Locality-aware Task-parallel Execution on GPUs

Jad Hbeika and Milind Kulkarni

Purdue University

Abstract. GPGPUs deliver high speedup for regular applications while remaining energy efficient. In recent years, there has been much focus on tuning irregular, task-parallel applications and/or the GPU architecture in order to achieve similar benefits for irregular applications running on GPUs. While most of the previous works have focused on minimizing the effect of control and memory divergence, which are prominent in irregular applications and which degrade the performance, there has been less attention paid to decreasing cache pressure and hence improving performance of applications given the small cache sizes on GPUs.
In this paper we tackle two problems. First we extract data parallelism from irregular task parallel applications, which we do by subdividing each task into sub tasks at the CPU side and sending these sub tasks to the GPU for execution. By doing so we take advantage of the massive parallelism provided by the GPU. Second, to mitigate the memory demands of many tasks that access irregular data structures, we schedule these subtasks in a way to minimize the memory footprint of each warp running on the GPU. We use our framework with 3 task-parallel algorithms and show that we can achieve significant speedups over optimized GPU code.

1 Introduction

GPGPUs have proven themselves to be a cost-effective way of accelerating applications. The single-instruction, multiple-thread (SIMT) execution model of GPUGPUs provides massive amounts of parallelism while remaining energy efficient. The hardware of the GPU is limited to keep power consumption low. Most prominently, the SIMT execution model requires that all threads in a *warp* perform the same instruction at the same time to enjoy parallelism; if different threads do different work, some threads sit idle and parallelism is lost. Second, there is relatively little hardware support for hiding latency—the core cannot execute instructions out of order, nor are there forwarding networks to help mitigate the penalty of long-latency instructions—instead, the GPU relies on massive multithreading to hide latency, keeping hundreds or even thousands of threads in context to swap in and out during long-latency operations. Perniciously, this means that not only does the GPU *support* massive parallelism, it *needs* massive parallelism for effective execution. Finally, caches are small compared to their CPU counterparts, especially when considered on a per-thread basis (thousands of threads spread a <1 MB cache very thin!)

As a result of these limitations, not all applications can execute efficiently on GPUs. GPUs are well-suited to *regular*, *data-parallel* applications, where the well-structured computation is performed on different pieces of data. In these applications, the similarity of the computations performed in parallel means that there are not substantial penalties for executing in a SIMT manner. Moreover, the data-parallel nature of the application means that it is easy to generate enough parallelism to fill the GPU, allowing for effective multithreading.

However, for *task* parallel applications, where parallelism arises from independent, distinct tasks that run simultaneously, GPUs are not nearly as attractive a target. First, the fork-join nature of task parallelism does not map well to standard GPU programming models such as CUDA [11] or OpenCL [9]. Second, even if there is data parallelism in these applications (either because individual tasks have data parallel work, or because there is some parallel outer loop in the application), there may not be enough parallelism to effectively use the GPU's resources. Third, even *if* the tasks of the program could be mapped to the GPU, the limited memory subsystem of the GPU can lead to poor performance in data-heavy tasks.

There has been recent work on turning task parallelism into data parallelism to map task-parallel applications to hardware (including GPUs) that is made for data parallelism [5, 12, 13]. These proposals either require hardware changes [5, 12] or target fine-grained data parallelism in SIMD units [13]. None of these approaches consider locality.

In this paper, we propose a *locality-aware*, *task-queue* abstraction for mapping task-parallel applications to GPUs. The basic approach is to expand task parallel work *on the CPU* to generate a large number of tasks. These tasks are then inserted into one or more task queues according to the type of computation they perform and, crucially, the locality properties of the tasks. These queues are then merged into a single queue that is sent to the GPU, where they are executed in a data-parallel manner, with each task executing to completion on the GPU. This model has several features. First, by expanding out the task-parallel work on the CPU, we avoid needing to handle task-parallelism on the GPU; instead, once execution begins on the GPU, it is purely data parallel. Second, because the task queues are partitioned based on operation type, the tasks that execute simultaneously are computationally similar, promoting efficient SIMT execution. Finally, the locality-aware nature of the queues promotes tasks in the same queue having overlapping memory footprints, reducing cache pressure and hence improving performance relative to a locality-unaware approach.

We evaluate this queue abstraction on three applications: a task-parallel implementation of the fast multipole method, and two data mining applications that feature a mix of data-parallelism and task-parallelism, nearest-neighbor and two-point correlation. In all three cases, we demonstrate that our locality-aware approach delivers better performance than a locality-agnostic one. For the mixed applications, not only do we show that our locality-aware approach is better than the locality-agnostic approach, but we also show that in the absence of very large amounts of data parallelism (for example, only 200,000 data-parallel

iterations), our task-parallel approach, by exploiting additional parallelism, is also significantly faster than the best-available implementations, which exploit only data parallelism.

The remainder of this paper is organized as follows. Section 2 discusses some previous works that considered different programing models for GPUs . Section 3 provides background on GPGPU programming and task parallelism. Section 4 discusses our basic task-queue-based technique for exploiting data parallelism in task-parallel applications. Section 5 discusses the need for locality aware scheduling of subtasks. Section 6 discusses the implementation, Section 7 evaluates our system on the four applications mentioned above, and Section 8 concludes.

2 Related Work

Due to the increasing prevalence of hardware resources for data parallelism (GPUs, SIMD units, etc.), there has been significant recent interest in techniques for mapping task-parallel computations to data parallel hardware. Gaster and Howes propose a *channels* abstraction for executing Cilk-style task-parallel programs on GPUs, where the GPU hardware manages queues for each type of task, and provides support for dequeueing and enqueuing new tasks [5]. Orr et al. presented an instantiation of the channels model and showed its efficacy on several small Cilk-style programs [12]. Both of these approaches require hardware support, and hence are not suitable to executing task-parallel programs on commodity data-parallel hardware. Moreover, the channels abstraction does not consider locality between tasks; it only concerns itself with grouping together tasks with similar computation.

More recently, Ren et al. described a series of code transformations that transform task-parallel algorithms into *blocked* recursive algorithms: recursive algorithms where each method invocation performs a block of tasks, rather than a single task [13]. These blocks can be executed efficiently in a vectorized manner. While Ren et al.'s general approach—transforming independent, parallel tasks into data-parallel blocks of tasks—is similar to ours, their technique does not apply in our setting for two reasons: 1) they target vector units on CPUs, and hence can support code transformations that require fine-grained interleaving of SIMD and scalar operations; 2) more importantly, the applications they study only manipulate the stack, and hence their technique does not have to account for locality considerations.

In a more general sense, models for executing work-queues on GPUs have been studied extensively in the literature. The *persistent threads* model [7] proposes maintaining a CPU-managed work-queue along with a specially-designed GPU kernel where a limited number of threads each run a simple get-work/execute-work loop until the software-managed queue is empty. This style of programming can be conducive to expressing idioms, such as producer-consumer dependences, that data-parallel programs are ill-suited to capture. This model has been used to implement several work-queue-style applications [10, 2]. For the most part, persistent-thread applications do not consider locality in mapping

tasks to threads. Moreover, unlike in a persistent thread model, our approach does not attempt to limit the number of threads executing on the GPU; instead, the entire queue of tasks is sent to the GPU at the same time, to maximize the effectiveness of hardware multithreading.

Chen at al. do consider locality concerns in a persistent-thread-like model [3]. In their programming model, tasks in a task-queue can generate new tasks that may operate on similar data to the parent tasks. They use a compiler-based code transformation to map child tasks to the threads that executed the parent task, to promote reuse. In contrast, our approach considers locality between the tasks that are mapped to the same warp—in other words, locality between tasks that are executed by different threads, rather than consecutively on a single thread—in an attempt to improve memory coalescing and minimize cache pressure. Wu et al. perform affinity scheduling, mapping tasks with overlapping footprints to the same Streaming Multiprocessor (SM) [15]; this notion of locality is far more coarse-grained than the warp-focused affinity scheduling we pursue. Moreover, neither Chen et al. nor Wu et al. focus on the type of task-parallel applications that we tackle.

Goldfarb et al. looked at mapping tree traversal applications to the GPU, as with two of our example benchmarks [6]. They adopt a fully data-parallel approach, meaning that their implementation relies on inputs with a large number of traversals to obtain good performance. Moreover, they do not consider locality between threads, relying on *ad hoc*, programmer-provided scheduling decisions. Liu et al. expanded on this work by developing a hybrid scheduling framework that attempts to reschedule traversals based on similarity [8]. As in Goldfarb et al.'s work, Liu et al. only consider fully data-parallel implementations, and rely on high degrees of data parallelism for their scheduling to be effective. Their implementations, which represent highly optimized GPU implementations of tree traversal applications, form the baseline for our experiments.

3 Background and Motivation

3.1 GPU architecture and limitations

A typical GPU consists of multiple streaming multiprocessor units (SMs), each of which features multiple simple cores, a register file, an L1 cache, and a shared memory used by threads within the same thread block to communicate (in NVIDIA GPUs, the shared memory and L1 cache share the same hardware structure, which can be partitioned between the two in different ratios, depending on the workload). There is also a shared L2 cache among all the SMs on the GPU. Finally, there is an off-chip, global memory accessible by all the SMs.

Execution on a GPU consists of a *kernel* that is expressed in terms of a *thread grid*—a set of threads that execute in parallel to complete the kernel[1]. The thread grid is executed across one or more of the SMs on the GPU. To

[1] For convenience, we use NVIDIA's CUDA terminology to explain the GPU programming model; OpenCL has analogous constructs, but uses different terms.

facilitate this execution, the threads in the grid are partitioned into multiple *thread blocks*. While different thread blocks may execute on different SMs, all the threads in a single block are guaranteed to execute on a single SM. As a result of this thread partitioning, threads in a thread block can communicate through shared memory, but threads in a grid can only communicate through global memory.

Within a thread block, execution proceeds by dividing the block into *warps*: groups of 32 threads that execute *simultaneously* on the SM's 32 cores. These threads execute in a lockstep, SIMT (single instruction multiple thread) manner, for efficiency: all threads must be executing the same instruction for computations to be performed in parallel, and if some threads want to execute different instructions, some of the 32 cores sit idle until the threads in the warp return to executing the same instruction. Paired with this *control divergence* is *memory divergence*: if multiple threads in the warp issue a load, all the threads must wait until all of the loads complete before proceeding. Hence, if some loads miss in the cache, the entire warp can stall for a long time before resuming execution.

As a final complication, GPU cores are extremely simple—in order, no forwarding networks, no branch prediction, etc. Instead, performance is maintained in the face of pipeline stalls and memory divergence through massive multithreading: NVIDIA's latest Kepler GPUs can keep up to 64 warps (2048 threads) in context at the same time, and will context switch between these warps on stalls. Note that this massive multithreading means that a GPU's memory system is much smaller than a CPU's on a per thread basis: a CPU hardware context (core) has access to a 64 KB private L1 cache, and a ∼1 MB L2 cache and ∼6 MB L3 cache shared among 4-12 cores. In contrast, an SM's 16-64KB of L1 cache is shared among up to *two thousand* threads, and its ∼1.5 MB of L2 cache is shared among all of the SMs in the system. On the flip side, while the amount of memory per thread may be small, the GPU features extremely high throughput to keep the threads fed.

These hardware features conspire in destructive ways when writing programs that do not have very well-structured computation and memory accesses:

- A GPU needs large amounts of parallelism to sustain throughput under memory stalls.
- Memory stalls are more likely due to the SIMT architecture if different threads in the same warp have divergent memory footprints, as each SIMT-coalesced memory access is more likely to result in cache misses.
- These memory stalls require even more parallelism to hide the resulting latency, which places even more pressure on the memory system.

Unsurprisingly, then, while GPU implementations attain extremely high speedups over CPU implementations for data-parallel, regular applications, as programs become less regular and less data-parallel, speedups become harder to attain. This paper describes one approach to achieve good speedup for a class of irregular, task-parallel applications.

```
1 corr(KDNode n, Point p, float r) {
2   if (!canCorrelate(n, p, r)) return;
3   else
4     if (n.isLeaf && dist(n, p) < r)
5       p.count.accum(1);
6     else if (!n.isLeaf)
7       spawn corr(n.left, p, r);
8       spawn corr(n.right, p, r);
9 }
```

Fig. 1. Example task-parallel algorithm for point correlation

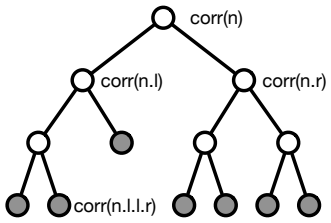

Fig. 2. Computation tree for point-correlation

3.2 Task parallelism

In this paper, we consider mapping *task-parallel* applications on GPUs. In particular, we consider recursive and task-parallel applications where parallelism arises from executing multiple recursive function invocations simultaneously. Figure 1 shows a task-parallel implementation of a recursive algorithm to compute two-point correlation. The algorithm takes a point p and traverses a kd-tree structure to determine how many points in a metric space are within a specified radius r of that point. Because the kd-tree is a binary tree, each subtree can be searched independently, as indicated by the use of the spawn keyword (we borrow the keyword from Cilk [1], perhaps the most well-known task-parallel programming languages). Following the approach of Ren et al. [13], rather than using sync and return values to perform the final correlation computation, we instead accumulate into a Cilk-style reducer [4]. In section 4, we explain how we use this reduction approach in our implementations.

Two-point correlation has substantial amounts of parallelism, but, nevertheless, is poorly suited to mapping to a GPU: the parallel operations do not arise from a data parallel loop; the parallel operations have very different memory footprints; and the computation itself is highly irregular. While some data-parallelism does arise because this task-parallel computation can be repeated for different points, there may not always be enough data parallelism to provide the parallelism the GPU requires. The best-available GPU implementation of point correlation [8] relies on massive data parallelism to effectively manage the irregularity of the computation.

4 Data parallel GPU execution of task parallel code

This section presents our basic technique for extracting data parallelism from task-parallel programs. It shares some basic similarities with the approach of Orr et al. [12] and Ren et al. [13], in that it "expands" out the task parallel work to generate enough tasks that can subsequently be executed in a data parallel manner. Unlike the prior two approaches, though, our approach focuses on cooperation between the CPU and GPU to generate the necessary tasks.

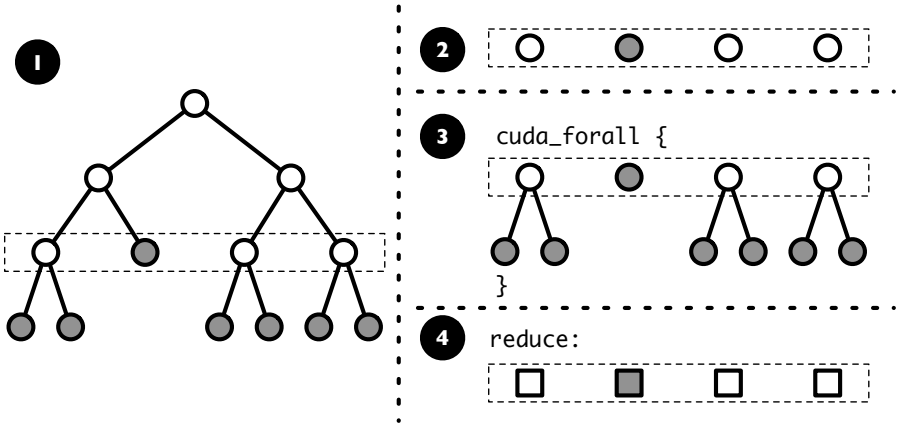

Fig. 3. Steps for executing task parallel code on GPU

4.1 Basic technique

The key insight underpinning our approach is that the execution of a recursive, task-parallel program with no syncs can be viewed as an execution tree: a program begins at the root of the tree, and at each spawn call generates two leaves: one for the spawn, and one for the continuation. If there is no continuation (i.e., the spawn is the last operation in the function), then only one leaf is generated. Hence, in a code like the point correlation code of Figure 1, non-base-case executions generate interior nodes of the tree with two leaves each, for each of the spawn calls, and base case invocations of the corr method create leaves of this execution tree. Figure 2 shows an example of the execution tree, with base-case tasks shaded gray. Some of the nodes are labeled with the node argument passed to the task. Note that the nature of the point correlation algorithm means that the execution tree corresponds to the *actual* tree that is being traversed: each method invocation corresponds to operating on a subtree of the overall kd-tree. This connection will become pertinent in Section 5.

Because of the nature of task-parallel execution, the nodes in this computation tree can be executed in any order, provided that ancestors always execute before descendants. As nodes are executed, the "frontier" of non-executed nodes represents the set of tasks that are left to be executed (so, for example, in a Cilk-style runtime system, the contents of the threads' deques are this frontier).

Our execution model, then, is straightforward. Figure 3 illustrates the steps.

1. Partially expand the computation tree on the CPU, until a sufficient frontier is generated. (The non-greyed out nodes in the computation tree shown in step 1 of Figure 3.)
2. Place all the tasks in this frontier into a queue (shown in step 2 of Figure 3). Note that each of these tasks, by definition, is independent from the others.

```
1 corr(KDNode n, Point p, float r) {
2   if (!canCorrelate(n, p, r)) return;
3   else
4     if (n.isLeaf && dist(n, p) < r)
5       p.count.accum(1);
6     else if (!n.isLeaf)
7       if (!thresholdMet)
8         spawn corr(n.left, p, r);
9         spawn corr(n.right, p, r);
10      else
11        gpuQ.addTask(n.left, p, r);
12        gpuQ.addTask(n.right, p, r);
13 }
```

Fig. 4. Transformed point correlation algorithm to enable GPU task queue.

Moreover, each task can be executed to completion sequentially, executing the entire subtree rooted at that task.

3. Execute this task queue in a *data parallel* manner on the GPU, with each task maintaining its own reduction result (shown in step 3 of Figure 3). Because of the nature of reduction computations [4], each of these tasks can perform its reductions independently.
4. Return the reduction objects (the squares in step 4 of Figure 3 to the CPU to be combined to produce the final result.

Note that because of the recursive nature of the tasks in the applications we consider, each of the tasks has a fairly similar computational fingerprint. This similarity between tasks helps reduce control divergence. Memory divergence, though, is another issue, as Section 5 elaborates.

4.2 Generating GPU Task Queues

The process of enqueuing tasks into the task queue for execution on the GPU is straightforward, and can be accomplished by a basic code transformation. Because spawned tasks are independent of their execution, it is also safe to *not* execute them, and instead defer their execution until a later point in time. Hence, during *sequential* execution of a task-parallel program on the CPU, whenever a threshold is hit, spawned tasks are not executed, but are instead enqueued onto a task queue that will be sent to the GPU. Figure 4 shows a version of our point correlation example that performs this enqueuing.

Note that we do not attempt to perform the CPU work in parallel, though it could reasonably be parallelized; because the work up to the frontier represented by the task queue is small compared to the overall work in the program, executing this work sequentially is a minor overhead, and simplifies implementation.

4.3 Mixing data parallelism and task parallelism

As mentioned in Section 3.2, some applications have a mix of data and task parallelism. In particular, applications such as point correlation often have a data parallel outer loop (in this case, iterating over multiple points), where each iteration of that data parallel loop performs task-parallel work.

Integrating such algorithms into our framework is simple: we merely execute the data-parallel outer loop sequentially, and, upon entering a task parallel iteration, execute it according to the scheme above. Because most of the work is performed by the tasks enqueued into the GPU queue, the data parallel outer loop will "finish," having enqueued most of its computation into the GPU queue. We can then execute the GPU queue and proceed as before. This transformation is safe, since the data-parallel iterations are independent of one another, and each iteration's task-parallel tasks are also independent, hence all the tasks, even if they arise from different iterations, are independent.

Note that in the particular case of point correlation, this execution strategy leads to poor data locality. Suppose we want to process 1000 points. Suppose, further, that the enqueuing threshold for the task parallel computation is two levels deep in the tree (as in Figure 3). Then each point's execution will lead to the creation of four tasks, touching four different subtrees of the kd-tree, and overall 4000 total tasks will be created, with 1000 tasks touching each subtree. However, if the tasks are placed into the GPU queue in order, then each of a point's tasks will be placed contiguously into the queue. Because these threads are likely to be placed in the same warp during GPU execution, each warp's memory footprint will span the entire tree, leading to very poor locality, and hence poor performance. The next section discusses how to solve this problem.

5 Scheduling for Locality

The execution strategy outlined in the previous section solves the initial problem of executing a task-parallel application on hardware built for data-parallelism. But, as pointed out in Section 3.1, while GPUs are very efficient data-parallel execution engines, their efficiency comes with several drawbacks. In particular, the memory resources of the GPU are not well-matched to the massive parallelism that efficient execution requires: the GPU features substantial bandwidth (allowing the threads to be fed), but very small cache resources. As a result, GPUs work well in streaming workloads or workloads with small reuse footprints. But in workloads with large amounts of reuse but also large footprints, the small caches can dramatically reduce performance. Indeed, to avoid the thrashing that can result from too many threads contending for the same small caches, it is often necessary to reduce an SM's warp count from 64 (the maximum supported) to only four or five [14]. Unfortunately, the kinds of irregular, task-parallel workloads we target are precisely workloads that feature substantial parallelism (due to our execution strategy), but potentially-large memory footprints (the trees traversed in point correlation, for example, can feature millions of nodes). Another problem arises in combating memory divergence: while keeping the foot-

print of a block to a minimum to avoid cache thrashing is important, it is also important to ensure that the threads of a single warp do not encounter widely varying memory latencies: if one thread in a warp encounters a cache miss while the others do not, all the threads pay the penalty of that cache miss.

To address these problems, we propose *locality-aware queue scheduling*. Orr et al. proposed a multi-queue strategy for executing task parallel program where different queues correspond to different types of computations (to reduce control divergence) [12]. Instead, we propose to use multiple queues where different queues correspond to different *locality domains*: an abstract notion of the region of memory a task might access. We create a separate queue for each such locality domain, and tasks expected to access similar regions of memory will be assigned to the same queue. By placing threads from the same locality domain together, we promote threads in the same warp touching similar pieces of memory, both reducing footprints and decreasing memory divergence.

Locality-aware scheduling requires some understanding of the memory access patterns of the tasks that are being scheduled. This is inherently an application-specific property: various features of the application, and the specific task, might be used to map the task to a particular locality domain. To support this type of scheduling, we add an additional parameter to the addTask hook, where the programmer can pass the result of evaluating a simple function to compute the locality domain for the scheduled task.

In many cases, it is straightforward to determine a task's locality domain. For example, in tree-based benchmarks, we originally start with independent tasks each accessing the whole tree i.e. we start with one locality domain. When subdividing these tasks into independent subtasks, each traversal gets divided into multiple traversals each accessing a part of the tree. This suggests a very simple representation of each locality domain: the node the task is invoked on, which is the root of the subtree it will access. Thus, the two addTask calls from Figure 4 can be replaced with the following:

```
1 gpuQ.getQueue(n.left).addTask(n.left, p, r);
2 gpuQ.getQueue(n.right).addTask(n.right, p, r);
```

Note the effect of this implementation on the mixed data- and task-parallelism scenario from Section 4.3. If there are four different subtrees that an enqueued task could access, there will be four separate queues. Each point will enqueue its four tasks onto the four separate queues, and task *from different points that access the same subtree* will be placed next to each other in each of the queues, promoting locality.

6 Implementation

This section describes a few aspects of implementing the scheduling and execution strategy from the previous two sections.

6.1 Determining the queue threshold

The pseudocode in Figure 4 uses an arbitrary threshold for determining when to stop expanding out the computation tree and to instead enqueue tasks into the GPU queue for later execution. In general, the threshold is application-specific, and possibly input-specific. However, we have found that a good rule of thumb is to ensure that the memory footprint of each task is fairly small. Recall that GPUs have limited per-thread memory resources. Hence, if the tasks in the task queue have large footprints, each thread will demand substantial amounts of memory, increasing the per-block memory footprint and lowering performance. Moreover, by keeping tasks relatively small, the likelihood that tasks diverge significantly during execution is reduced, helping further mitigate control divergence. Finally, by keeping individual task footprints small, the total number of tasks increases (because the CPU waits longer before hitting the enqueue threshold), creating enough parallelism to keep the GPU busy. So, for example, in our point correlation example, we might set the threshold at a depth such that the subtree visited by each enqueued task is relatively small.

6.2 Queue merging

If there are many locality-based queues, the overhead of sending each queue to the GPU separately can be prohibitive. Instead, once the queues are constructed by the CPU expansion of the computation tree are complete, the queues are merged into a single queue and sent to the GPU. Because the threads in the unified queue are still ordered according to their locality-based queues, the schedule of execution will still be consistent with the locality-aware grouping.

6.3 Queue size reduction

If there are too many tasks in the queues after expansion, there may be too much parallelism for the GPU, resulting in more scheduling overhead. To avoid this, we implement a queue size reduction optimization. Rather than passing the task queue to the GPU to be executed with a simple do-all loop, where each iteration gets mapped to a separate thread, we can instead rewrite the task queue loop so that each thread processes two (or more) tasks from the queue (essentially, by strip mining the loop that processes the task queue). Because each thread will execute consecutive tasks in the task queue, the locality-aware scheduling policy will promote those two tasks' having overlapping memory footprints, thus not increasing the memory footprint of a given thread, coarsening the computation without increasing cache pressure.

Note that this general strategy: of having a smaller number of threads execute a larger number of tasks by scheduling multiple tasks per thread, is similar to the approach advocated by persistent threads [7]. A key difference is that persistent threads approaches *dynamically* schedule tasks to threads, while we *statically* schedule tasks to threads. While dynamic scheduling can be more flexible, our static scheduling has two advantages: 1) static scheduling means that

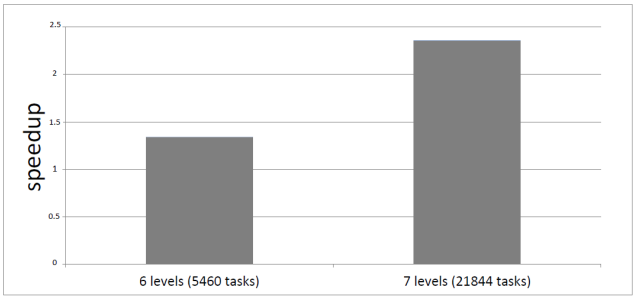

Fig. 5. Speedup of locality-aware FMM over locality-agnostic FMM

the multiple tasks scheduled to each thread are guaranteed to come from the same locality domain, improving locality; 2) static scheduling means that we avoid the runtime overhead of managing the task queue.

7 Evaluation

We evaluate our locality-aware task scheduling approach on three task-parallel applications: fast multipole method (FMM), point correlation (PC), and nearest-neighbor (NN). The CPU on which the experiments are conducted has 2 AMD Opteron 6164 HE processors, each of which has 12 cores running at 1.7 GHz, with 32 GB of system memory. The GPU on which the experiments are run is an nVidia Tesla K20C with 5120 MB of RAM and 2496 CUDA cores. The runtimes of our implementations include the time spent on the CPU to generate tasks and communicate the task queue to the GPU, as well as the time to retrieve the reduction objects from the GPU and complete the reduction. In other words, our implementations' runtimes are directly comparable to GPU-only execution.

7.1 Fast multipole method

The fast multipole method is a fast approximation algorithm for the n-body problem. It operates by performing a bottom-up traversal of an quad-tree, at each level of the tree computing for each subtree at that level the forces contributed by the bodies in that subtree on neighboring subtrees. The task parallelism in this program arises because the subtrees can be processed in parallel.

For FMM, we compare a locality-agnostic implementation of our approach to one where subtrees represent locality domains, and hence tasks that operate on the same subtree are grouped together. Figure 5 shows the speedup of the locality-aware approach to the locality-agnostic approach, with two different task granularities: one where tasks originate 6-levels deep in the tree, and one where they originate 7-levels deep. We see that with the coarser-grained tasks, being locality-aware provides a 1.34× speedup. With finer-grained tasks, where grouping together similar tasks results in a smaller footprint that better utilizes the GPU's caches, we can achieve a 2.36× speedup over the locality-agnostic implementation.

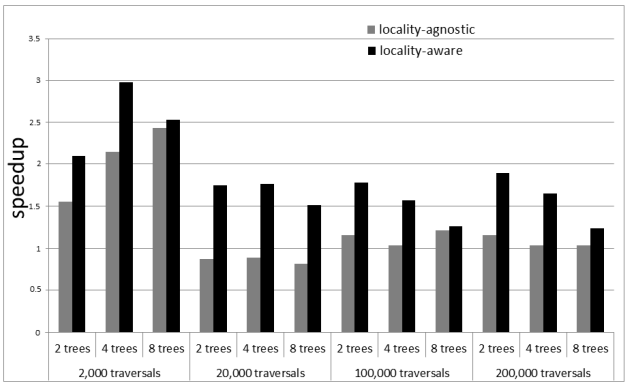

Fig. 6. Speedup of PC over data-parallel GPU baseline

7.2 Point correlation

Point correlation, our running example, is a mixed data- and task-parallel benchmark: each of a set of independent points traverses a tree (data parallelism), and those points can traverse the tree in parallel by processing independent subtrees simultaneously (task parallelism). Unlike for FMM, where we compare the locality-aware and locality-agnostic implementations of our framework, for PC, we also compare against an optimized GPU baseline [8]. This GPU baseline exploits only data parallelism.

Figure 6 shows the speedup of our locality-agnostic and locality-aware task queue implementations over the data-parallel baseline. We varied both the enqueuing threshold (a given number of trees is the number of tasks per point we generate) and the number of traversals we perform. We see that the locality-aware implementation is consistently faster than the locality-unaware implementation. Indeed, the locality-aware implementation is consistently faster than the baseline data-parallel implementation. Further, we see the effect of the enqueuing threshold on performance: for this particular application, generating four tasks per point provides a good balance between overhead (more tasks means more work generating tasks and performing reductions on the CPU) and locality. Finally, we see that when there are only a small number of traversals, the data-parallel implementation is significantly slower than our mixed implementation, as we are able to generate additional parallelism to keep the GPU busy, achieving a speedup of almost 3× over the optimized baseline.

7.3 Nearest neighbor

We perform a similar experiment as PC for our nearest-neighbor benchmark, again comparing our locality-agnostic and locality-aware implementations to an optimized, data-parallel-only baseline. Figure 7 shows again that our mixed implementations are consistently faster than the data-parallel-only implementation, and that adding locality awareness consistently adds performance. We again see

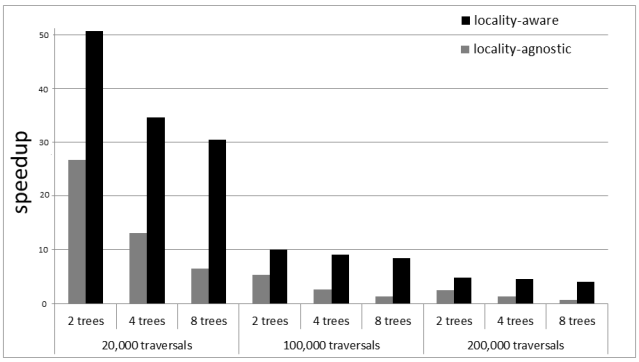

Fig. 7. Speedup of NN over data-parallel GPU baseline

the effect of being able to exploit additional parallelism: for 20K point inputs, our locality-aware task queue implementation is over 50× faster than the baseline!

8 Conclusions

This paper presented an approach to exploiting task-parallelism on GPUs by performing partial execution on the GPU to generate a queue of data-parallel tasks that can be executed on the GPU. We then showed how multiple such task queues can be used to add locality-awareness, with the goal of shrinking the memory footprint of individual warps to reduce cache pressure. Preliminary results are promising, showing not only that our approach to exploiting task-parallelism can result in efficient implementations, but also that locality-awareness can significantly boost performance. Indeed, for an implementation of nearest-neighbor, our approach yields a performance improvement of 50× over an optimized data-parallel implementation. Future work could explore methods of defining locality domains for applications where statically defining the domain is not possible.

9 Acknowledgments

The authors would like to thank the anonymous referees for their comments and feedback regarding the paper. This work was supported in part by the U.S. Department of Energy's (DOE) Office of Science, Office of Advanced Scientific Computing Research, under DOE Early Career Award DE-SC0010295. This work was also supported in part by NSF awards CCF-1150013 (CAREER) and CCF-1439126.

References

1. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime

System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25, 1996.

2. Nicola Capodieci and Paolo Burgio. Efficient implementation of genetic algorithms on gp-gpu with scheduled persistent cuda threads. In *Parallel Architectures, Algorithms and Programming (PAAP), 2015 Seventh International Symposium on*, pages 6–12. IEEE, 2015.
3. Guoyang Chen and Xipeng Shen. Free launch: optimizing gpu dynamic kernel launches through thread reuse. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 407–419. ACM, 2015.
4. Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and Other Cilk++ Hyperobjects. In *SPAA '09*, pages 79–90, 2009.
5. B.R. Gaster and L. Howes. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer*, 45(8):42–52, August 2012.
6. Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. General Transformations for GPU Execution of Tree Traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, SC '13, 2013.
7. K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14, May 2012.
8. Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. Hybrid cpu-gpu scheduling and execution of tree traversals. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 41:1–41:2, New York, NY, USA, 2016. ACM.
9. A. MUNSHI. OpenCL Parallel Computing on the GPU and CPU. *SIGGRAPH*.
10. Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-driven versus topology-driven irregular computations on gpus. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 463–474. IEEE, 2013.
11. NVIDIA. CUDA. `http://www.nvidia.com/object/cuda_home_new.html`.
12. Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. Fine-grain Task Aggregation and Coordination on GPUs. In *ISCA '14*, pages 181–192, 2014.
13. Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. Efficient Execution of Recursive Programs on Commodity Vector Hardware. In *PLDI*, pages 509–520, 2015.
14. Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.
15. Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130. ACM, 2015.