2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

CUDA Grid-Level Task Progression Algorithms

Christos Kartsaklis, Wayne Joubert, Oscar R. Hernandez, Markus Eisenbach, Wael R. Elwasif, David E. Bernholdt
Oak Ridge National Laboratory
Oak Ridge, TN, United States
Email: {kartsaklisc, joubert, oscarh, eisenbachm, elwasifwr, bernholdtde}@ornl.gov

***Abstract*—Tasking is a prominent parallel programming model. In this paper we conduct a first study into the feasibility of task-parallel execution at the CUDA grid, rather than the stream/kernel level, for regular, fixed in-out dependency task graphs, similar to those found in wavefront computational patterns, making the findings broadly applicable. We propose and evaluate three CUDA task progression algorithms, where threadblocks cooperatively process the task graph, and argue about their performance in terms of tasking throughput, atomics and memory IO overheads. Our initial results demonstrate a throughput of 38 million tasks/second on a Kepler K20X architecture.**

I. Introduction

Tasking is a promising parallel programming model that is often suggested as a candidate for future exascale systems if scheduling and overheads issues can be overcome ([1], [2], [3], [4]).

Programming unconventional computing devices, such as GPGPU accelerators, remains a complex problem mainly due to their programming model, which exhibits new levels of concurrency, less guarantees on ordering and intricate forms of scheduling, often programmable via high-level parallel APIs ([5], [6], [7], [8]). Defining or even implementing tasking for GPUs is particularly challenging, with existing approaches targeting coarser threads of execution ([9], [10], [11], [12]). So dramatic are, however, the improvements when successfully tapping these devices' performance, that makes the race towards enabling parallel programming APIs, which are known to address irregular and unstructured forms of parallelism, even more important.

This work will show that it is possible to manage task progression in its entirety on a GPU, using the threadblocks of a single CUDA kernel as the means for doing so. specifically, we will show that CUDA threadblocks can co-operate, in a concurrent fashion, to notify each-other of task availability without the host's support.

Our work had originally targeted the KBA sweep algorithm in an effort to enable tasking for the Denovo radiation transport kernel ([13], [14]), but it immediately became clear that it had broader application. The KBA sweep algorithm is used to solve problems in deterministic radiation transport. For these problems, modeling the flux of particles flowing through a spatial volume results in a coupling of the computations between neighboring gridcells. In particular, for the 3D structured grid case which is commonly implemented, each gridcell depends for the computation of its result on the neighbor upstream gridcells in each of the three coordinate directions. These dependencies lead to a restriction on the allowable computational patterns or sequence of operations that could be used to perform the computation. Problems of this type are often solved by wavefront methods. Originally proposed by Lamport [15], wavefront computations have had applications to diverse areas including linear equation solvers [16], [17], sequence alignment [18] and radiation transport [19]. Their parallelization challenge lies in the inherently recursive data coupling [20] which necessitates the decomposition of the problem into wavefronts with restricted parallelism and potential load imbalance.

Section II provides an overview of related work. Section III, proceeds to describe our wavefront problem, its "taskification", and three CUDA grid-parallel algorithms for processing the corresponding task graph. It is then followed by Section IV, which presents some preliminary results and their analysis. We finally conclude with some remarks on findings and next steps.

II. Related Work

There are several task parallel languages and runtime libraries that have been used to parallelize irregular and dynamic applications. OpenMP task parallelism applies to dynamic applications because it provides a mechanism for expressing parallelism on irregular regions of code where dependencies can be satisfied at runtime. It has been shown ([21], [22], [23]) that OpenMP tasks are more efficient in parallelizing wavefronts and graph-based applications than thread-level parallelism because it is easier to express the parallelism on unstructured regions while leaving the task scheduling decisions to the runtime. However, such studies are limited in how to map tasks to accelerators or applications with large numbers of threads. Load imbalances, scheduling overheads and work inflation (due to data locality) can adversely affect the efficiency of task parallelism at scale [24]. These sources of overhead need to be mitigated carefully in applications, especially at large scale. OpenMP 3.1 provides mechanisms to manage some of these overheads by allowing work stealing with the *untied* clause to improve load balance, reducing the memory overheads by merging the data environment of tasks with the mergeable clause, and by reducing the task overhead with the specification of undeferred and included tasks via the *if* and *final* clauses. However, little is understood about how

these overheads will be affected when mapping tasks to accelerators, especially GPUs. Most approaches focus supporting tasks at the CUDA/OpenCL kernel level, which manifests as a host runtime that is capable of orchestrating the concurrent launching of kernels on the devices and the transfer of data back and forth, which are often modeled as tasks too ([9], [10], [11], [12]). Efficiently isolating these responsibilities at the device level is a prerequisite for making task programming models a viable model for accelerator computing.

III. Tasked Wavefront Computations

The left panel of Figure 2 depicts our source problem's data grid (cells) and their dependencies: to update each cell (i, j) one needs to update the adjacent cells $(i-1, j)$ and $(i, j-1)$ first, if defined. In a typical wavefront loop, one may imagine the computation proceeding as follows: cell $(0, 0)$ first, then cells $(1, 0)$ and $(0, 1)$, cells $(2, 0)$, $(1, 1)$ and $(0, 2)$, i.e. in "waves". Waves have to be interleaved by a barrier to prevent races such as working on $(2, 0)$ before $(1, 0)$ has been updated. Synchronization is an expensive operation. In task-parallel mode, one decomposes work into tasks while defining the order by which these tasks may be executed in. Tasking runtimes operate on a task graph data structure – a dependency graph variant that captures the dependencies among tasks. The main responsibility of the runtime then is to notify the processing elements (threads/processes) of tasks that are ready to run. We will talk now about the structures that are specific to our domain.

We have selected an ultra fine-grain decomposition, where tasks are made to correspond directly to gridcells. Let us define the dependency matrix D as an $nrows \times ncols$ matrix, with its entries initialized as shown below. The value of $D_{i,j}$ is the number of cells that cell (i, j) depends on. At runtime, if $D_{i,j}$ has never been visited and is found equal to 0, then this signifies that task (i, j) is ready to run. Clearly, since cell $(0, 0)$ is 0, the corresponding task is immediately ready to run. The right panel of Figure 2 shows an initialized dependency matrix.

$$D_{i,j} = \begin{cases} 0 & \text{, if } i = 0 \text{ and } j = 0 \\ 1 & \text{, if } (i > 0 \text{ and } j = 0) \text{ or } (i = 0 \text{ and } j > 0) \\ 2 & \text{, if } i > 0 \text{ and } j > 0 \end{cases} \quad (1)$$

The algorithm shown in Figure 1 provides a generic, serial, method for processing the task graph: find a cell (i, j) with $D_{i,j}$ equal to 0, execute it and notify those cells that depend on it that one of their dependencies, i.e. the one due to (i, j) has been satisfied. The next sections provide our three CUDA grid-parallel implementations for this algorithm.

Listing 1 provides the implementation of lock/unlock primitives that we built on top of CUDA atomics [25]. Each CUDA threadblock owns a unique tag (assuming an 1D grid), computed as `WQ_LOCKED_OFFSET` plus the block's index ($blockIdx.x$). The lock is expressed as a 32-bit integer such that when its contents are set to `WQ_UNLOCKED` the lock is considered unlocked, otherwise it is considered locked. The locking primitive then proceeds as follows: given a lock stored in $laddr$, the primitive attempts to swap its contents with the threadblocks' lock tag ($self$). If the lock has been already acquired by another threadblock, then the $atomicCAS$ will return a value other than `WQ_UNLOCKED`, i.e. the other threadblock's tag, and the acquisition will fail leading to a retry. The unlock operation then proceeds by swapping the threadblock's tag with the unlocked tag (`WQ_UNLOCKED`), which will allow other threads that may be invoking $lock$ to eventually acquire the lock. With this in mind, let us now discuss the three implementations.

while *tasks-exist()* **do**
  $(i, j) \leftarrow$get-ready();
  process((i, j));
  if $i < nrows - 1$ **then**
    $depmat(i+1, j) \leftarrow depmat(i+1, j) - 1$;
    if $depmat(i+1, j) = 0$ **then**
      set-ready($(i+1, j)$);
    end
  end
  if $j < ncols - 1$ **then**
    $depmat(i, j+1) \leftarrow depmat(i, j+1) - 1$;
    if $depmat(i, j+1) = 0$ **then**
      set-ready($(i, j+1)$);
    end
  end
end

Fig. 1. Progression algorithm for executing the graph's tasks.

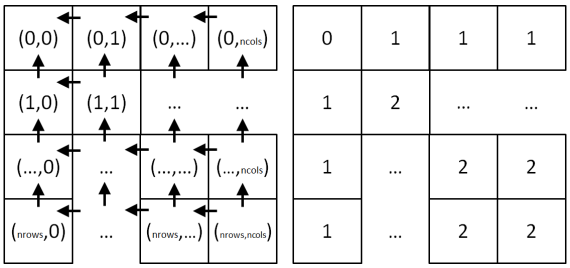


Fig. 2. Dependency graph (left) and dependency matrix D (right).

```
#define WQ_UNLOCKED         0
#define WQ_LOCKED_OFFSET 1

__device__ void trylock(tsint_t* laddr) {
   const tsint_t self = WQ_LOCKED_OFFSET+
        blockIdx.x;
   return (atomicCAS(laddr,WQ_UNLOCKED, self)
        == WQ_UNLOCKED);
}

__device__ void lock(tsint_t* laddr) {
   while (!trylock(laddr));
}

__device__ void unlock(tsint_t* laddr) {
   const tsint_t self =
            WQ_LOCKED_OFFSET + blockIdx.x;
   atomicCAS(laddr, self, WQ_UNLOCKED);
}
```

Listing 1. CUDA CAS-based lock/unlock primitives.

```
while (*counter != count) {
  for (int r=0 ; r<nrows ; ++r)
    for (int c=0 ; c<ncols ; ++c) {
      // cell id (linear)
      const int i = r*ncols + c;

// if all dependencies have been satisfied,
// try to lock the item:
      if (depmat[i] == 0
          && trylock(workqueue+i)) {
        ... // computational part
        depmat[i] = 1; // any !=0 value is OK

// decrease dependencies for neighbor (on
// our right)
        if (c!=(ncols-1)) {
          const int i_right = r*ncols + c+1;
          // CUDA intrinsic for atomic depmat
              [i_right]--;
          atomicDec(depmat+i_right, 2);
        }

// decrease dependencies for neighbor (below
// us)
        if (r!=(nrows-1)) {
          const int i_below = (r+1)*ncols+c;
          atomicDec(depmat+i_below, 2);
        }

        atomicInc(counter, count+1);
      }
    }
  }
}
```

Listing 2. Naive, dependency matrix re-scanning approach.

```
while (*counter != count) {
  lock(qstat);
  int avail = *qsize;
  if (avail>0) {  // pop the last task
    int taskid = queue[avail-1];
    *qsize--;
    unlock(qstat);
    ... // computational part

    bool append_right = false;
    bool append_below = false;

// update the dependencies of our two
// neighbors:
    const int i_right=...
    const int i_below=...
    if (c!=(ncols-1))
      append_right =
        atomicDec(depmat+i_right, 2)==1;
    if (r!=(nrows-1))
      append_below =
        atomicDec(depmat+i_below, 2)==1;

    if (append_right || append_below) {
      lock(qstat);
      int len = *qsize;
      if (append_right) queue[len++]= i_right;
      if (append_below) queue[len++]= i_below;
      *qsize = len;
      unlock(qstat);
    }
    atomicInc(counter, count+1); // *counter++
  } else
    unlock(qstat);
}
```

Listing 3. Single task queue – accessible by a one threadblock at a time.

A. Scanning

The first approach parallelizes the serial algorithm by atomically updating the dependencies ($depmat$) using CUDA atomics and via a per-task lock for claiming exclusive access ($workqueue$); it is shown in Listing 2. Each threadblock repeatedly scans (hence the naming) the dependency matrix for zero-valued entries since that signifies the relative task's readiness. If exclusive access cannot be granted immediately ($trylock$), the threadblock proceeds with processing the rest of the matrix, and eventually a restart of the entire operation. Otherwise, the threadblock proceeds with the computational task and the signaling of those depending on it. A global counter ($counter$) is atomically updated to reflect how many tasks have finished. Note that there is no $unlock$ operation present as it is unnecessary: simply setting $D_{i,j}$ value to non-zero will guarantee that the task will never appear in a ready state again.

B. Single-task queue

The second approach implements a shared queue of tasks protected by a single lock, thus allowing one threadblock in at a time. The CUDA code is shown in Listing 3. The queue occurs as a pre-allocated array, where tasks whose dependencies have been satisfied are meant to be placed in; the queue is initialized with task 0 (cell $(0,0)$) appended to it. The queue has a fixed capacity of $(nrows-1)\times(ncols-1)$, which corresponds to the maximum number of tasks that could be in-flight at any time. As in the first implementation, a global counter indicates completeness. Threadblocks atomically poll the queue for available work, pulling one task out a time. Once a threadblock is done with the computational part, it updates dependencies similarly to the previous algorithm (downcounting). However, while in the the previous algorithm the threadblock goes on to finding a new task to work on, this algorithm will immediately append those tasks (i_right and/or i_below) whose dependencies were just found satisfied to the queue.

C. Multi-task queue

The third approach, shown in Listing 4, uses multiple task queues and two hash functions, $hash1$ and $hash2$, to map threadblock ids ($blockIdx.x$) and task ids to the available queues, respectively. As before, each threadblock is assigned a queue that remains fixed for the entirety of the tasks' processing, while tasks get distributed to the various queues. The mechanism essentially relieves contention by distributing the locking operations to multiple locks. Similarly to before, threadblocks keep spinning until all tasks have been completed. However, care must be taken to avoid the following deadlock situation. In the CUDA model, kernels launch with a configurable number of threadblocks. The con-

```
myid = hash1(blockIdx.x);
while (*counter != count) {
  lock(qstat[myid]);
  int avail = *qsize[myid];
  if (avail>0) {  // pop the last task
    int taskid = queue[myid][avail-1];
    qsize[myid]--;
    unlock(qstat[myid]);
    ... // computational part

    bool append_right = false;
    bool append_below = false;

// update the dependencies of our two
// neighbors:
    const int i_right=...
    const int i_below=...
    if (c!=(ncols-1)) {
      if (atomicDec(depmat+i_right, 2)==1) {
        const int h = hash2(i_right);
        lock(qstat[h]);
        queue[h][len++] = i_right;
        qsize[h] = len;
        unlock(qstat[h]);
      }
    }
    ... // similarly for i_below
    atomicInc(counter, count+1); // *counter++
  } else
    unlock(qstat[myid]);
}
```

Listing 4. Multiple task queues.

currency in threadblock execution, or whether all requested threadblocks can be simultaneously active, is largely dependent on resource availability – the so-called CUDA occupancy. Under-resourcing leads to threadblock queuing, which means that some threadblocks cannot start unless some others have finished. For such, over-provisioning, threadblock counts tasks may be placed in queues whose owners will never get the chance to process them since the threadblocks which occupy the resources cannot finish unless all tasks have finished. In other words, we must chose a threadblock count that achieves 100% occupancy.

IV. Evaluation

We present results for a problem size of $10,000 \times 10,000$ cells, i.e. a total of 10^8 tasks (value of *count* in all algorithms). We have profiled our algorithms on a Kepler K20X NVIDIA GPU, using version 6.5 of the developers toolkit (CUDA timing routines and *nvprof* hardware counter probing). All algorithms are executed in a CUDA kernel context of 220 threadblocks with 32 threads each. The initialization of the dependency matrix and the queues have been excluded from our analysis due to their very low overhead. The computational part of the task is set to a *no-op*, hence all figures depict the task progression algorithms' overheads. Although these costs are likely to be hidden by a computationally-intense workload, our focus in this study is to assess overheads as a starting point in determining the granularity of computational tasks that our algorithms can be supportive of. The multi-queue algorithms have been profiled with three different queue counts, which are shown in parentheses. They correspond to 100%, 50% and 25% of the threadblock count. Both multi-queue algorithms use $hash1(x) = x$ % QUEUE-COUNT. Multi-queue algorithm 'a' implements $hash2$ identically to $hash1$, while algorithm 'b' uses $(x+3)$ % QUEUE-COUNT for i_right and $(x+5)$ % QUEUE-COUNT for i_below.

Figure 3a displays the task throughput, calculated as the number of tasks divided by the total time, with performance ranging between 315,000 to 38 million tasks/second. The single queue algorithm is the slowest, with the scanning algorithm and the multi queue algorithms improving performance 3-fold and 100-fold, respectively. The multi-queue algorithms are 27 times faster than the scanning approach. Throughput improves as queue count increases, with improvements being more pronounced for counts greater than 25%. The hash function of choice does play a significant role, with choice 'b' improving performance by 6% (2:1 threadblock:queue ratio) and 13% (1:1) over 'a'. For lower queue counts, the hash function's gains are masked by other overheads, that we will discuss shortly.

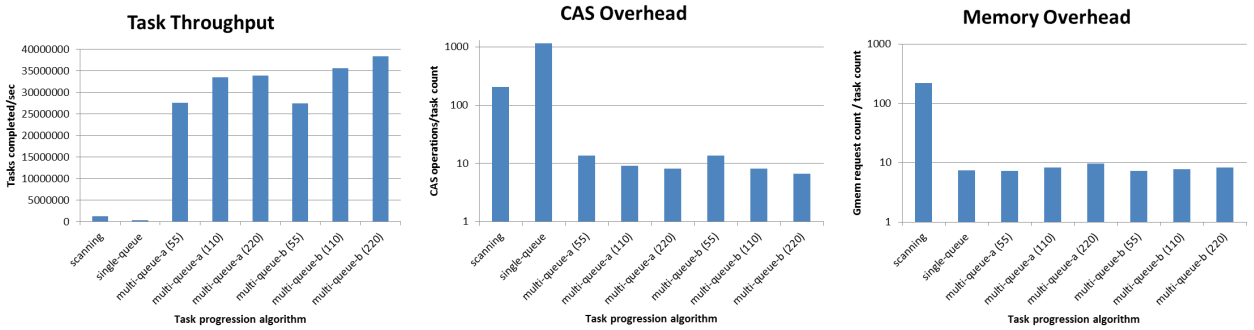
Figure 3b, which is in logarithmic scale, depicts the number of atomic *compare-and-swap* operations as a function of the task count (operations divided by tasks). One may, immediately, notice the correlation to Figure 3a: both *scanning* and *single* experience a staggering amount – more than a 100 times more – of CAS operations per task. For the *single* algorithm we know that it is the common queue, which threadblocks retry to acquire. For the scanning approach we believe it is the naivety of the algorithm that causes a high CAS count. Every time a threadblock completes a task, it then begins rescanning the matrix looking for ready tasks. Looking at memory overheads is supportive of this view. Figure 3c, which is also in logarithmic scale, depicts the number of of memory operations (load and store requests; `gld_request` and `gst_request` hardware counters) per task, computed as the sum of operations divided by the task count. The scanning approach clearly accesses memory significantly more than the rest of the algorithms – about $\sim$20 more operations. This is a result of scanning the matrix multiply due to both the restart, i.e. having completed the for-loop while additional work remains, but, most importantly, because every threadblock does so redundantly.

V. Conclusion

We have presented three different parallel algorithms for processing task graphs originating from wavefront programs on CUDA devices. The algorithms are implemented at the CUDA grid-level without any interference by the host and serve as a proof of concept that fine-grain tasking models can be supported efficiently at such a level. We are currently investigating the application of these algorithms to irregular patterns as well as different hash functions for better task load balance.

Acknowledgment

Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. De-AC05-00OR22725.



(a) Throughput (task completion per second).

(b) Compare-and-Swap operations per task (log scale).

(c) Memory load and store requests per task (log scale).

Fig. 3. Profiling results for a 10,000×10,000 problem size using a 220 threadblock CUDA grid.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] R. Cook, E. Dube *et al.*, "Survey of novel programmingmodels for parallelizing applications at exascale," *Raport instytutowy LLNL-TR-515971, Lawrence Livermore National Laboratory*, 2011.

[2] K. Wang, K. Brandstatter, and I. Raicu, "Simmatrix: Simulator for many-task computing execution fabric at exascale," in *Proceedings of the High Performance Computing Symposium*. Society for Computer Simulation International, 2013, p. 9.

[3] V. Subotić, S. Brinkmann *et al.*, "Programmability and portability for exascale: Top down programming methodology and tools with starss," *Journal of Computational Science*, vol. 4, no. 6, pp. 450–456, 2013.

[4] S. Amarasinghe, D. Campbell *et al.*, "Exascale software study: Software challenges in extreme scale systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

[5] S.-Z. Ueng, M. Lathara *et al.*, "Cuda-lite: Reducing gpu programming complexity," in *Languages and Compilers for Parallel Computing*. Springer, 2008, pp. 1–15.

[6] T. D. Han and T. S. Abdelrahman, "hi cuda: a high-level directive-based language for gpu programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 52–61.

[7] P. Du, R. Weber *et al.*, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[8] Q. Hou, K. Zhou, and B. Guo, "Bsgp: bulk-synchronous gpu programming," in *ACM Transactions on Graphics (TOG)*, vol. 27, no. 3. ACM, 2008, p. 19.

[9] M. Guevara, C. Gregg *et al.*, "Enabling task parallelism in the cuda scheduler," in *Workshop on Programming Models for Emerging Architectures*, ser. PMEA, Raleigh, NC, September 2009, pp. 69–76.

[10] L. Chen, O. Villa, and G. Gao, "Exploring fine-grained task-based execution on multi-gpu systems," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, Sept 2011, pp. 386–394.

[11] J. Bueno, J. Planas *et al.*, "Productive programming of GPU clusters with ompss," in *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, 2012, pp. 557–568.

[12] W. Wu, A. Bouteiller *et al.*, "Hierarchical dag scheduling for hybrid distributed systems," in *29th IEEE International Parallel & Distributed Processing Symposium*, Hyderabad, India, May 2015.

[13] C. Baker, G. Davidson *et al.*, "High performance radiation transport simulations: Preparing for titan," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–10.

[14] W. Joubert, R. Archibald *et al.*, "Accelerated application development: The ORNL Titan experience," *Computers & Electrical Engineering*, no. 0, pp. –, 2015.

[15] L. Lamport, "The Parallel Execution of DO Loops," *Commun. ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[16] R. Li and Y. Saad, "GPU-accelerated Preconditioned Iterative Linear Solvers," *J. Supercomput.*, vol. 63, no. 2, pp. 443–466, Feb. 2013.

[17] S. Pennycook, S. Hammond *et al.*, "On the Acceleration of Wavefront Applications Using Distributed Many-Core Architectures," *Comput. J.*, vol. 55, no. 2, pp. 138–153, Feb. 2012.

[18] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981.

[19] K. R. Koch, R. S. Baker, and R. E. Alcouffe, "Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor," *Transactions of the American Nuclear Society*, vol. 65, no. 108, pp. 198–199, 1992.

[20] J. Ortega and R. Voigt, *Solution of Partial Differential Equations on Vector and Parallel Computers*. Society for Industrial and Applied Mathematics, 1985.

[21] A. J. Dios, A. G. Navarro *et al.*, "A case study of the task-based parallel wavefront pattern," in *PARCO*, 2011, pp. 65–72.

[22] S. L. Olivier and J. F. Prins, "Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs," in *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, ser. IWOMP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 63–78.

[23] C. Terboven, D. Schmidl *et al.*, "Assessing OpenMP Tasking Implementations on NUMA Architectures," in *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, ser. IWOMP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 182–195.

[24] S. L. Olivier, B. R. de Supinski *et al.*, "Characterizing and Mitigating Work Time Inflation in Task Parallel Programs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 65:1–65:12.

[25] "CUDA C Programming Guide v7.0," http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 3 2015.