An efficient new static scheduling heuristic for accelerated architectures[⋆]

Thomas McSweeney[1][0000−0001−9866−2229]

University of Manchester, Manchester, UK, M13 9PL
thomas.mcsweeney@postgrad.manchester.ac.uk

Abstract. Heterogeneous architectures which make use of *Graphics Processing Units* (GPUs) for general computations, in addition to multicore CPUs, are increasingly common in high-performance computing. However many of the existing methods for scheduling precedence-constrained tasks on such platforms were intended for more diversely heterogeneous clusters, such as the classic *Heterogeneous Earliest Finish Time* (HEFT) heuristic. We propose a new static scheduling heuristic called *Heterogeneous Optimistic Finish Time* (HOFT) which has the same structure as HEFT but exploits the low degree of heterogeneity of CPU-GPU platforms. Through extensive experimentation using custom software for simulating task scheduling problems on user-defined CPU-GPU platforms, we show that HOFT can obtain schedules at least 5% shorter than HEFT's for medium to large-sized numerical linear algebra application task graphs and around 2% shorter on average for a large collection of randomly-generated graphs.

Keywords: GPU computing · static scheduling · task-based programming.

1 Introduction

Modern *High-Performance Computing* (HPC) machines typically comprise hundreds or even thousands of networked nodes. These nodes are increasingly likely to be *heterogeneous*, hosting one or more GPUs in addition to multicore CPUs. For example, Summit, which currently heads the Top500[1] list of the world's fastest supercomputers, comprises over 4000 nodes, each with two 22-core IBM Power9 CPUs and six NVIDIA Tesla V100 GPUs.

Task-based parallel programming is a paradigm that aims to harness this processor heterogeneity. Here a program is described as a collection of *tasks*—logically discrete atomic units of work—with *precedence constraints* that define the order in which they can be executed. This can be expressed in the form of a graph, where each vertex represents a task and edges the precedence constraints between them. We are interested only in the case when such task graphs are *Directed Acyclic Graphs* (DAGs)—directed and without any cycles.

[⋆] Supported by the Engineering and Physical Sciences Research Council.

[1] https://www.top500.org/

The immediate question is, how do we find the optimal way to assign the tasks to a set of heterogeneous resources while still respecting the precedence constraints? In other words, what *schedule* should we follow? This DAG scheduling problem is known to be NP-complete, even for homogeneous processors [16], so typically we must rely on heuristic solutions that give us reasonable solutions in a reasonable time.

A fundamental distinction is made between *static* and *dynamic* scheduling problems. Static schedules are fixed before execution based on the information available at that time, whereas dynamic schedules are determined during runtime. There are generic advantages and disadvantages to both: static scheduling makes greater use of the data so is superior when it is sufficiently accurate, whereas dynamic scheduling uses more recent data. In practice task scheduling is usually handled by a *runtime system*, such as OmpSs [10], PaRSEC [5], or StarPU [3]. Most such systems analyze previous execution traces to predict task execution and data transfer times at runtime. On a single machine this is tricky since shared buses make estimating available bandwidth difficult and the possibility of asynchronous data transfers complicates matters further. Hence at present dynamic scheduling is typically preferred. However static schedules can be surprisingly robust, even when the estimates used are poor [1]. Furthermore, robustness can be improved by sampling [17]. In addition, superior performance can be achieved in dynamic environments by modifying an existing static schedule, rather than computing a new one from scratch [1].

In this paper we therefore focus on the problem of finding good static schedules for multicore and GPU platforms. To facilitate this investigation, we developed an open-source software simulator which allows users to simulate the static scheduling of arbitrary task DAGs on arbitrary CPU-GPU platforms, without worrying about the time or energy usage constraints imposed by real systems. Using this, we evaluate and optimize HEFT for these environments.

HEFT comprises two phases: a *task prioritization* phase in which the order tasks are to be scheduled is determined and a *processor selection* phase in which they are actually assigned to the processing resources. In this article we introduce HOFT, which follows the HEFT framework but modifies both phases, without significantly increasing the complexity of the algorithm. HOFT works by first computing a table of *optimistic* estimates of the earliest possible times all tasks can be completed on both processor types and using this to guide both phases. Simulations with real and randomly-generated DAGs on both single and multiple GPU target platforms suggest that HOFT is always at least competitive with HEFT and on average obtains schedule makespans around 3% smaller.

Explicitly, the two main contributions of this paper are:

1. A new static scheduling heuristic that is optimized specifically for accelerated heterogeneous architectures;
2. Open-source simulation software that allows researchers to implement and evaluate their own scheduling methods for user-defined CPU-GPU platforms in a fast and reproducible manner.

The remainder of this paper is structured as follows. In Section 2 we summarize the relevant existing literature, before explicitly defining the simulation model we use to study the static task scheduling problem and software we have created which implements this in Section 3. We then describe HEFT in detail in Section 4, including also benchmarking results with our simulation model and a minor modification to the algorithm that we have found to perform well. In Section 5 we describe our new HOFT heuristic, before detailing the numerical experiments that we undertook to evaluate its performance in Section 6. Finally in Section 7 we state our conclusions from this investigation and outline future work that we believe may be useful.

2 Related work

Broadly, static scheduling methods can be divided into three categories: mathematical programming, guided-random and heuristics. (Note this taxonomy is our own.) The first is based on formulating the scheduling problem as a mathematical program and solving it using any applicable solver. See, for example, Kumar's constraint programming formulation in [13]. However these are usually so computationally expensive that they are restricted to very small DAGs. Guided-random is a term used for any method that generates a large population of potential schedules and then selects the best among them. Typically these are more general optimization schemes such as genetic algorithms which are refined for the task scheduling problem. As a rule, these methods tend to find very high-quality schedules but take a long time to do so; for example, a comparison by Braun et al. found that genetic algorithms usually obtained superior schedules to all other alternatives considered but were up to 300 times as expensive [6].

Heuristics are the most popular approach in practice as they are often both competitive with the alternatives and considerably faster. In turn *listing* heuristics are the most popular kind. They follow a two-phase structure: an ordered list of all tasks is first constructed (task prioritization) and they are then scheduled in this order according to some rule (processor selection). HEFT is the most prominent example: all tasks are prioritized according to their *upward rank* and then scheduled on the processor expected to complete their execution at the earliest time; a fuller description is given in Section 4. Canon et al. [7] compared twenty different task scheduling heuristics and found that HEFT was almost always among the best in terms of both schedule makespan and robustness.

Many modifications of HEFT have been proposed in the literature, such as HEFT *with lookahead* from Bittencourt, Sakellariou and Madeira [4], which has the same task prioritization phase but schedules all tasks on the resources estimated to minimize the completion time of their *children*. This has the effect of increasing the time complexity of the algorithm so, in an attempt to incorporate a degree of lookahead into the HEFT framework without increasing the cost, Arabnejad and Barbosa proposed *Predict Earliest Finish Time* (PEFT) [2]. The main innovation is that rather than just minimizing the completion time of a task during processor selection, we instead minimize an *optimistic* estimate of the time it will take to execute the entire DAG once the task has been scheduled.

The majority of methods for static DAG scheduling in heterogeneous computing were originally intended for grid or cluster platforms but are also applicable to multicore and GPU, albeit perhaps with different behavior. Agullo et al. [1] consider how well-suited static scheduling is for CPU-GPU environments in particular. After extensive experimentation on real systems they concluded that static schedules can be stable even when the estimates used to compute them are inaccurate. Furthermore, they found that incorporating static features into dynamic schedules often improves their performance.

The most similar previous work to this is by Shetti, Fahmy and Bretschneider [14], in which they propose the HEFT-NC (*No Cross*) heuristic as an optimization of HEFT for CPU-GPU environments. Although this is also our aim, the new HOFT heuristic differs significantly from HEFT-NC in both the task prioritization and processor selection phases of the algorithm.

3 Simulation model

In this paper we use a simulation model to study the static task scheduling problem for multicore and GPU environments. This is implemented in `Python` and simulates the scheduling of user-defined DAGs on user-defined CPU-GPU platforms. The source code is available at the following Github repository:

`https://github.com/mcsweeney90/heterogeneous_optimistic_finish_time`.

The code used to generate all results presented in this paper is available in the folder `simulator/scripts` so interested researchers may repeat our experiments for themselves. In addition, users may make modifications to the simulator that they believe will more accurately reflect their own target environment.

To build the simulation model we gathered data from a single heterogeneous node of a local computing cluster. This comprises four octacore Intel (Skylake) Xeon Gold 6130 CPUs running at 2.10GHz with 192GB RAM and four Nvidia V100-SXM2-16GB (Volta) GPUs, each with 16GB GPU global memory, 5120 CUDA Cores and NVLink interconnect. We used *Basic Linear Algebra Subroutine* (BLAS) [9] kernels for these data-gathering experiments because they are widely-used in scientific computing applications.

3.1 Mathematical model

The simulator software implements the following mathematical model of the problem. Suppose we have a task DAG G consisting of n tasks and e edges that we wish to execute on a target platform H comprising P processing resources of two types, P_C CPU resources and $P - P_C = P_G$ GPU resources. In keeping with much of the related literature and based on current programming practices, we consider CPU cores individually but regard entire GPUs as discrete [1]. For example, a node comprising 4 GPUs and 4 octacore CPUs would be viewed as 4 GPU resources and $4 \times 8 = 32$ CPU resources.

We assume that all tasks $t_1, \ldots, t_n$ are atomic and cannot be divided across multiple resources or aggregated to form larger tasks. Further, all resources can only execute a single task at any one time and can in principle execute all tasks, albeit with different processing times. In our experiments, we found that the spread of BLAS kernel processing times on CPU cores and GPUs was usually tight, with the standard deviation often being two orders of magnitude smaller than the mean. Thus we assume that all task execution times on all processing resources of a single type are identical. In particular, this means that each task has only two possible *computation costs*: a CPU execution time $w_C(t_i)$ and a GPU execution time $w_G(t_i)$. When necessary, we denote by w_{im} the processing time of task t_i on the specific resource p_m.

The *communication cost* between task t_i and its child t_j is the length of time between when execution of t_i is complete and execution of t_j can begin, including all relevant latency and data transfer times. Since this depends on which processing resource executes each task, we view this as a function $c_{ij}(p_m, p_n)$. Based on our analysis of the time taken from when a specific processing resource completed execution of a task and another was able to begin execution of a child task on our data-gathering node, we assume in our simulation model that there are only four possible communication costs between tasks t_i and t_j: $c_{ij}(C, C)$, from a CPU to a different CPU; $c_{ij}(C, G)$, from CPU to GPU; $c_{ij}(P, C)$, from GPU to CPU; and $c_{ij}(G, G)$ from GPU to a different GPU.

A *schedule* is a mapping from tasks to processing resources, as well as the precise time at which their execution should begin. Our goal is to find a schedule which minimizes the *makespan* of the task graph, the total execution time of the application it represents. Although we assume that all computation and communication costs represent time, they could be any other cost we wish to minimize, such as energy consumption, so long as this is done consistently. We do not however consider the simultaneous optimization of two or more different types of costs. A task with no successors is called an *exit* task. Once all tasks have been scheduled, the makespan is easily computed as the earliest time all exit tasks have been executed.

3.2 Testing environments

In the numerical experiments described later in this article, we consider two simulated target platforms: *Single GPU*, comprising 1 GPU and 1 octacore CPU, and *Multiple GPU*, comprising 4 GPUs and 4 octacore CPUs. The latter represents the node we used to guide the development of our simulator and the former is considered in order to study how the number of GPUs affects performance. We follow the convention that a CPU core is dedicated to managing each of the GPUs [3], so these two platforms are actually assumed to comprise 7 CPU and 1 GPU resources, and 28 CPU and 4 GPU resources, respectively. Based on our exploratory experiments, we make two further assumptions. First, since communication costs between CPU resources were negligible relative to all other combinations, we assume they are zero—i.e., $c_{ij}(C, C) = 0, \forall i, j$. Second, because CPU-GPU communication costs were very similar to the corresponding GPU-CPU and GPU-GPU costs, we take them to be identical—i.e., $c_{ij}(C, G) = c_{ij}(G, C) = c_{ij}(G, G), \forall i, j$. Again, these assumptions will not be representative of all architectures, but the simulator software allows users who wish to repeat our experiments to do so for more accurate representations of their own target platforms.

We consider the scheduling of two different sets of DAGs. The first consists of ten DAGs comprising between 35 and $22,100$ tasks which correspond to the *Cholesky factorization* of $N \times N$ tiled matrices, where $N = 5, 10, 15, \ldots, 50$. In particular, the DAGs are based on a common implementation of Cholesky factorization which uses `GEMM` (matrix multiplication), `POTRF` (Cholesky factorization), `SYRK` (symmetric rank-k update) and `TRSM` (triangular solve) BLAS kernels [9]. All task CPU/GPU processing times are means of 1000 real timings of that task kernel. Likewise, communication costs are sample means of real communication timings between the relevant task and resource types. All numerical experiments were performed for tile sizes 128 and 1024; which was used will always be specified where results are presented. Those sizes were chosen as they roughly mark the upper and lower limits of tile sizes typically used for CPU-GPU platforms.

The standard way to quantify the relative amount of communication and computation represented by a task graph is the *computation-to-communication ratio* (CCR), the mean computation cost of the DAG divided by the mean communication cost. For the Cholesky DAGs, the CCR was about 1 for tile size 128 and about 18 for tile size 1024, with minor variation depending on the total number of tasks in the DAG.

We constructed a set of randomly-generated DAGs with a wide range of CCRs, based on the topologies of the 180 DAGs with 1002 tasks from the *Standard Task Graph* (STG) set [15]. Following the approach in [8], we selected GPU execution times for all tasks uniformly at random from $[1, 100]$ and computed the corresponding CPU times by multiplying by a random variable from a Gamma distribution. To consider a variety of potential applications, for each DAG we made two copies: *low acceleration*, for which with mean and standard deviation of the Gamma distribution is defined to be 5, and *high acceleration*, for which both are taken to be 50 instead. These values roughly correspond to what we observed in our benchmarking of BLAS kernels with tile sizes 128 and 1024, respectively. Finally, for both parameter regimes, we made three copies of each DAG and randomly generated communication costs such that the CCR fell into each of the intervals $[0, 10]$, $[10, 20]$ and $[20, 50]$. Thus in total the random DAG set contains $180 \times 2 \times 3 = 1080$ DAGs.

4 HEFT

Recall that as a listing scheduler HEFT comprises two phases, an initial *task prioritization* phase in which the order all tasks are to be scheduled is determined and a *processor selection* phase in which the processing resource each task is to be scheduled on is decided. Here we describe both in order to give a complete description of the HEFT algorithm. Then in Section 4.1 we benchmark HEFT's performance using our simulation model. Finally, in Section 4.2 we describe a simple modification to the HEFT algorithm that we found to be effective.

The *critical path* of a DAG is the longest path through it, and is important because it gives a lower bound on the optimal schedule makespan for that DAG. Heuristics for homogeneous platforms often use the *upward rank*, the length of the critical path from that task to an exit task, including the task itself [16], to determine priorities. Computing the critical path is not straightforward for heterogeneous platforms so HEFT extends the idea by using *mean* values instead. Intuitively, the task prioritization phase of HEFT can be viewed as an approximate dynamic program applied to a simplified version of the task DAG that uses mean values to set all weights.

More formally, we first define the *mean execution cost* of all tasks t_i through

$$\overline{w_i} := \sum_{m=1}^{P} \frac{w_{im}}{P} = \frac{w_C(t_i)P_C + w_G(t_i)P_G}{P}, \tag{1}$$

where the second expression is how $\overline{w_i}$ would be computed under the assumptions of our model. Likewise, the *mean communication cost* $\overline{c_{ij}}$ between t_i and t_j is then the average of all such costs over all possible combinations of resources,

$$\overline{c_{ij}} = \frac{1}{P^2} \sum_{m,n} c_{ij}(p_m, p_n) = \frac{1}{P^2} \sum_{k,\ell \in \{C,G\}} A_{k\ell} c_{ij}(k, \ell), \tag{2}$$

where $A_{CC} = P_C(P_C - 1)$, $A_{CG} = P_C P_G = A_{GC}$, and $A_{GG} = P_G(P_G - 1)$. For all tasks t_i in the DAG, we define their upward ranks $rank_u(t_i)$ recursively, starting from the exit task(s), by

$$rank_u(t_i) = \overline{w_i} + \max_{t_j \in Ch(t_i)} (\overline{c_{ij}} + rank_u(t_j)), \tag{3}$$

where $Ch(t_i)$ is the set of t_i's immediate successors in the DAG. The task prioritization phase then concludes by listing all tasks in decreasing order of upward rank, with ties broken arbitrarily.

The processor selection phase of HEFT is now straightforward: we move down the list and assign each task to the resource expected to complete its execution at the earliest time. Let R_{m_i} be the earliest time at which the processing resource p_m is actually free to execute task t_i, $Pa(t_i)$ be the set of t_i's immediate predecessors in the DAG, and $AFT(t_k)$ be the time when execution of a task t_k is actually completed (which in the static case is known precisely once it has been scheduled). Then the *earliest start time* of task t_i on processing resource p_m is computed through

$$EST(t_i, p_m) = \max \left\{ R_{m_i}, \max_{t_k \in Pa(t_i)} (AFT(t_k) + c_{ki}(p_k, p_m)) \right\} \tag{4}$$

and the *earliest finish time* $EFT(t_i, p_m)$ of task t_i on p_m is given by

$$EFT(t_i, p_m) = w_{im} + EST(t_i, p_m). \tag{5}$$

HEFT follows an *insertion-based* policy that allows tasks to be inserted between two that are already scheduled, assuming precedence constraints are still respected, so R_{m_i} may not simply be the latest finish time of all tasks on p_m. A complete description of HEFT is given in Algorithm 1. HEFT has a time complexity of $O(P \cdot e)$. For dense DAGs, the number of edges is proportional to n^2, where n is the number of tasks, so the complexity is effectively $O(n^2 P)$ [16].

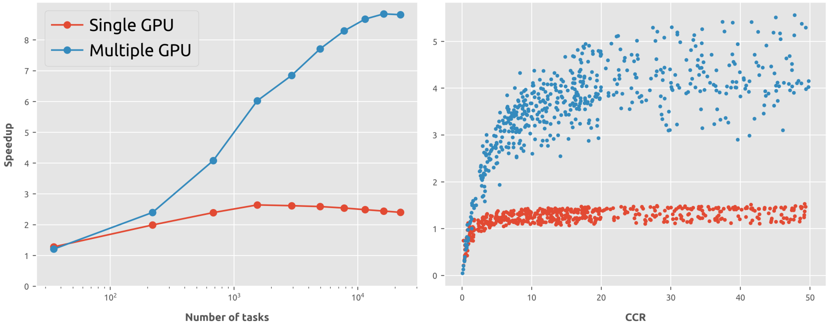
Algorithm 1: HEFT.

```
Set the computation cost of all tasks using (1)
Set the communication cost of all edges using (2)
Compute $rank_u$ for all tasks according to (3)
Sort the tasks into a priority list by non-increasing order of $rank_u$
for task in list do
    for each resource $p_k$ do
        Compute $EFT(t_i, p_k)$ using (4) and (5)
    end
    $p_m := \arg\min_k(EFT(t_i, p_k))$
    Schedule $t_i$ on the resource $p_m$
end
```

4.1 Benchmarking

Using our simulation model, we investigated the quality of the schedules computed by HEFT for the Cholesky and randomly-generated DAG sets on both the single and multiple GPU target platforms described in Section 3. The metric used for evaluation was the *speedup*, the ratio of the *minimal serial time* (MST)—the minimum execution time of the DAG on any single resource—to the makespan. Intuitively, speedup tells us how well the schedule exploits the parallelism of the target platform.

Figure 1a shows the speedup of HEFT for the Cholesky DAGs with tile size 128. The most interesting takeaway is the difference between the two platforms. With multiple GPUs the speedup increased uniformly with the number of tasks until a small decline for the very largest DAG, but for a single GPU the speedup stagnated much more quickly. This was due to the GPU being continuously busy and adding little additional value once the DAGs became sufficiently large and more GPUs therefore postponing this effect. Results were broadly similar for Cholesky DAGs with tile size 1024, with the exception that the speedup values were uniformly smaller, reaching a maximum of just over four for the multiple GPU platform. This is because the GPUs are so much faster for the larger tiles that HEFT makes little use of the CPUs and the speedup is almost entirely determined by the number of GPUs available.



(a) Cholesky. (b) Random.

Fig. 1: Speedup of HEFT for Cholesky (tile size 128) and random (high acceleration) DAG sets.

Figure 1b shows the speedups for all 540 high acceleration randomly-generated DAGs, ordered by their CCRs; results were broadly similar for the low acceleration DAGs. The speedups for the single GPU platform are much smaller with a narrower spread compared to the other platform, as for the Cholesky DAGs. More surprising is that HEFT sometimes returned a schedule with speedup less than one for DAGs with small CCR values, which we call a *failure* since this is obviously unwanted behavior. These failures were due to the greediness of the HEFT processor selection phase, which always schedules a task on the processing resource that minimizes its earliest finish time without considering the communication costs that may later be incurred by doing so. The effect is more pronounced when the CCR is low because the unforeseen communication costs are proportionally larger. A likely candidate for dealing with such cases would seem to be HEFT with lookahead [4], but the additional computational cost of the modified algorithm proved impractical for our purposes. We investigated cheaper *sampling-based* selection phases that consider only a small selection of the child tasks—selected either at random or based on priorities—during the lookahead. These did reduce the failure probability slightly for DAGs with low CCR values but the improvement was only minor. An alternative to lookahead that may be useful is duplication-based heuristics [12].

4.2 HEFT-WM

The implicit assumption underlying the use of mean values when computing task priorities in HEFT is that the probability of a task being scheduled on a processing resource is identical for all resources. But this is obviously not the case: if a task's GPU execution time is ten times smaller than its CPU execution time then it is considerably more likely to be scheduled on a GPU, even if not ten times so because of resource contention. This suggests that we should weight the mean values according to each task's *acceleration ratio* r_i, the CPU time divided by the GPU time. In particular, for each task t_i we estimate its computation cost to be

$$\overline{w_i} = \frac{w_C(t_i)P_C + r_i w_G(t_i)P_G}{P_C + r_i P_G}, \tag{6}$$

and for each edge (say, between tasks t_i and t_j) the estimated communication cost is

$$\overline{c_{ij}} = \frac{A_{CC} \cdot c_{ij}(C,C) + A_{CG}\big(r_i c_{ij}(G,C) + r_j c_{ij}(C,G)\big) + r_i r_j A_{GG} \cdot c_{ij}(G,G)}{(r_i P_G + P_C) \cdot (r_j P_G + P_C)}. \tag{7}$$

We call the modified heuristic which follows Algorithm 1 but uses (6) instead of (1) and (7) instead of (2) HEFT-WM (for *Weighted Mean*). We were unable to find any explicit references to this modification in the literature but given its simplicity we do not think it is a novel idea and suspect it has surely been used before in practice.

5 HOFT

If we disregard all resource contention, then we can easily compute the earliest possible time that all tasks can be completed assuming that they are scheduled on either a CPU or GPU resource, which we call the *Optimistic Finish Time* (OFT). More specifically, for $p, p' \in \{C, G\}$, we move forward through the DAG and build a table of OFT values by setting $OFT(t_i, p) = w_p(t_i)$, if t_i is an entry task, and recursively computing

$$OFT(t_i, p) = w_p(t_i) + \max_{t_j \in Pa(t_i)} \left\{ \min_{p'} \{OFT(t_j, p') + \delta_{pp'} c_{ij}(p, p')\} \right\}, \tag{8}$$

for all other tasks, where $\delta_{pp'} = 1$ if $p = p'$ and 0 otherwise. We use the OFT table as the basis for the task prioritization and processor selection phases of a new HEFT-like heuristic optimized for CPU-GPU platforms that we call *Heterogeneous Optimistic Finish Time* (HOFT). Note that computing the OFT table does not increase the order of HEFT's time complexity.

Among several possible ways of using the OFT to compute a complete task prioritization, we found the most effective to be the following. First, define the weights of all tasks to be the ratio of the maximum and minimum OFT values,

$$\overline{w_i} = \frac{\max\{OFT(t_i, C), OFT(t_i, G)\}}{\min\{OFT(t_i, C), OFT(t_i, G)\}}. \tag{9}$$

Now assume that all edge weights are zero, $\overline{c_{ij}} \equiv 0, \forall i, j$, and compute the upward rank of all tasks with these values. Upward ranking is used to ensure that all precedence constraints are met. Intuitively, tasks with a strong preference for one resource type—as suggested by a high ratio—should be scheduled first.

We also propose a new processor selection phase which proceeds as follows. Working down the priority list, each task t_i is scheduled on the processing resource p_m with the smallest EFT as in HEFT except when p_m is not also the fastest resource type for that task. In such cases, let p_f be the resource of the fastest type with the minimal EFT and compute

$$s_m := EFT(t_i, p_f) - EFT(t_i, p_m), \tag{10}$$

the saving that we expect to make by scheduling t_i on p_m rather than p_f. Suppose that p_m is of type $T_m \in \{C, G\}$. By assuming that each child task t_j of t_i is scheduled on the type of resource T_j which minimizes its OFT, we define $E(Ch(t_i)|p_m)$ to be an optimistic estimate of the earliest finish time of all child tasks assuming that t_i is scheduled on p_m. In particular, we compute

$$E(Ch(t_i)|p_m) := \max_{t_j \in Ch(t_i)} \Big(EFT(t_i, p_m) + c_{ij}(T_m, T_j) + w_{T_j}(t_j) \Big). \tag{11}$$

Likewise for p_f we compute $E(Ch(t_i)|p_f)$. As in HEFT with lookahead, we call the estimates *optimistic* since they disregard the possible need to wait for other parent tasks to complete. Finally, if

$$s_m > E(Ch(t_i)|p_m) - E(Ch(t_i)|p_f) \tag{12}$$

we schedule task t_i on p_m; otherwise, we schedule it on p_f. Intuitively, the processor selection always chooses the resource with the smallest EFT unless by doing so we expect to increase the earliest possible time at which all child tasks can be completed.

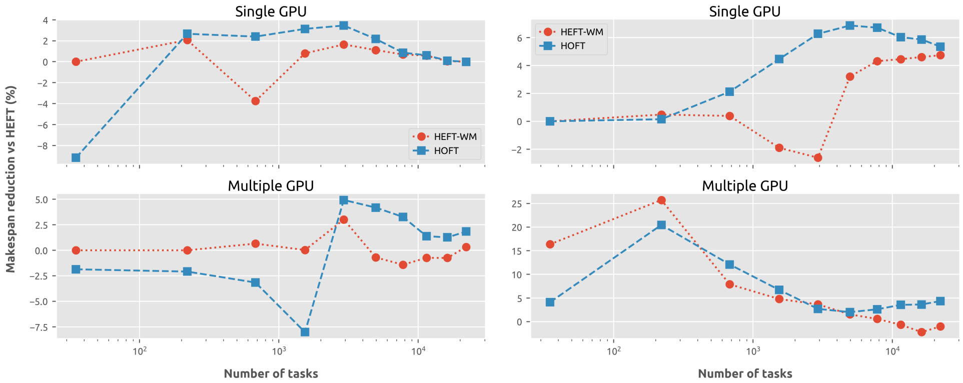
6 Simulation results

Figure 2 shows the reduction in schedule makespan, as a percentage of the HEFT schedule, achieved by HOFT and HEFT-WM for the set of Cholesky DAGs, on both the single and multiple GPU target platforms. The overall trend for the multiple GPU platform is that HOFT improves relative to both HEFT variants as the number of tasks in the DAG grows larger; it is always better than standard HEFT for tile size 1024. For the single GPU platform, HOFT was almost always the best, except for the smallest DAGs and the largest DAG with tile size 128 (for which all three heuristics were almost identical). Interestingly, we found that the processor selection phase of HOFT never actually differed from HEFT's for these DAGs and so the task prioritization phase alone was key.

The relative performance of the heuristics was rather variable for the randomly-generated DAGs, depending on the target platform and acceleration regime. However both HOFT and HEFT-WM achieved smaller makespans than HEFT on average. Table 1 summarizes the results... Also included in the table is HOFT-WM, the heuristic obtained by using the HEFT-WM task prioritization phase with the HOFT processor selection.

Algorithm 2: HOFT.

```
Compute the OFT table for all tasks using (8)
Set the computation cost of all tasks using (9)
Set the communication cost of all edges to be zero
Compute $rank_u$ for all tasks according to (3)
Sort the tasks into a priority list by non-increasing order of $rank_u$
for task in list do
    for each resource $p_k$ do
        Compute $EFT(t_i, p_k)$ using (4) and (5)
    end
    $p_m := \arg\min_k(EFT(t_i, p_k))$
    if $w_{im} \neq \min(w_C(t_i), w_G(t_i))$ then
        $p_f := \arg\min_k \left(EFT(t_i, p_k) | w_{ik} = \min(w_C(t_i), w_G(t_i))\right)$
        Compute $s_m$ using (10).
        Compute $E(Ch(t_i)|p_m)$ and $E(Ch(t_i)|p_f)$ using (11)
        if (12) holds then
            Schedule $t_i$ on $p_m$
        end
        else
            Schedule $t_i$ on $p_f$
        end
    end
end
```



(a) Tile size 128.

(b) Tile size 1024.

Fig. 2: HEFT-WM and HOFT compared to HEFT for Cholesky DAGs.

Table 1: Average makespan reduction (%) vs. HEFT.

	1 GPU, 1 CPU		4 GPUs, 4 CPUs	
Heuristic	Low acc.	High acc.	Low acc.	High acc.
HEFT-WM	0.8	2.3	1.6	2.4
HOFT	−0.2	3.8	1.4	2.3
HOFT-WM	0.8	4.6	1.4	3.7

7 Conclusions

Taking the Cholesky and random DAG sets as a whole, HOFT improved on HEFT by an average of about 3% on both the single and multiple GPU target platforms. For Cholesky DAGs in particular, makespan reductions for DAGs with more than 2000 tasks were often even higher. In those cases, the reductions were entirely due to the alternative task prioritization phase, but for random DAGs the HOFT processor selection phase proved to be useful. Overall, the safest conclusion from our simulations is that HOFT appears to be at least competitive with—and often superior to—both standard HEFT and HEFT-WM for both single and multiple GPU target platforms. It should be noted that HEFT-WM also performed very well in our simulations and was almost invariably superior to the standard algorithm, suggesting that it should perhaps be the default in accelerated environments.

As stated in Section 1, static schedules for multicore and GPU are most useful in practice as a foundation or guide for efficient dynamic schedules; this paper was concerned only with how such static schedules may be computed in the first place. Extending static scheduling to dynamic environments is often called *stochastic* scheduling, since the costs are usually modeled as random variables from some possibly unknown distribution. The goal is typically to find methods that bound variation in the schedule makespan, which is notoriously difficult given that Hagstrom [11] showed that even computing its probability distribution is a $\#P$-complete problem. Small-scale experimentation with existing stochastic scheduling heuristics, such as *Monte Carlo Scheduling* [17], suggested to us that the main issue with adapting such methods for multicore and GPU is their cost, which can be much greater than computing the initial static schedule. Hence in future work we intend to investigate cheaper ways to make effective use of static schedules for real CPU-GPU platforms.

References

1. Agullo, E., Beaumont, O., Eyraud-Dubois, L., Kumar, S.: Are static schedules so bad? A case study on Cholesky factorization. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 1021–1030 (May 2016). https://doi.org/10.1109/IPDPS.2016.90

2. Arabnejad, H., Barbosa, J.G.: List scheduling algorithm for heterogeneous systems by an optimistic cost table. IEEE Transactions on Parallel and Distributed Systems **25**(3), 682–694 (March 2014). https://doi.org/10.1109/TPDS.2013.57
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011). https://doi.org/10.1002/cpe.1631
4. Bittencourt, L.F., Sakellariou, R., Madeira, E.R.M.: DAG scheduling using a lookahead variant of the Heterogeneous Earliest Finish Time algorithm. In: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. pp. 27–34 (Feb 2010). https://doi.org/10.1109/PDP.2010.56
5. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J.J.: PaRSEC: Exploiting heterogeneity to enhance scalability. Computing in Science & Engineering **15**(6), 36–45 (2013). https://doi.org/10.1109/MCSE.2013.98
6. Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., Freund, R.F.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. Journal of Parallel and Distributed Computing **61**(6), 810 – 837 (2001). https://doi.org/10.1006/jpdc.2000.1714
7. Canon, L.C., Jeannot, E., Sakellariou, R., Zheng, W.: Comparative evaluation of the robustness of DAG scheduling heuristics. In: Grid Computing. pp. 73–84. Springer (2008). https://doi.org/10.1007/978-0-387-09457-1_7
8. Canon, L.C., Marchal, L., Simon, B., Vivien, F.: Online scheduling of task graphs on hybrid platforms. In: European Conference on Parallel Processing. pp. 192–204. Springer (2018). https://doi.org/10.1007/978-3-319-96983-1_14
9. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Softw. **16**(1), 1–17 (Mar 1990). https://doi.org/10.1145/77626.79170
10. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel processing letters **21**(02), 173–193 (2011). https://doi.org/10.1142/S0129626411000151
11. Hagstrom, J.N.: Computational complexity of PERT problems. Networks **18**(2), 139–147. https://doi.org/10.1002/net.3230180206
12. Ishfaq Ahmad, Yu-Kwong Kwok: On exploiting task duplication in parallel program scheduling. IEEE Transactions on Parallel and Distributed Systems **9**(9), 872–892 (Sep 1998). https://doi.org/10.1109/71.722221
13. Kumar, S.: Scheduling of Dense Linear Algebra Kernels on Heterogeneous Resources. PhD thesis, University of Bordeaux (Apr 2017)
14. Shetti, K.R., Fahmy, S.A., Bretschneider, T.: Optimization of the HEFT algorithm for a CPU-GPU environment. In: PDCAT. pp. 212–218 (2013). https://doi.org/10.1109/PDCAT.2013.40
15. Tobita, T., Kasahara, H.: A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. Journal of Scheduling **5**(5), 379–394. https://doi.org/10.1002/jos.116
16. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE transactions on parallel and distributed systems **13**(3), 260–274 (2002). https://doi.org/10.1109/71.993206
17. Zheng, W., Sakellariou, R.: Stochastic DAG scheduling using a Monte Carlo approach. Journal of Parallel and Distributed Computing **73**(12), 1673 – 1689 (2013). https://doi.org/10.1016/j.jpdc.2013.07.019