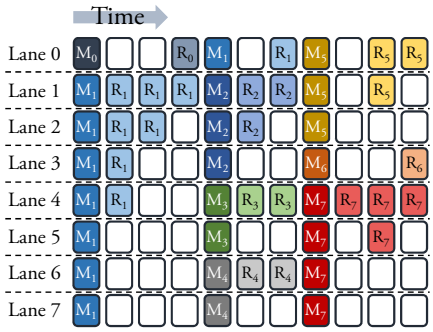
III. COLLABORATIVE TASK ENGAGEMENT

In this section, we introduce our technique, Collaborative Task Engagement (CTE), that eliminates warp inefficiencies induced by irregular nested patterns. We first describe our

[2]Warp Execution Efficiency is a metric provided by Nvidia Profiler defined as the "ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor." It is a measure indicating what fraction of threads of warps in a kernel have been active on an average. It can also be seen as a measure that is inversely proportional to the overall thread divergence.



(a) The warp acts as a sliding window executing the expanded list of fine-grained tasks in the regions with the size equal to the warp size.



(b) CTE reduces the execution time of irregular nested tasks by enhancing the warp efficiency.

Figure 3. The visualization of the SpMV CUDA kernel in Figure 1(a) after applying CTE.

solution, then explain its efficient CUDA implementation, and finally, we present our template library that allows developers to use this technique with ease.

A. Dynamic Task Assignment in CTE

To handle irregularities in nested patterns, we propose Collaborative Task Engagement (CTE). In CTE, similar to sub-warp decomposition, a fine-grained task is defined as a combination of a *mapping function* and an *associative reduction function*. While a mapping function computes a candidate value for the fine-grained task, the reduction function does a summary operation over fine-grained tasks' candidate values for the coarse-grained task. For instance, lines 11 and 12 in Figure 1(a) show the mapping function which computes the result by multiplying the matrix element with the corresponding vector element, and line 13 shows the reduction function which accumulates the resulting values to yield the output vector element.

In CTE, instead of assigning one coarse-grained task to one thread (1D decomposition) or a fixed number of threads inside the warp (sub-warp decomposition), **we assign a group of coarse-grained tasks to a warp, and let the threads in the warp collaborate to carry out the fine-grained tasks belonging to the coarse-grained tasks assigned to the warp**. More specifically, threads of the warp view and iterate over the list of fine-grained tasks resulting from the expansion of coarse-grained ones. Figure 3 demonstrates this scheme for the example in Figure 1.

Unlike previous static task decomposition methods, **Each thread inside the warp is assigned to execute one compute function corresponding to a fine-grained task; regardless of the coarse-grained task from which the fine-grained task comes from**. By unbundling fine-grained tasks from

Input: thread's initial coarse-grained task (`ThreadCoarseTask`).
Output: thread's initial coarse-grained task final outcome.

```
1  volatile shared scans[ N_CTA_WARPS ][ WARP_SIZE ];
2  volatile shared reds[ N_CTA_WARPS ][ WARP_SIZE ];
3  volatile shared mapped[ N_CTA_WARPS ][ WARP_SIZE ];
4  volatile shared taskDesc[ N_CTA_WARPS ][ WARP_SIZE ];
5  scans[ warp_id ][ lane_id ] =
    prefix_sum( ThreadCoarseTask.size );
6  NFineTasks = scans[ warp_id ][ WARP_SIZE - 1 ];
7  firstLoad = scans[ warp_id ][ 0 ];
8  reds[ warp_id ][ lane_id ] =
    ThreadCoarseTask.initRedVal;
9  taskDesc[ warp_id ][ lane_id ] =
    ThreadCoarseTask.fineTaskDescriptor;
10 for( fineTaskID = lane_id;
       fineTaskID < NFineTasks;
       fineTaskID += WARP_SIZE ) {
11   coarseTaskID =
      binary_search( fineTaskID, scans[ warp_id ] );
12   fineTask = task_descriptor( fineTaskID,
      taskDesc[ warp_id ], coarseTaskID );
13   mapped[ warp_id ][ lane_id ] = map( fineTask );
14   inSegIdx = min( lane_id, fineTaskID -
      scans[ warp_id ][ coarseTaskID ] + firstLoad );
15   segSize = min( scans[ warp_id ][ coarseTaskID ]
      - fineTaskID, WARP_SIZE - lane_id ) + inSegIdx;
16   redElemPos = ( inSegIdx != 0 ) ?
      ( mapped[ warp_id ] + lane_id - 1 ) :
      ( reds[ warp_id ] + coarseTaskID );
17   for( i = WARP_SIZE / 2; i > 0; i /= 2 )
18     if( ( inSegIdx + i ) <= segSize )
19       *redElemPos = reduce( *redElemPos,
          mapped[ warp_id ][ lane_id + i - 1 ] );
20 }
21 return reds[ warp_id ][ lane_id ];
```

Figure 4. GPU pseudo-code for CTE.

their coarse-grained task, the execution of compute stage in CTE completely avoids the warp inefficiency. After the compute stage, since fine-grained tasks from a coarse-grained task are processed by consecutive threads in the warp, a thread can find its corresponding coarse-grained task and execute the reduction function over the results with its neighbors, if necessary. The parallel reduction is performed over the results of the computation for a coarse-grained task to produce its final result. Figures 3(a) and 3(b) exhibit this procedure. As can be seen, warp inefficiencies due to load imbalance in CTE can only appear during the reduction phases, however, their effect will not last longer than at most $\log warpSize$ reduction steps. The advantages of CTE include:

- It avoids under-subscribing or over-subscribing warp threads during the mapping by assigning a map function to every thread in each round regardless of their corresponding coarse-grained task; and
- It reduces the effect of load imbalance between coarse-grained tasks by performing parallel reduction over the fine-grained tasks belonging to a coarse-grained one.

B. Efficient CUDA Implementation of CTE

Next, we describe the details of CTE's CUDA implementation using the pseudo-code presented in Figure 4. We provide a step by step description of the pseudo-code.

Shared memory declaration and allocation – CTE requires shared memory buffers to exchange data between threads of a warp. To enforce sequential consistency between the shared memory accesses within a warp, `volatile` qualifier accompanies shared memory declarations (lines 1-4). Using this technique— and since in CTE the set of interactions between threads is confined to within their own warp— our procedure avoids introducing any explicit syncing or fencing primitives.

Coarse-grained task feature extraction – Initially, each thread corresponds to one coarse-grained task. This coarse-grained task is the input to the procedure in Figure 4. First, at line 5, we compute the *inclusive prefix sum* of coarse-grained task sizes that threads of the warp hold and save them into the `scan` buffer. This buffer is necessary for multiple uses in the iterative code section (lines 10-20). For a fast intra-warp prefix sum, we employed the method introduced in [32] that utilizes the *shuffle intrinsic*. After calculating prefix sums, the last element gives the total number of fine-grained tasks (line 6). Plus, we put the first element into a variable (line 7) so we can use it inside the iterative segment to get the *exclusive prefix sum* results. At line 8, each thread inserts the initial value (given by the user-specified algorithm) for the reductions of fine-grained tasks over its initially assigned coarse-grained task. The `reds` buffer collects the reduction results in the iterative section and eventually to be output by the program (line 21). Moreover, `taskDesc` specifies the set of shared memory buffers that collect the fine-grained task descriptors for coarse-grained tasks.

Each thread, which is initially assigned to a coarse-grained task, has a number of variables that are used by the fine-grained tasks inside the coarse-grained task and vary among the coarse-grained tasks. For example, in Figure 1(a) `startPos` and `endPos` are thread-private variables that directly affect the execution of fine-grained tasks inside the loop. We call such variables *task descriptor*. Since in CTE fine-grained tasks of one coarse-grained task are executed by multiple threads, the thread saves its task descriptor variables inside the shared memory so as to make them accessible by all the threads inside the warp. Note that this is necessary for all decomposition methods that exploit parallelism within coarse-grained tasks (such as sub-warp decomposition).

Fine-grained task assignment and mapping – Lines 11-20 present the iterative segment in which every thread inside the warp is assigned to one fine-grained task identified by `fineTaskID`. First, the coarse-grained task owning the thread's assigned fine-grained task is found via a binary search on the `scans` buffer inside the shared memory at line 11. Then, at line 12, thread's assigned fine-grained task is retrieved from the `taskDesc` buffer using thread's fine-grained task index and its corresponding coarse-grained task index. The thread executes the mapping portion of the described fine-grained task in line 13 and saves the result inside the designated shared memory buffer position.

Parallel reduction of mapped fine-grained tasks – At this point, threads inside the warp performed mapping on fine-grained task, and now, need to properly reduce mapped values. Since fine-grained tasks belonging to one coarse-grained task were assigned to consecutive threads, they form segments when they are processed using the `for` loop specified in line 10. If the thread discovers its index inside the segment and also the segment size, parallel reduction inside the segment will become feasible. Thus, line 14 calculates the intra-segment index using `scans` buffer and line 15 computes the segment size by adding the intra-segment index with the fine-grained task index inside the segment when observed from right to left. In line 16, we assign the first thread inside the segment to the segment's reduction element inside `reds` buffer and assign the rest of the threads to the mapped value of the thread before them inside the segment. This re-assignment becomes beneficial by eliminating the need for an additional reduction with the corresponding element inside the `reds` buffer at every iteration. Finally, an intra-segment parallel reduction (with unrolled loop in the actual implementation) reduces the mapped values and saves the outcome inside the corresponding `reds` buffer position. Threads keep executing the code section in lines 10-20 until all the fine-grained tasks are carried out. Finally each thread returns the reduced value for its initially-assigned coarse-grained task (line 21).

C. CTE as A Device-side Template Library

While CTE provides an efficient method to handle irregular nested patterns, its implementation from scratch for every GPU kernel can be time-consuming and challenging for CUDA programmers. To enable easy usage of CTE by developers, we provide our technique as a CUDA C++ device-side template library. A CUDA developer only needs to include our library header file and call the designated library function. The library function takes as its parameters the thread's fine-grained task index range, mapping and reduction functions, and initial content for thread's coarse-grained task's reduction value. While the programmer expresses the tasks as if each thread is assigned to one coarse-grained task (similar to 1D decomposition), the library manages the CTE execution behind the scenes.

Figure 5 shows the usage of our library for the SpMV kernel. In this example, the mapping function is defined as a lambda (lines 10 and 11) that takes the iteration index as the parameter and returns the corresponding element, i.e. the multiplication outcome. The signature of the mapping function for our library requires the first parameter to be the iteration index while the rest of the parameters can be passed by the user as the *lane state*. Lines 12 and 13 in Figure 5 present the reduction function— again as a lambda expression— for this example. The reduction function signature for the library accepts only two parameters and returns one value of the same type. Finally, lines 14 and 15 give the

```
1  #include <cte.cuh>  // CTE library inclusion.
2  template<uint BlockDim, typename valT, typename idxT>
3  __global__ void spmv_CSR_with_CTE( const valT* mat,
4   const idxT* nnzRowScan, const valT* inVec,
5   const idxT* colIdx, valT* outVec ) {
6     int rowID = threadIdx.x + blockIdx.x * blockDim.x;
7     valT sum = 0;
8     const idxT startPos = nnzRowScan[ rowID ];
9     const idxT endPos = nnzRowScan[ rowID + 1 ];
10    auto mapF = [&]( idxT idx ) {  // MAP.
11     return mat[ idx ] * inVec[ colIdx[ idx ] ]; };
12    auto redF = []( valT lhs, valT rhs ) {  // REDUCE.
13     return lhs + rhs; };
14    sum = cte::for_each_index<BlockDim, cte::scanned>
15     ( startPos, endPos, mapF, redF, sum );
16    outVec[ rowID ] = sum; }
```

Figure 5. Expressing the nested pattern in Fig. 1(a) CUDA C++ kernel in CTE form using our template library interface.

function call to execute the tasks with CTE technique.

CTA (thread-block) dimension needs to be sent to the function as the first template argument so the library would have the correct size for the static shared memory allocation. Also, the second template argument hints the library that the indices of consecutive threads are prefix summed. In other words, the ending index for thread i's region is the beginning index for thread $(i+1)$'s. This template specialization will allow the library to avoid recalculation of the prefix sum of the fine-grained task sizes. If such relationship between indices does not exist, the user will have to pass `cte::disjoint` as the template argument. We mentioned earlier that in this example `startPos` and `endPos` act as task descriptors and need to be passed as function arguments. We specialized the CTE function calls with more template signatures so that for an arbitrary mapping function, other task descriptor variables can be passed as the last variables of the CTE function call in the order they appear as the further mapping function parameters.

Finally, since all the threads of the warp need to be present for a correct CTE execution, upon entering the execution function, the library performs a `__ballot()` operation with `true` predicate. If the result of this operation is not a variable with all bits set, it means one or a number of warp threads are absent. In this case, as a safety procedure the library falls back to the 1D decomposition method.

D. CTE Analysis for Comparison with Static Decomposition Methods

To analyze the CTE characteristics and compare it with static decomposition methods, we briefly provide analysis over the execution time for static decomposition methods and CTE. We show that while the execution time for 1D decomposition and the upper-bound for sub-warp decomposition execution times are a function of the *maximum*(s) of the set of coarse-grained task sizes, the upper-bound for the execution time of CTE is a function of the *average* of the workload.

Assumptions and notation. For analysis, let us assume that the warp size is W, and for simplicity, further assume that there are W coarse-grained tasks to be processed – the time for bigger coarse-grained tasks can be obtained via scaling. Let us denote the execution time of the *mapping* and *reduction* functions by T_{MAP} and T_{RED} respectively. Note that for simplicity of analysis we assume that these times are constant and not affected by memory access latencies.

1D decomposition – Given a set of coarse-grained tasks $L = \{l_1, l_2, ..., l_W\}$, the execution time of this set of loads with 1D decomposition is given by:

$$t_{1D}(L) = \max L \times (T_{MAP} + T_{RED}) \tag{1}$$

Equation 1 above can be easily understood by examining Figure 1(b). Note that the above equation shows that t_{1D} is a function of the *maximum* of the set of loads.

Sub-warp decomposition – For sub-warp decomposition, if the width of the chosen *sub-warp* is S (note $\log_S W \in \mathbb{N}$), then the upper-bound for its execution time is given by:

$$\begin{aligned} t_{SW}(L,S) &= \sum_{s=1}^{S}(T_{RED} \times \log_2 S + \\ &\quad (T_{MAP} \times \lceil \frac{\max\{l_i | \frac{(s-1)\times W}{S} < i \leq \frac{s\times W}{S}\}}{S} \rceil)) \\ &= T_{RED} \times S \log_2 S + \\ &\quad T_{MAP} \times \sum_{s=1}^{S} \lceil \frac{\max\{l_i | \frac{(s-1)\times W}{S} < i \leq \frac{s\times W}{S}\}}{S} \rceil \end{aligned} \tag{2}$$

Initial form of Equation 2 sums up the execution times in different rounds since sub-warp decomposition assigns V threads to process a coarse-grained task. While $\log_2 V$ is the maximum number of steps required for the reduction, the mapping operation is repeated within a round by the warp as long as the largest coarse-grained task's mapping functions assigned to a sub-warp are being performed. The final form of Equation 2 shows that t_{VW} is still a function of the largest tasks. Also, V appears at both top and the bottom of the fractions of Equation 2 which usually makes t_{VW} a non-monotonic function of V. The V for which t_{VW} is minimum depends on the load distribution.

CTE – On the other hand, the upper-bound of the execution time for CTE is expressed as below:

$$\begin{aligned} t_{CTE}(L) &= \lceil \frac{\sum_{i=1}^{W} l_i}{W} \rceil \times (T_{MAP} + T_{RED} \times \log_2 W) \\ &= \lceil Avg(L) \rceil \times (T_{MAP} + T_{RED} \times \log_2 W) \end{aligned} \tag{3}$$

In CTE, in every round the fine-grained compute portion of the tasks are assigned to the warp threads and therefore the sum of loads divided by the warp size is the coefficient of both T_{MAP} and T_{RED} in Equation 3. Also, the reduction

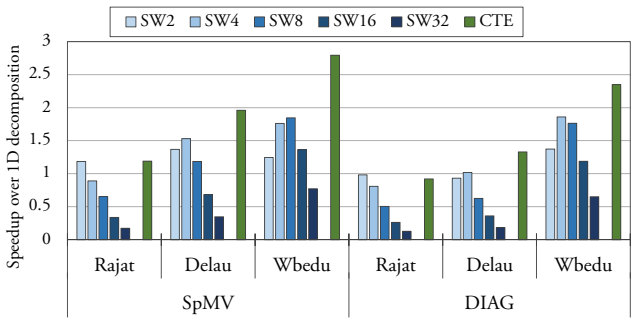


Figure 6. The kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for matrix operations on real-world matrices.

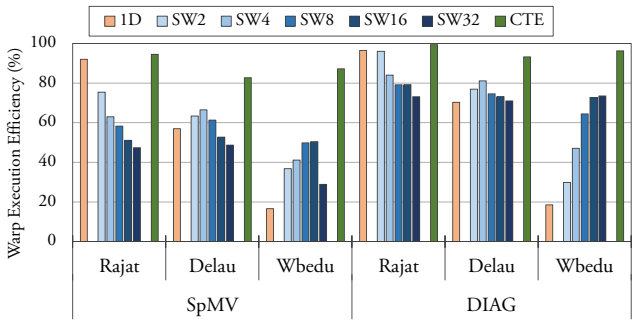


Figure 7. Profiled warp execution efficiency of CTE, sub-warp decomposition, and 1D decomposition kernels for experiments in Figure 6.

will take $\log_2 W$ steps at most, therefore, this term accompanies T_{RED}. Considering the final form of Equation 3, we can see that the upper-bound for t_{CTE} is a function of the *average* of the loads, not their *maximum* unlike the previous two methods, i.e. 1D and sub-warp decomposition.

IV. Experimental Evaluation

Next, we evaluate the performance of CTE and compare it with 1D and sub-warp decomposition methods. We use applications from various domains including sparse matrix operations, scientific computing, and graph analytics. We performed the experiments on a Nvidia GeForce GTX 780 with 12 Streaming Multiprocessors from the Kepler family. We compiled and ran all the programs for CUDA Compute Capability 3.5 with -O3 and C++11 compilation flags on a system with Ubuntu 14.04 and CUDA 7.0.

A. Performance Analysis

Sparse matrix operations – Figure 6 presents the speedup of CTE over 1D decomposition and compares it with the speedup provided by sub-warp decomposition from CUSP library [6] for two application from sparse matrix operation domain. SpMV is the Sparse Matrix Vector Multiplication and DIAG is the extraction of the diagonal of the given matrix. Input graphs are from The University of Florida sparse matrix collection [7] and exhibit different structures and therefore nested load size variation. Rajat31 (Rajat) is an unsymmetric and rather regular matrix with a dimension of 4.69M and approximately 20M non-zero elements. Delauny_n24 (Delau) is a symmetric irregular matrix with 16.7M rows and columns and 100M non-zero elements. Wb-edu (Wbedu) is a more irregular unsymmetric matrix compared to Delau with 9.8M rows and columns and 57M non-zero elements. Also, to further verify the CTE performance compared to static decomposition methods, we profiled the warp execution efficiency of CTE, sub-warp and 1D decomposition kernels with Nvidia Profiler and plotted the results in Figure 7.

Starting with Wbedu in Figure 6 as the most irregular input, CTE provides 2.8x and 2.3x speedup compared to 1D decomposition for SpMV and DIAG respectively. SpMV is a more compute-intensive application compared to DIAG and can benefit more from our technique. CTE speedup becomes more compelling in the light of sub-warp decomposition speedup over 1D decomposition which can be less than 1 (for sub-warp width 32) and maximize at 1.8x. The CTE supremacy is explained via Figure 7 in which CTE shows 87% and 96% warp execution efficiency for SpMV and DIAG respectively while 1D decomposition warp efficiency does not exceed 20% and sub-warp decomposition warp efficiency for different sub-warp widths varies greatly from SpMV to DIAG. As we move toward more regular input matrices (Delau and Rajat), the variation in the size of coarse-grained loads reduces hence 1D and sub-warp decomposition exhibit a better warp utilization and perform better. For Delau, CTE provides 1.9x and 1.3x speedup over 1D decomposition by enhancing the warp efficiency 25% and 23% for SpMV and DIAG respectively. It also provides 1.28x and 1.3x speedup over the sub-warp decomposition method with the best sub-warp width. Finally, for Rajat, since the graph is very regular, CTE shows approximately the same speedup as the best sub-warp decomposition width (1.18x) for SpMV.

Scientific applications – In this section, we measured the performance of two scientific applications, Fast Multiple Method [19] (FMM) and Dynamical Quadrature Grids [22] (DQG) when performed using CTE and 1D and sub-warp decomposition methods. The results are depicted in Figure 8. For FMM, which is an n-body approximation that groups the particles in a quad-tree, we consider 10M points in 3D space as the input, and vary the maximum density (Q) of points in each leaf between 5 and 10. We calculate the U-list phase of FMM procedure and distribute the points with non-equal probability distribution (nEquProb) and equal probability distribution (EquProb). It is evident that for two irregular inputs, our technique outperforms both 1D decomposition and sub-warp decomposition by up to 1.5x and 1.15x-1.67x. However, for the regular input sub-warp decomposition with sub-warp width 16 shows slightly better performance. Moreover, DQG computes the points inside a quadrature grid. For DQG, we vary the maximum number of atoms per molecule between 20 and 40 and

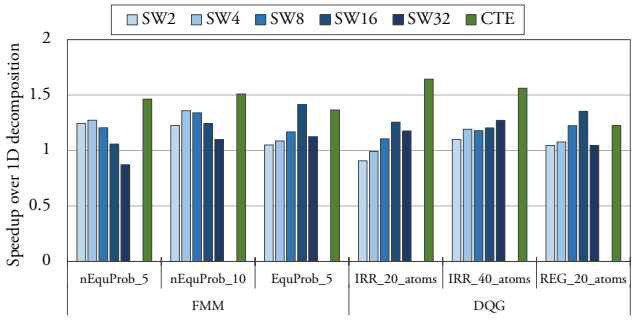


Figure 8. The kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for Fast Multiple Method [19] and Dynamical Quadrature Grids [22] with different inputs.

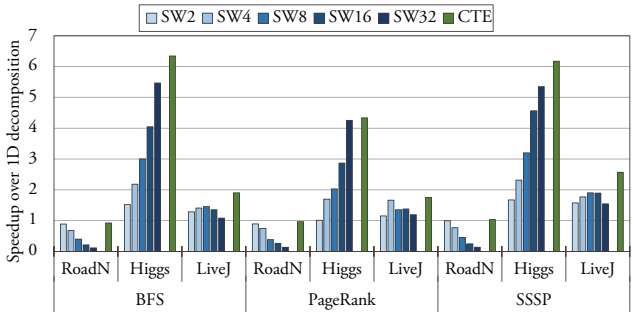


Figure 9. The kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for different graph applications and inputs.

provide regular and irregular input sets. For the regular input, the number of atoms in molecules are randomly selected between 1 and maximum allowed. For irregular inputs, they are selected using normal distribution. Similar to FMM, irregularity in inputs manifests the CTE supremacy while for a regular input CTE performs on-par with the best sub-warp decomposition width. Also, since compared to FMM, DQG kernel has a more compute-intensive map portion, resulting speedups are slightly higher, covering CTE overhead of re-bundling fine-grained tasks with their corresponding coarse-grained ones.

Graph Analytics – Figure 9 shows the kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for 3 graph applications (BFS, SSSP, and PageRank [26]) over 3 real-world graphs. These graphs have different number of nodes and edges and exhibit various degree distribution patterns. A coarse-grained task in this case processes a node which includes visiting its neighbors as the fine-grained tasks. For this section, we used the sub-warp decomposition implementation in [18] and hand-wrote 1D decomposition. First, LiveJournal [1] (LiveJ) has around 4.85M nodes and 69.0M edges and has a power-law degree distribution. CTE shows better performance for this graph in all application by being 1.30x, 1.08x, and 1.34x better than the best sub-warp decomposition option for BFS, PageRank, and SSSP respectively. Also note that the best sub-warp for different applications differ; this signifies the need for try-and-error or profiling in sub-warp decomposition for every algorithm and input combination. Second, HiggsTwitter [8] (Higgs) is even more irregular compared to LiveJournal and contains 0.46M nodes and 14.8M edges. The results for this graph demonstrate the ineffectiveness of 1D decomposition confronting heavy amount of load imbalance in nested patterns. Finally, RoadNetCA [21] (RoadN) with 1.96M nodes and 5.53M edges is an example of a regular input for the benchmarks due to its internal connectivity. Most of the nodes in this graph have approximately 1 to 4 neighbors. Therefore, for this graph, 1D decomposition usually performs the best since coarse-grained tasks have roughly equal amounts of fine-grained loads. However, even

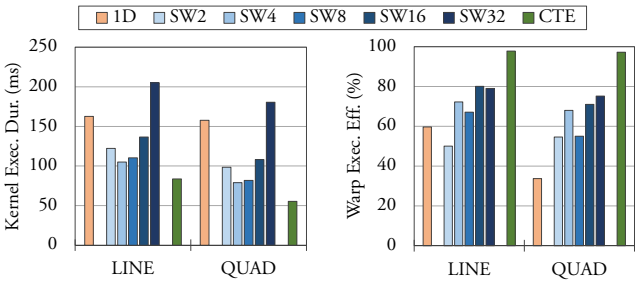


Figure 10. Kernel execution duration (left plot) and Warp execution efficiency (right plot) for decomposition methods when the task sizes vary linearly and quadratically proportional to the lane index. Map and reduce portion of the fine-grained tasks each contain 20 FMAD instructions. For the LINE scenario, the coarse-grained task size is calculated with $4 \times laneID$ while for the QUAD scenario it is calculated with $\frac{laneID^2}{8}$. Task sizes for the sub-warp decomposition are calculated using their sub-warp index.

for this regular pattern, CTE exhibits performance in-par with 1D decomposition by 0.92x, 0.96x, and 1.03x speedup for BFS, PageRank, and SSSP respectively. Also, it is clear that sub-warp decomposition performance becomes worse as we increase the sub-warp size due to over-subscription.

B. Sensitivity Analysis: varying coarse-grained task sizes

In this section, we analyze the performance of 1D, sub-warp decomposition, and CTE with two synthetic compute-intensive scenarios. The first scenario assigns each thread inside the warp a task size linearly proportional to its $laneID$. Whereas, in the second scenario, the task sizes are proportional to $laneID^2$. These two scenarios are distinguished in Figure 10 with *LINE* and *QUAD* respectively. Note that in sub-warp decomposition threads calculate their task size using their sub-warp ID so that kernels for all decomposition methods get the same overall task sizes for a fair comparison. The coefficients for these two scenarios are selected so that they give approximately the same overall kernel execution duration for the 1D decomposition.

While 1D decomposition kernel takes the same amount of time for both LINE and QUAD scenarios to finish, as it is shown in Figure 10, average number of fine-grained tasks per a coarse-grained task for LINE and QUAD are approximately 60.8 and 40.5. This confirms our previous

statement about the kernel duration being a function of the maximum of coarse-grained load sizes in 1D decomposition. Also, by making the loads more irregular (LINE vs QUAD), warp execution efficiency for 1D decomposition kernel halves, demonstrating its vulnerability to the intra-warp load imbalance. For sub-warp decomposition, although the kernel duration for different sub-warp sizes reduces by moving from LINE to QUAD by 12 to 24 percent, it does not reflect 33% reduction in the load size. On the other hand, CTE kernel duration drops by 33% confirming the CTE performance dependency to the average of the loads. Plus, unlike sub-warp decomposition that exhibits varying warp execution efficiency for different sub-warp widths in both scenarios, CTE exhibits excellent warp efficiency.

V. Related Work

In addition to the methods and works mentioned in Section II, queue-based approaches are also implemented to handle irregularities of task loads. [30] gives a dynamic decomposition scheme based on task stealing and donation using queues. However, the implementation of such queues involves heavy contention over the atomic variable and also over global locks that have to be passed around spawned CTAs. Design that works around a central lock, such as [23] for BFS graph traversal, imposes inefficiencies due to latencies. Early works of graph computation on CUDA platform [12] employ 1D decomposition and assign every coarse-grained task (e.g., processing of a graph node) to one GPU thread. The thread then iterates over fine-grained tasks (e.g., visiting the node's neighbors). Later, [14] presented sub-warp decomposition for graph algorithms and named each sub-warp a virtual warp. Similarly, virtual warps have a fixed power-of-2 size throughout the kernel computation. While only one thread inside the virtual warp performs the SISD (Single Instruction Single Data) phase of the kernel, all the threads inside the virtual warp participate in the SIMD phases. [18] generalized this solution for vertex-centric graph algorithms using intra-virtual-warp reductions. Merril et.al. [24] realized the significance of load balancing with parallel scan [25] in sparse graph processing. However, their solution is limited to BFS graph traversal and does not consider the interaction between fine-grained tasks of a coarse-grained task. Finally, Warp Segmentation [16] introduces a SIMD efficient reduction mechanism between neighbors of a vertex. In comparison, CTE breaks the associativity between fine-grained and coarse-grained tasks, formulates the idea of expansion of fine-grained tasks, and generalizes the solution by introducing an efficiently implemented template library.

Xiang et. al. examined the effect of inter-warp load imbalance in [33], in which they referred to it as *warp-level divergence*. This solution compliments intra-warp decomposition methods (1D, sub-warp, CTE); even though inter-warp load imbalance effect is insignificant especially for GPU kernels with high occupancy.

Similar to our library, Thrust [13] is a CUDA C++ template library that provides interfaces such as `thrust::for_each` for the expression of iterative code segments. However, underlying scheme to carry out the fine-grained tasks in nested patterns is 1D decomposition.

Load imbalance, as a problem for SIMT architecture, is generally a variation of thread divergence; thus, offered solutions for divergence are of importance for irregular nested parallel patterns. Collaborative Lanes [15] is a method to overcome intra-warp underutilization during batched insertions in GPU Hashing. To mitigate thread divergence, [9] schedules the path in the program that most threads take using `__all()` and `__any()` CUDA primitives; however, it fails to provide full warp utilization. Other solutions that rely on majority voting, [10], [11], [29], attempt to eliminate thread divergence similarly by enforcing all or none of the threads to take the divergent path. While [10], [11] require information from the program to schedule the execution of divergent path, [29] approximates the final outcome and accepts errors in the output. Moreover, CCC [17] implements an efficient all-or-none discipline for repetitive divergent tasks. Warp specialization [2] is another method to overcome thread divergence but only when there are tasks for threads within a warp that are of differing nature. Furthermore, data remapping techniques [35], [36] may reduce divergence at the expense of static analysis or disrupting GPU kernel autonomy. [28], [5], [27] aim to mitigate the effect of divergence by profiling the GPU application. Profile-guided approaches are orthogonal to our technique and can be applied contemporaneously.

VI. Conclusion

In this paper, we introduced a novel software technique named Collaborative Task Engagement (CTE) for efficient expression and execution of GPU kernels containing nested patterns. Unlike existing solutions where static assignment of threads to tasks does not provide portable application performance across multiple inputs and induces warp underutilization, CTE assigns threads inside the warp to process a group of tasks collaboratively. Consecutive threads process the consecutive fine-grained tasks resulted from the expansion of coarse-grained tasks, determine the coarse-grained task they belong to, and participate in parallel reduction with their neighbors. We prepared a CUDA C++ device-side template library to facilitate the employment of our technique. We showed that CTE is resilient against irregularities and provides up to 1.51x speedup over the best sub-warp decomposition width and exhibits excellent warp execution efficiency.

Acknowledgment

This work is supported by NSF Grants CCF-0905509, CNS-1157377, CCF-1318103, CCF-1524852, and CCF-1423108 to UC Riverside.

REFERENCES

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *SIGKDD Int. Conf. on Knowledge Discovery and Data Mining* (KDD), pp. 44–54, 2006.

[2] M. Bauer, S. Treichler, and A. Aiken, "Singe: Leveraging warp specialization for high performance on gpus," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), pp. 119–130, 2014.

[3] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Conference on High Performance Computing Networking, Storage and Analysis* (SC), pp. 18:1–18:11, 2009.

[4] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[5] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira, "Profiling divergences in gpu applications," *Concurrency and Computation: Practice and Experience*, 25(6):775–789, 2013.

[6] S. Dalton, N. Bell, L. Olson, and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2014, version 0.5.0. [Online]. Available: http://cusplibrary.github.io/

[7] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[8] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi, "The anatomy of a scientific rumor," *Sci. Rep.*, vol. 3, 2013.

[9] S. Frey, G. Reina, and T. Ertl, "Simt microscheduling: Reducing thread stalling in divergent iterative algorithms," in *International Conference on Parallel, Distributed and Network-Based Processing* (PDP), pp. 399–406, 2012.

[10] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *Workshop on General Purpose Processing on Graphics Processing Units* (GPGPU-4), pp. 3:1–3:8, 2011.

[11] ——, "Reducing divergence in gpgpu programs with loop merging," in *Workshop on General Purpose Processor Using Graphics Processing Units* (GPGPU-6), pp. 12–23, 2013.

[12] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *International Conference on High Performance Computing* (HiPC), pp. 197–208, 2007.

[13] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2015, version 1.8.1. [Online]. Available: http://thrust.github.io

[14] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *ACM Symposium on Principles and Practice of Parallel Programming* (PPoPP), pp. 267–276, 2011.

[15] F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan, "Stadium Hashing: Scalable and Flexible Hashing on GPUs," in *International Conference on Parallel Architectures and Compilation Techniques* (PACT), pp. 63–74, 2015.

[16] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *International Conf. on Parallel Architectures and Compilation* (PACT), 2015.

[17] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection," in *ACM International Symposium on Microarchitecture* (MICRO-48), pp. 204–215, 2015.

[18] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: Vertex-centric graph processing on gpus," in *International Symposium on High-performance Parallel and Distributed Computing* (HPDC), pp. 239–252, 2014.

[19] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W.-m. Hwu, *Performance Analysis and Tuning for General Purpose Graphics Processing Units*, Morgan & Claypool Publishers, 2012.

[20] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun, "Locality-aware mapping of nested parallel patterns on gpus," in *IEEE/ACM International Symposium on Microarchitecture* (MICRO-47), pp. 63–74, 2014.

[21] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[22] N. Luehr, I. Ufimtsev, and T. Martinez, "Chapter 3 - dynamical quadrature grids: Applications in density functional calculations," in {*GPU*} *Computing Gems Emerald Edition*, ser. Applications of GPU Computing Series, W.-m. W. Hwu, Ed. pp. 35–42, 2011.

[23] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Design Automation Conference* (DAC), pp. 52–55, 2010.

[24] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *SIGPLAN Symp. on Principles and Practice of Parallel Programming* (PPoPP), pp. 117–128, 2012.

[25] D. Merrill and A. Grimshaw, "Parallel scan for stream architectures," *University of Virginia, Department of Computer Science, Charlottesville, VA, Tech. Rep. CS2009-14*, 2009.

[26] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[27] D. Sampaio, R. M. d. Souza, S. Collange, and F. M. Q. a. Pereira, "Divergence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 4, pp. 13:1–13:36, Jan. 2014.

[28] S. Sarkar and S. Mitra, "A profile guided approach to optimize branch divergence while transforming applications for gpus," in *India Software Engineering Conf.*, pp. 176–185, 2015.

[29] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," in *International Conference on Parallel Architectures and Compilation Techniques* (PACT), pp. 427–428, 2012.

[30] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the gpu," in *Conference on High Performance Graphics* (HPG), pp. 29–37, 2010.

[31] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *IEEE International Symposium on Workload Characterization*, October 2014.

[32] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013, pp. 410–411.

[33] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in gpus: Characterization, impact, and mitigation," in *IEEE International Symposium on High Performance Computer Architecture* (HPCA), pp. 284–295, 2014.

[34] Y. Yang and H. Zhou, "Cuda-np: Realizing nested thread-level parallelism in gpgpu applications," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), pp. 93–106, 2014.

[35] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, "Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data remapping," in *ACM International Conference on Supercomputing* (ICS), pp. 115–126, 2010.

[36] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," in *International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS XVI), pp. 369–380, 2011.