MANCHESTER 1824
The University of Manchester

MCrNA
Manchester Numerical Analysis

An efficient new static scheduling heuristic for accelerated architectures

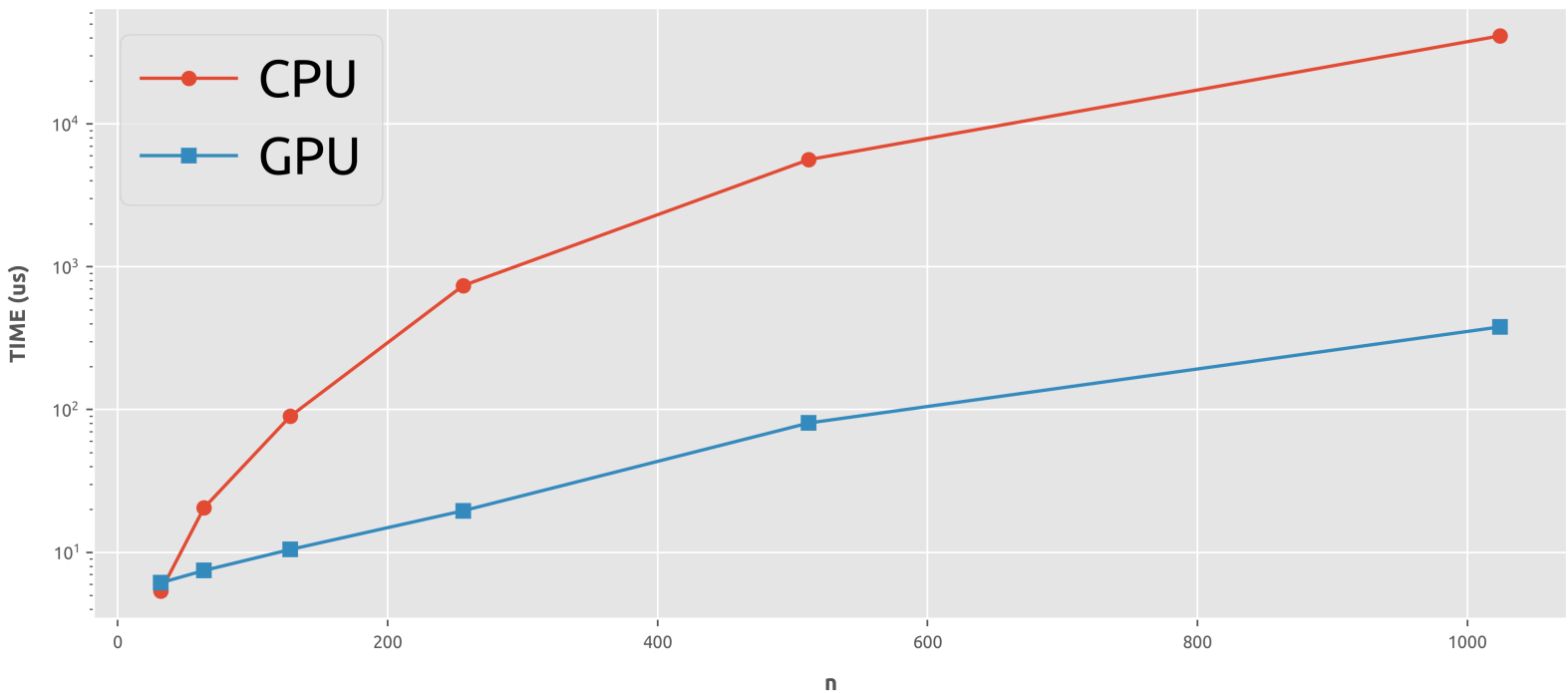
Tom McSweeney
Alan Turing Building, 2.111
thomas.mcsweeney@postgrad.manchester.ac.uk

NLA Group meeting
November 28, 2019

Rise of the GPU

Using **Graphics Processing Units** (**GPUs**) for general computations—in addition to **multicore CPUs**—is increasingly common in **high-performance computing** (**HPC**).

GPUs are particularly effective for **numerical linear algebra** applications, such as **matrix multiplication**.

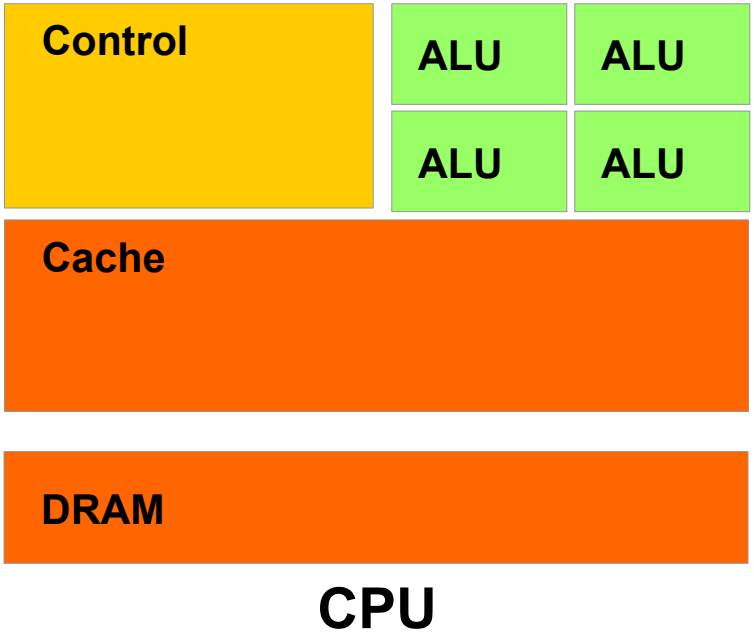


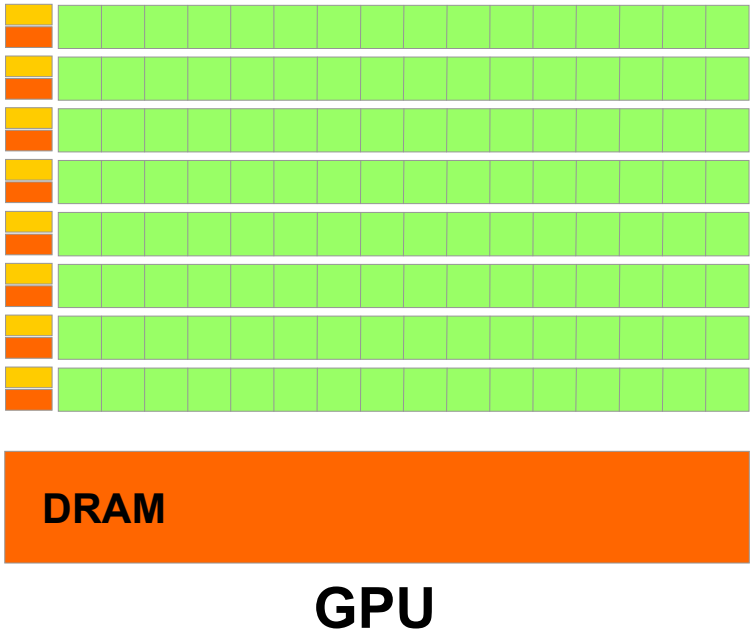
Summit

World's fastest supercomputer*, it has 4,608 **nodes** each with:

- 2 Power9 22-core **CPUs**,
- 6 NVIDIA Tesla V100 **GPUs**.

How do we take advantage?





Task-based programming

Divide application up into a set of **tasks**. Do the GPU-friendly ones on GPU and the CPU-friendly ones on CPU. But what about the **precedence constraints**?

From tasks to DAGs

Problem

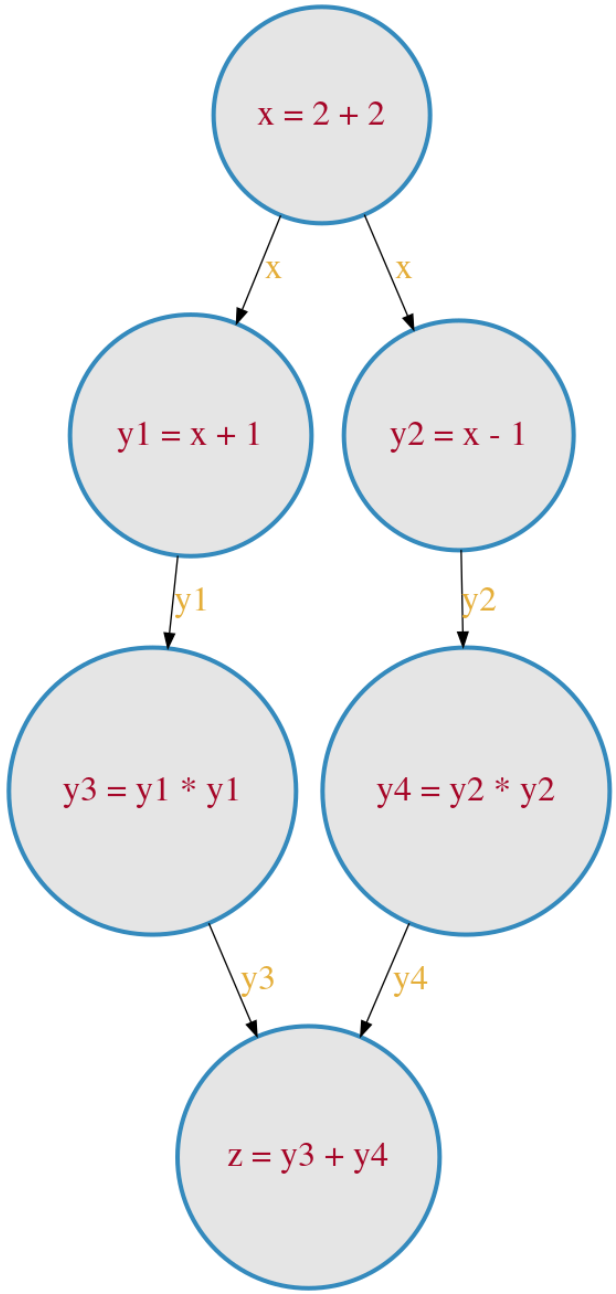
Compute $z = \|y\|_2^2$ for

$$y = \begin{bmatrix} x + 1 \\ x - 1 \end{bmatrix}$$

and

$$x = 2 + 2.$$

Only interested in directed graphs without cycles—**DAGs**.



Example: Cholesky factorization

$$A = \begin{bmatrix} A_{00} & \dots & A_{N0} \\ \vdots & \ddots & \vdots \\ A_{N0} & \dots & A_{NN} \end{bmatrix}.$$

Implementation

For $k = 0, 1, \dots, N$:

$A_{kk} = \mathbf{POTRF}(A_{kk})$

For $m = k+1, \dots, N-1$:

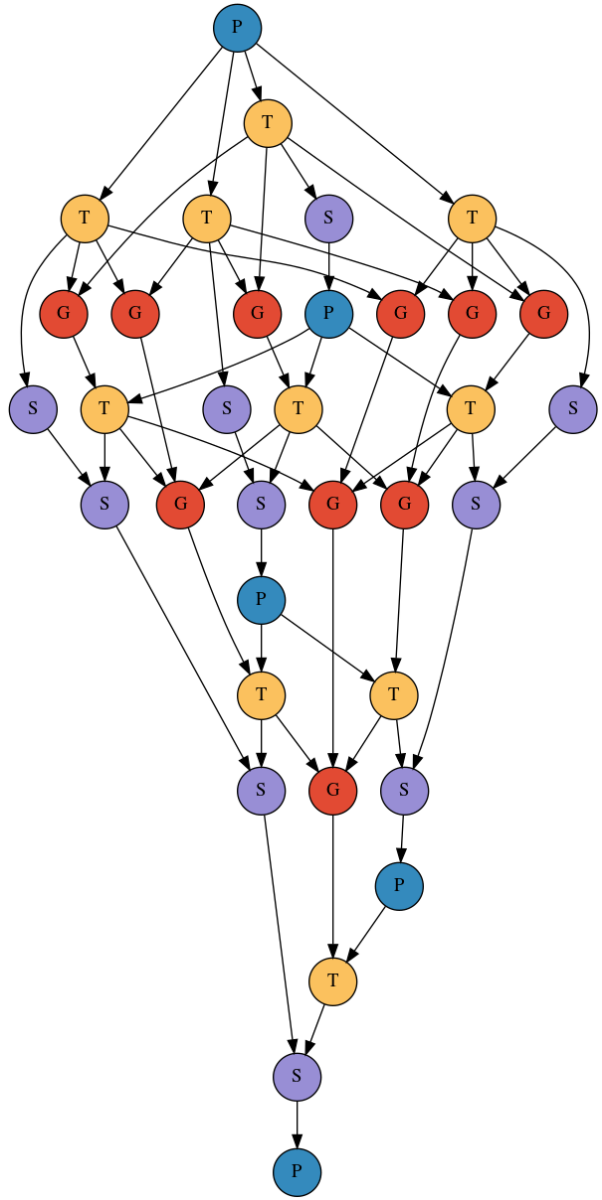
$A_{mk} = \mathbf{TRSM}(A_{kk}, A_{mk})$

For $n = k+1, \dots, N-1$:

$A_{nn} = \mathbf{SYRK}(A_{nk}, A_{nn})$

For $m = n+1, \dots, N-1$:

$A_{mn} = \mathbf{GEMM}(A_{mk}, A_{nk}, A_{mn})$



The task scheduling problem

We want to find the optimal **schedule** that tells us which tasks will be done by each processor, in what order, and (ideally) at what time, so the **makespan** is minimized.

Unfortunately this is an **NP-complete** problem
$\Longrightarrow$ (probably) can't find a truly optimal schedule.

Note

In practice, scheduling is normally handled by a **runtime system**. Examples include:

- **StarPU** [Augonnet et al., Inria, 2010–],
- **PaRSEC** [Bosilca et al., ICL, 2013–],
- **OmpSs** [Duran et al., Barcelona, 2011–].

Static and dynamic scheduling

Computation and communication times are **never** known precisely before the DAG is actually executed.

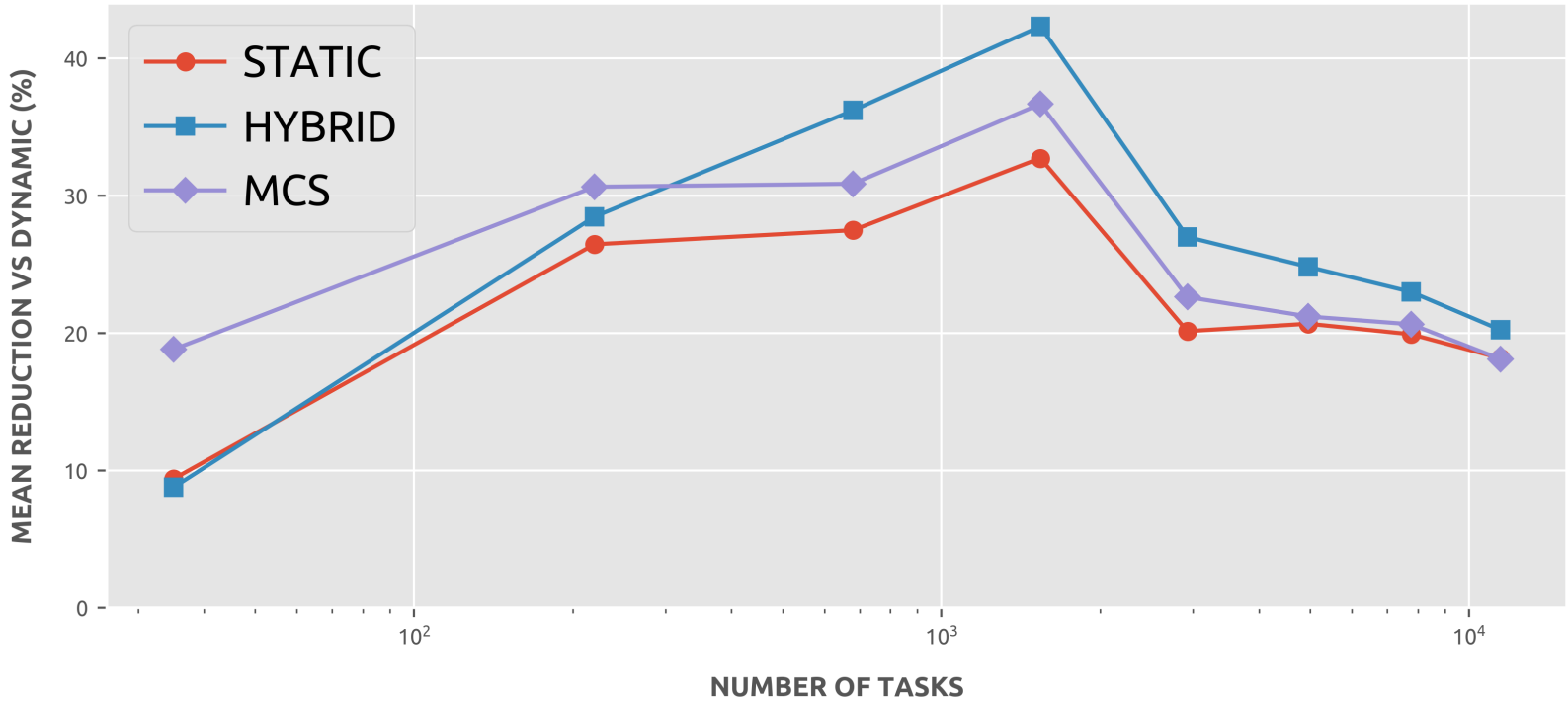
- **Static** scheduling.
  Compute schedule before runtime based on best estimates available, or probability distributions of timings.

- **Dynamic** scheduling.
  Determine schedule at runtime using the latest data.

Conventional wisdom

Static scheduling better when estimates are good but otherwise use dynamic.

Static scheduling for multicore and GPU?



- Static schedules can be **robust** in practice. In addition, using static schedule as a **guide** better than wholly dynamic [Agullo et al., 2016].
- Can use timing **distribution information** to make schedule even more robust [Zheng and Sakellariou, 2013].

The plan

1. Find a good static schedule **quickly** for a given DAG and CPU-GPU **target platform**.
2. If necessary, use static schedule as a **basis** for better dynamic schedule at runtime, or do some **post-processing** using distribution information to make it more robust.

This talk is focused only on the **first part**.

Model and assumptions

We have a DAG G consisting of n tasks and e edges and a target platform with P processors, P_C CPUs and $P - P_C = P_G$ GPUs.
– View CPU **cores** individually but GPUs as **discrete**.

Assume that:

1. All tasks $t_1, \ldots, t_n$ are **atomic**.
2. All processors can only execute a single task **at once**.
3. All processors can do all tasks, but with **different times**.
   – Task t_i has CPU time $w_C(t_i)$ and GPU time $w_G(t_i)$.
   – Task t_i takes time w_{im} on processor p_m.

Model and assumptions II

Communication time between tasks t_i and t_j is time between when t_i is completed and execution of t_j can begin, including all relevant **latency** and **data transfer** times.
– Function $c_{ij}(p_m, p_n)$ of processors.

Assume only **five** possible communication costs:

- **0**, from processor to itself;
- $c_{ij}(C, C)$, from CPU to another CPU;
- $c_{ij}(C, G)$, from CPU to GPU;
- $c_{ij}(P, C)$, from GPU to CPU;
- $c_{ij}(G, G)$, from GPU to another GPU.

(Usually also assume $c_{ij}(C, C) \equiv 0$ and other costs are identical to each other, $c_{ij}(C, G) = c_{ij}(P, C) = c_{ij}(G, G)$, for results, but this depends on memory architecture of target platform!)

Simulation model

Impractical to use real machines for testing, so we use a **simulation model** which implements the mathematical model.

This allows users to simulate the scheduling of **arbitrary** DAGs on **arbitrary** multicore CPU-GPU platforms.

Written in `Python` and available at my Github (mcsweeney90).

```
1 class Task:
2   def __init__(self, task_type=None):
3     self.type = task_type
4     self.ID = None
5     self.entry = False
6     self.exit = False
7     ...
```

Calibrating the simulator

As a guide, we used a single node of the UoM **CSF3** with four **Xeon Gold 6130** CPUs @ 2.10GHz and four Nvidia **V100-SXM2-16GB (Volta)** GPUs.

BLAS kernels (**MKL**/**cuBLAS**) were used for benchmarking.

Table: Mean CPU time / mean GPU time (1000 runs).

Tile size	DGEMM	DPOTRF	DSYRK	DTRSM
64	2.5	0.6	0.8	1.0
128	8.0	1.7	2.3	1.7
256	35.0	1.7	8.6	4.2
512	64.6	5.0	27.3	12.8
1024	92.7	13.7	55.8	24.2

HEFT

The most popular approaches to scheduling in heterogeneous environments are **heuristics**: methods that do well in practice but offer no performance guarantees.

The most prominent is probably **Heterogeneous Earliest Finish Time** (**HEFT**) [Topcuoglu, Hariri and Wu, 2002]. As a **listing** heuristic, it consists of two phases:

1. **Task prioritization**.
   Determine the order tasks are to be scheduled.
2. **Processor selection**.
   Assign tasks to the processors.

Note

HEFT was intended for **clusters** with **diversely heterogeneous** nodes, not just CPU and GPU!

Task prioritization I

Define the **average computation cost** $\overline{w_i}$ of all tasks t_i over all processors

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

The **average communication cost** $\overline{c_{ij}}$ between t_i and t_j is the average over all combinations,

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

where $A_{CC} = P_C(P_C - 1)$, $A_{CG} = P_C P_G = A_{GC}$, and $A_{GG} = P_G(P_G - 1)$.

Task prioritization II

Let:

- $C(t_i)$ be the **children** of task t_i,
- $P(t_i)$ be the **parents** of task t_i.

For all tasks, compute the **upward rank** r_u recursively, starting from the **exit** task, by

$$r_u(t_i) = \overline{w_i} + \max_{t_j \in C(t_i)} (\overline{c_{ij}} + r_u(t_j)). \tag{3}$$

This ensures precedence constraints are met!

Finally, make a scheduling list of all tasks in **decreasing** order of upward rank.

Processor selection

Let:

- F_{m_i} be the earliest time p_m is **free** to do task t_i,
- $AFT(t_k)$ be the **actual** finish time of t_k.

Note F_{m_i} may **not** be the latest finish time of all tasks on p_m!

Then the **earliest start time** of t_i on p_m is

$$EST(t_i, p_m) = \max\left\{F_{m_i}, \max_{t_k \in P(t_i)}(AFT(t_k) + c_{ki}(p_k, p_m))\right\}$$

and the **earliest finish time** of t_i on p_m is

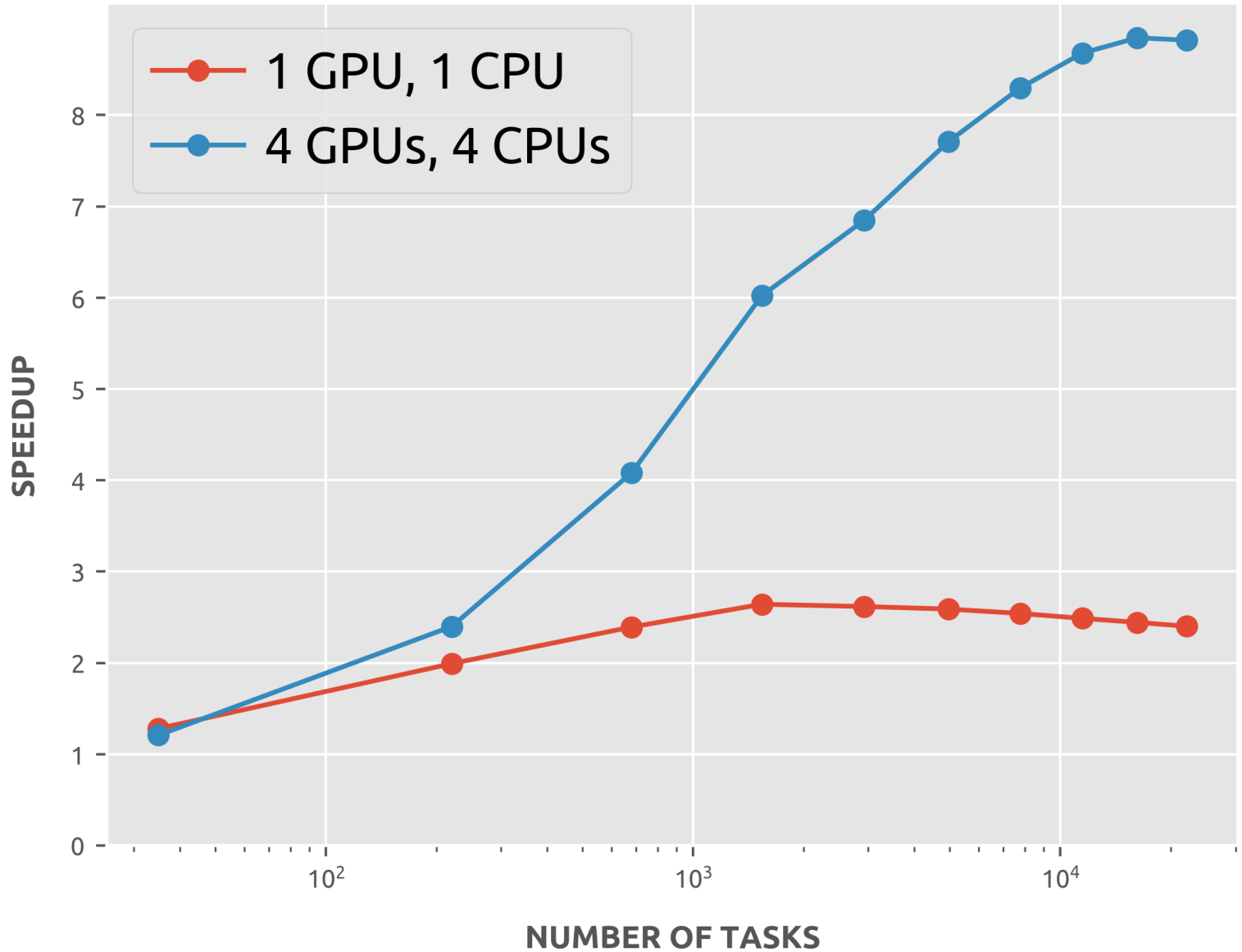
$$EFT(t_i, p_m) = w_{im} + EST(t_i, p_m). \tag{4}$$

HEFT algorithm

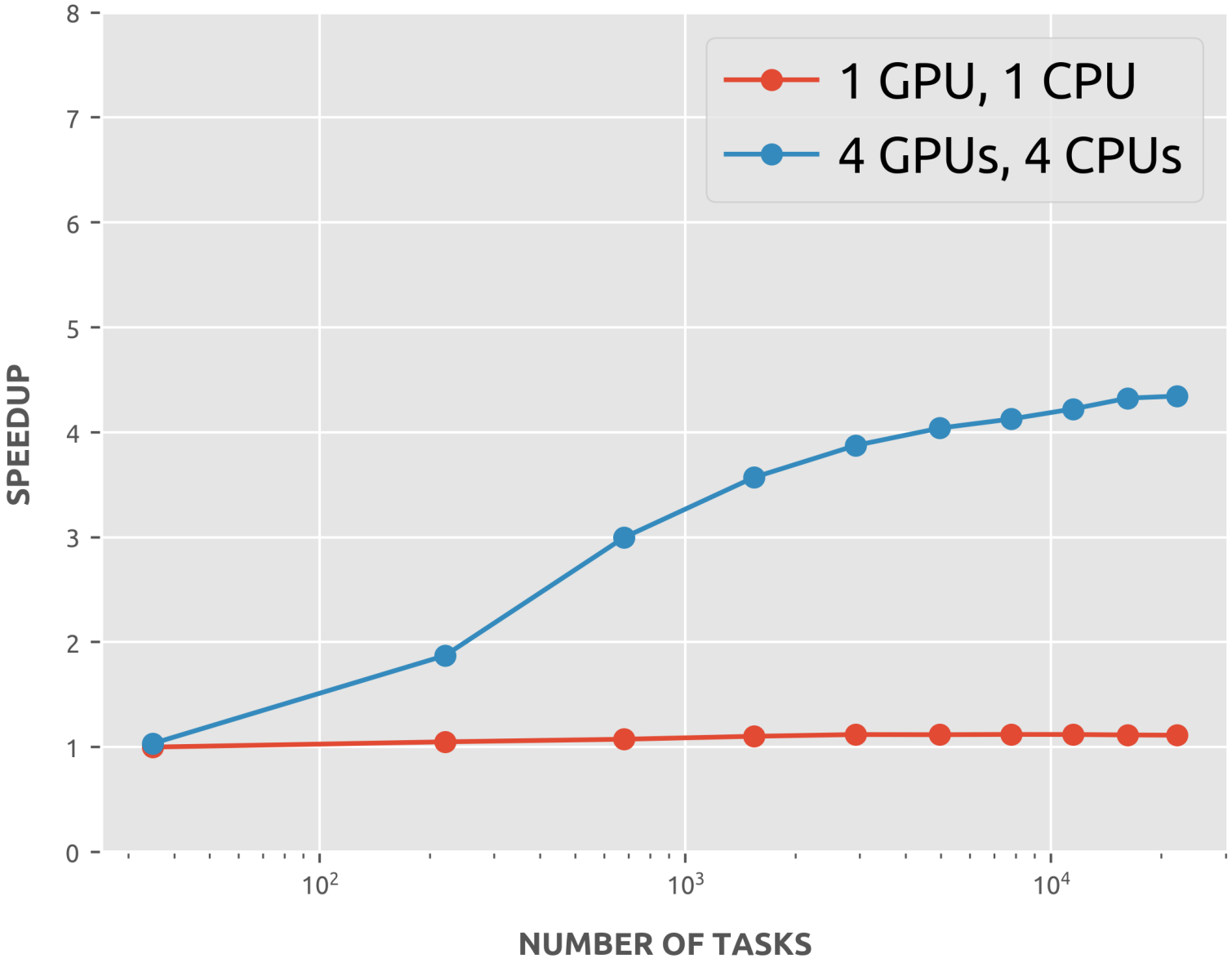
1. Set the computation cost of all tasks using (1)
2. Set the communication cost of all edges using (2)
3. Moving up the DAG, compute r_u for all tasks using (3)
4. Sort tasks into list by decreasing r_u
5. For each task t_i in list:
   For each processor p_k:
   Compute $EFT(t_i, p_k)$ using (4)
   Schedule t_i on $p_m := \arg\min_p EFT(t_i, p)$

HEFT has **time complexity** $O(e \cdot P)$. Usually $e \propto n^2$ so the complexity is $O(n^2 P)$.

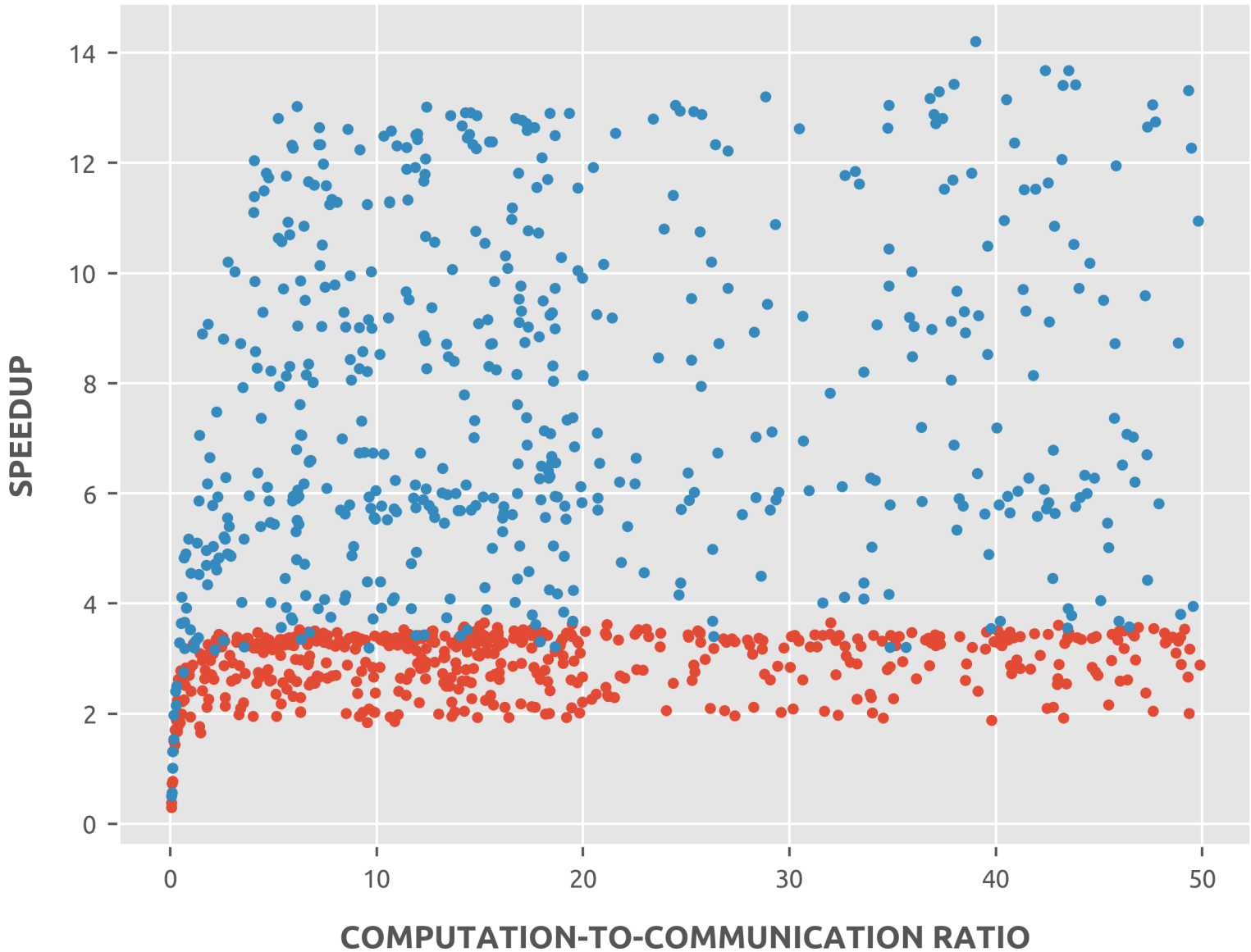
Benchmarking—Cholesky, tile size 128



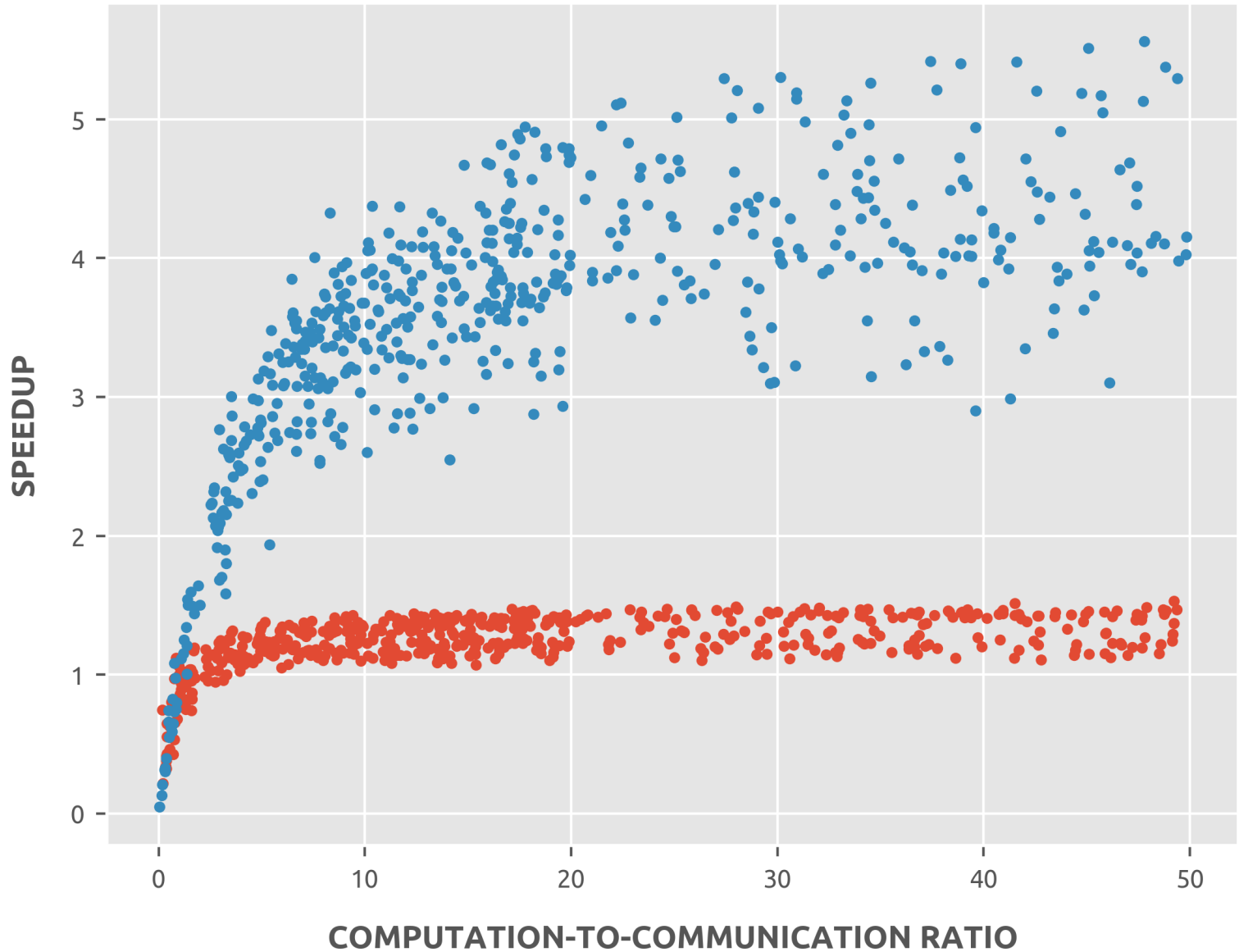
Benchmarking—Cholesky, tile size 1024



Benchmarking—Random, low acceleration



Benchmarking—Random, high acceleration



Possible optimization

Idea

Weight mean values by relative processor power.

Define **acceleration ratio** r_i as CPU time/GPU time.

Use

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

instead of (1) and

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

instead of (2).

We call this **HEFT-WM**.

HOFT

We propose a new heuristic which exploits platforms with only CPUs and GPUs called **Heterogeneous Optimistic Finish Time** (**HOFT**).

Key idea is to compute a **table** of **optimistic** times that all tasks can be completed on both processor types. For $p \in \{C, G\}$, move **forward** through the DAG and **recursively** compute

$$OFT(t_i, p) = w_p(t_i) + \max_{t_j \in P(t_i)} \left\{ \min_{p'} \{ OFT(t_j, p') + \delta_{pp'} c_{ij}(p, p') \} \right\}. \tag{5}$$

This table is the basis for new task prioritization and processor selection phases.

Task prioritization

Define weights of all tasks to be the **ratio** of the maximum and minimum OFT values,

$$\overline{w_i} = \frac{\max\{OFT(t_i, C), OFT(t_i, G)\}}{\min\{OFT(t_i, C), OFT(t_i, G)\}}, \tag{6}$$

and assume all edge weights are zero, $\overline{c_{ij}} \equiv 0$.

Compute the upward rank of all tasks with these values to ensure precedence constraints are met.

Intuition

High ratio means task has a strong **preference** for one resource type and these should be scheduled first. OFT uses more info from the DAG than e.g., acceleration ratio.

Processor selection I

Let $T_k \in \{C, G\}$ be the type of processor p_k.

If $p_m := \arg\min_p EFT(t_i, p)$ and T_m is the **fastest** processor type for t_i, schedule it on p_m.

If not, let p_f be the specific processor of fastest type with minimal EFT. Compute

$$s_m := EFT(t_i, p_f) - EFT(t_i, p_m). \tag{7}$$

Assume that all children of t_i will be scheduled on processor of type that minimizes their OFT. Compute

$$E(C(t_i)|p_m) := \max_{t_j \in C(t_i)} \Big(EFT(t_i, p_m) + c_{ij}(T_m, T_j) + w_{T_j}(t_j) \Big). \tag{8}$$

Processor selection II

Likewise for p_f compute $E(C(t_i)|p_f)$.

If

$$s_m > E(C(t_i)|p_m) - E(C(t_i)|p_f) \tag{9}$$

schedule task t_i on p_m. Else, schedule it on p_f.

Intuition

Choose processor with smallest EFT as in HEFT unless we **expect** to increase the earliest possible time all children can be completed by doing so.

HOFT algorithm

1. Compute the OFT table using (5)
2. Set the computation cost of all tasks using (6)
3. Set the communication cost of all edges to 0
4. Moving up the DAG, compute r_u for all tasks using (3)
5. Sort tasks into list by decreasing r_u
6. For each task t_i in list:

   $p_m := \arg\min_k EFT(t_i, p_k)$

   $p_f := \arg\min_k \left(EFT(t_i, p_k) | w_{ik} = \min(w_C(t_i), w_G(t_i))\right)$

   If $p_m = p_f$:

     Schedule t_i on p_m

   Else:

     Compute s_m using (7)

     Compute $E(C(t_i)|p_m)$ and $E(C(t_i)|p_f)$ using (8)

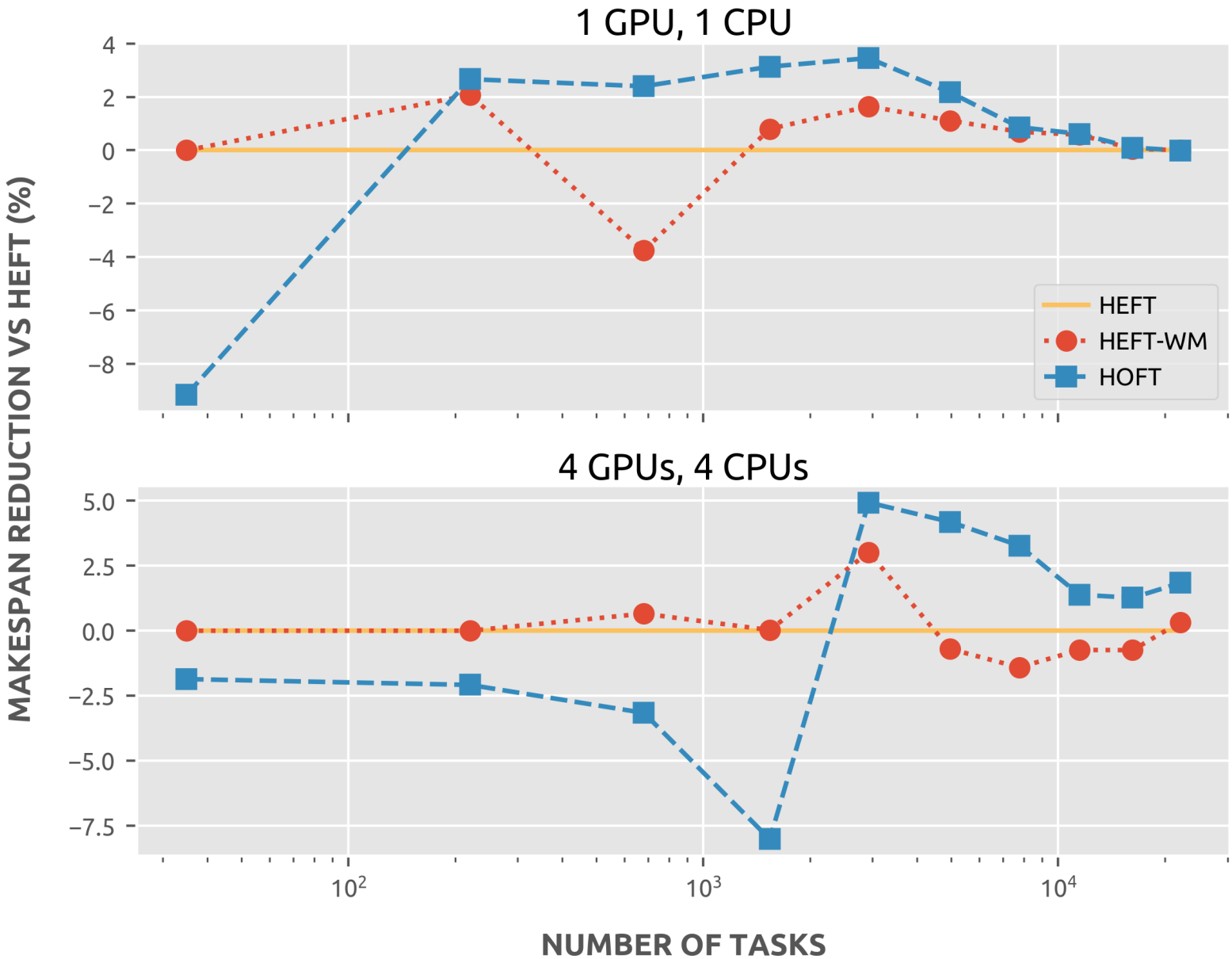
     If (9) holds:

       Schedule t_i on p_m

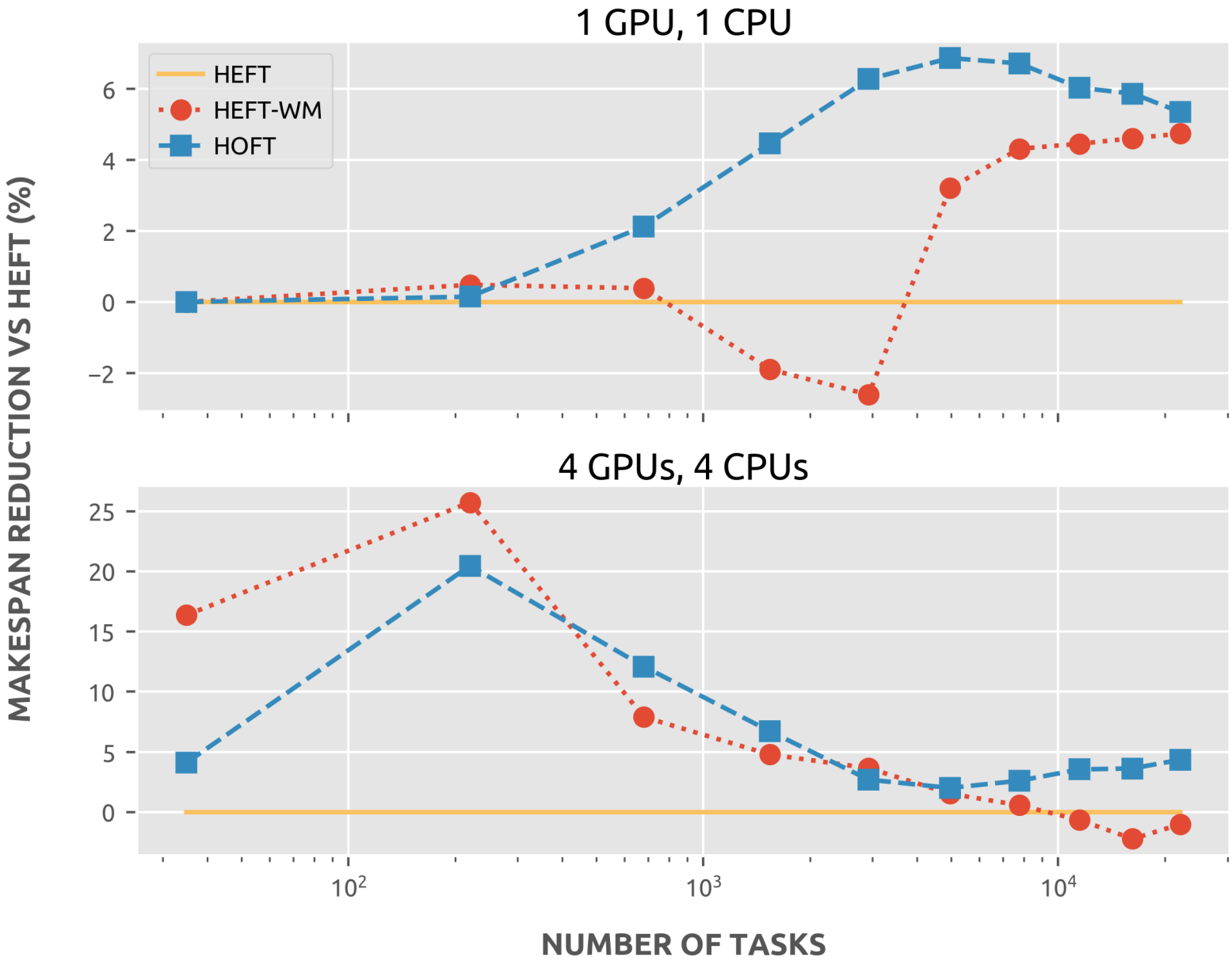
     Else:

       Schedule t_i on p_f

Cholesky—tile size 128



Cholesky—tile size 1024



Random DAGs—1002 tasks

Table: 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

Conclusions

Summary of results

- HOFT **often superior** and **always competitive**.
- HEFT-WM almost always an **improvement**.
- HOFT processor selection more effective **in general**.

Our two main contributions are:

1. A new static scheduling heuristic optimized specifically for accelerated architectures.
2. Open-source simulation software that allows the evaluation of scheduling methods for user-defined CPU-GPU platforms in a fast and reproducible manner.

Future work

- Alternative heuristics for DAGs with low CCR.
  - **Duplication**?
  - **Aggregation**?
- Consider a **wider range** of architectures and applications.
- How do we **construct** good task DAGs for a given application?
- Most importantly, new methods for adapting static schedules to real environments!
  - How to improve robustness **cheaply**?
  - **Reinforcement learning**?