A Hierarchical Task Scheduler for Heterogeneous Computing*

Narasinga Rao Miniskar[1] [0000−0001−8259−8891], Frank Liu[1] [0000−0001−6615−0739], Aaron R. Young[1] [0000−0002−5448−4667], Dwaipayan Chakraborty[2] [0000−0002−3524−9071], and Jeffrey S. Vetter[1] [0000−0002−2449−6720]

[1] Advanced Computing Systems Research Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, U.S.A.
{miniskarnr, liufy, youngar, vetter}@ornl.gov
[2] Rowan University, 201 Mullica Hill Road, Glassboro, NJ 08028, U.S.A.

Abstract. Heterogeneous computing is one of the future directions of HPC. Task scheduling in heterogeneous computing must balance the challenge of optimizing the application performance and the need for an intuitive interface with the programming run-time to maintain programming portability. The challenge is further compounded by the varying data communication time between tasks. This paper proposes RANGER, a hardware-assisted task-scheduling framework. By integrating RISC-V cores with accelerators, the RANGER scheduling framework divides scheduling into global and local levels. At the local level, RANGER further partitions each task into fine-grained subtasks to reduce the overall makespan. At the global level, RANGER maintains the coarse granularity of the task specification, thereby maintaining programming portability. The extensive experimental results demonstrate that RANGER achieves a 12.7× performance improvement on average, while only requires 2.7% of area overhead.

Keywords: Extreme Heterogeneity · Accelerators · HPC System Architecture · Challenges in Programming for Massive Scale

1 Introduction

As technology scaling comes to a standstill, heterogeneous computing has become a viable solution for ensuring the continuous performance improvement of high-performance computing (HPC). One specific notion of heterogeneous computing is the future of "extreme" heterogeneity [35], which is when the general-purpose microprocessors are augmented by diverse types of accelerators in vastly different architectures (e.g., general

* This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

purpose GPUs, FPGAs, neuromorphic arrays and special-purpose accelerator ASICs). These accelerators can have diverse functionalities (e.g., different computational kernels in machine learning) but are also spatially distributed. To fully materialize the potential of heterogeneous accelerators, it is crucial to maintain and improve the programming portability and productivity of the applications and to intelligently manage the resources presented by the heterogeneous accelerators.

Task parallelism is a highly effective parallel programming model for achieving programming portability and productivity. It allows the run-time to automatically schedule atomic computing tasks on the available resources while honoring the data dependencies between tasks. Task parallelism is widely used in many programming systems as either a direct abstraction available to the user or in the underlying implementation (e.g., OpenMP [23], OpenACC [22], CUDA [21], Charm++ [14], Cilk [6], OpenCL [16]). In task parallelism, the dependencies among different tasks can be represented by a directed acyclic graph (DAG), which contains information such as task computation time on different devices (i.e., CPUs or accelerators) represented by the nodes or vertices of the DAG, the data dependencies among the tasks represented by the directed edges, and the amount of data that must be communicated between different tasks represented as an edge property of the DAG. Even for homogeneous devices, task scheduling is NP-complete [34]. Hence, many research activities are focused on developing heuristics to ensure good completion times (i.e., makespans). Task scheduling in a heterogeneous computing environment is a much more challenging problem not only due to the different execution times of the heterogeneous devices but also due to varying data communication latency between devices.

With its many diverse accelerators, extremely heterogeneous computing poses some unique challenges for task scheduling. First, as the accelerators become more diverse, one-size-fits-all, cookie-cutter-style device management might not be optimal. Each kernel accelerator has a unique data access pattern and requirement for the hardware resources (e.g., sustained bandwidth to the global memory, size of the scratchpad memory). It is difficult to balance the needs of all kernel accelerators by a generic, centralized scheduler. Second, as more kernel accelerators with diverse capabilities become available, it becomes necessary to ensure that a larger pool of tasks is present to ensure application scalability. Performing task scheduling for many tasks for an increasing number of devices will require substantial computing resources. Finally, a larger pool of tasks poses a widening dichotomy between the optimal management of the resources and the need to ensure programming portability.

This work proposes RANGER, a hierarchical task scheduler, to address these challenges. From an algorithm perspective, RANGER performs task scheduling at two levels. At the top (i.e., global) level, RANGER considers the scheduling decision of the current task and its immediate child tasks on the decision tree to ensure global optimality. At the lower (i.e., local) level, RANGER deploys an accelerator-specific scheduler to further partition the task into subtasks while considering the nature of the computational kernel, its computational density, and the available hardware resources, such as the scratchpad memory module. Because the local schedulers have direct control over the interconnect switching fabric and other available hardware resources (e.g., accelerators, DMAs), they are capable of making optimal control decisions. In regard to imple-

mentation, instead of burdening the top-level global task scheduler with many subtasks, which are substantially more in quantity, each kernel accelerator is augmented with a customized RISC-V core to off-load the computational overhead of low-level scheduling and resource management. The most notable benefit of the hierarchical scheduling approach is that it bridges the dichotomy of coarse-grained scheduling desired by interfacing with programming models with the need for fine-grained, kernel/accelerator-dependent local task scheduling to ensure the overall optimality. Through extensive experimentation, we demonstrated that RANGER achieves a 12.7× makespan improvement on average compared with an equivalent centralized scheduler, while only requires a 2.74% area overhead. The experiments also demonstrated the excellent scalability of the RANGER architecture with respect to the number of parallel applications

The main contributions of this work are as follows.

1. We propose RANGER, a hierarchical task-scheduling framework for extremely heterogeneous computing.
2. We design and implement the overall RANGER architecture, as well as customized RISC-V cores and related logic in the GEM5 simulator [5].
3. We design and implement local Accelerator-Specific Command Schedulers (ASCS).
4. We conduct thorough experimentation to demonstrate the effectiveness of RANGER and to provide quantitative area and computational overhead.

The remainder of this paper is organized as follows. Section 2 discusses the background of task scheduling and related work. Section 3 describes the details of RANGER and its implementations. Section 4 presents experimental evaluations, followed by conclusions and future work in Section 5.

2 Background and Related Work

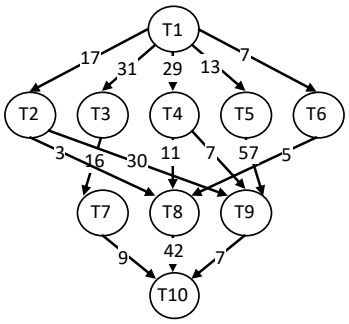
Task scheduling is a well-studied topic in disciplines such as computer architecture, programming languages, embedded systems, and real-time computing. A list of representative related works [1, 6, 7, 9, 11, 15, 17, 19, 20, 24, 29, 34] cover diverse topics, such as static (i.e., offline) and dynamic (i.e., online) scheduling, scheduling with hard deadlines, and hardware-enabled scheduling policies.

In this paper, we assume that the dependencies among tasks are either fully or partially known. A widely accepted formalism to describe task dependencies is based on graphs [1, 33]. An example is shown in Fig. 1 in which an application is represented by a DAG, defined by the tuple $G = (V, E)$, and a companion computation cost matrix (CCM). The vertex set V of size v represents the tasks in the application, and the edge set E of size e represents the data dependencies between the tasks. If there is an edge (e_{ij}) from task T_i to T_j, it means that task T_j has a data dependency to task T_i. Hence, task T_j cannot start until task T_i is completed. The CCM of size $v \times p$ represents the execution time of each task on each processing device, where p is the size of the processor set P. Each edge also contains a weight, which represents the communication cost between the tasks. Theoretically, the communication time depends not only on the tasks but also on which device the communication is originated and on which device it is terminated. However, one common approximation is to estimate the communication

time based on the amount of data that must be transferred and the average communication bandwidth, as well as the average starting latency. This common approximation is defined as follows [33]:

$$\overline{c}_{(i,j)} = \overline{L} + \frac{data_{(i,j)}}{\overline{B}}, \tag{1}$$

where $\overline{B}$ is the average communication bandwidth between computing devices, and $data_{(i,j)}$ reflects the amount of data that must be transferred from task i to task j. $\overline{L}$ represents the average latency before any bulk data communication can be started. Generally, $\overline{L}$ could include hardware latency, such as the signal handshaking time for interconnects, and software latency, such as the time needed for context switching by the operating systems. With this approximation, the average communication time between tasks can be determined as shown by the edge weights in Fig. 1.



Task	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
P1	22	22	32	7	29	26	14	29	15	13
P2	21	18	27	10	27	17	25	23	21	16
P3	36	18	43	4	35	24	30	36	8	33

Fig. 1: DAG of an application with 10 tasks and the CCM for each task on three devices.

The objective of task scheduling is to minimize the overall execution time of the application or makespan. Because the task scheduling problem is NP-complete [34], many existing methods are based on heuristics. Among static task schedulers, Predict Earliest Finish Time (PEFT) [1] has a good trade-off between accuracy and computational complexity. Based on the Heterogeneous Earliest Finish Time (HEFT) [33] scheduler, PEFT considers the impact that the current scheduling decision has on all subsequent scheduling decisions within the decision tree. To estimate the potential impact of a scheduling decision, it uses a clever method to compute an optimal cost table, thereby avoiding the costly operation of fully traversing the whole decision tree.

The concept of *a task* is the atomic unit for data transferring and computation. Implicitly, it is assumed that the data needed for each task have been readily transferred to the local scratchpad memory on each device before the computation can start. For each application, the task specifications can be given at different levels. For example, tasks can be further partitioned into finer subtasks by inspecting the computation kernels and the dependencies between more fine-grained subcomponents. The granularity of the

task specification has significant practical implications. On one hand, coarse-grained task specifications are more acceptable and have better programming portability in the programming models. They also imply fewer number data transfers and better data transferring efficiency. However, a higher computational density in each task also means that larger amounts of data must be transferred and stored on the local memory, which requires a larger local scratchpad memory module on each device. On the other hand, partitioning tasks into finer granularity has the benefit of requiring smaller scratchpad memory on each accelerator device. The larger number of tasks also gives the scheduler the opportunity to fully leverage all available devices, thus potentially achieving better scalability. However, because there are many more data transferring jobs, data transferring becomes less efficient due to the larger number of starting latency.

The biggest hurdle of fine-grained tasks specification is the severe loss of the programming portability. As shown later in this paper, the need to specify many tasks in the programming systems makes it difficult to interface with the programming models. This constraint was investigated in a recent study [31] by analyzing applications' performance and their efficiency via a proposed metric called *minimum effective task granularity*. The paper concluded that the cost of sending data reflected as part of hardware latency and dispatching tasks reflected in the software latency impose a floor on the granularity.

Another body of closely related work is off-loading scheduling tasks to hardware. The concept was explored in several studies [3,8,26,30,30]. In a more recent study [20], a Rocket Chip [4]-based hardware scheduler demonstrated impressive performance improvements when compared with a pure software implementation of the same scheduling method.

In this paper, we propose RANGER, a hierarchical scheduler, to address the conflicting dichotomy between programming portability and the requirements for better resource management in heterogeneous computing. At the top (i.e., global) level, RANGER maintains the task specifications at a coarse granularity. Hence, it is easy for RANGER to interface with existing task dependency specification mechanisms implemented in various programming models. At the lower level, RANGER uses Accelerator-Specific Command Scheduler (ASCS), which are specifically designed for each device to interface with the top-level scheduler and global memory. Based on the characteristics of the computational kernel and available resources, ASCS partitions each task into subtasks and manages them. In the RANGER architecture, the ASCS is executed on a customized RISC-V [36] processor core embedded in each accelerator to off-load the computations of task scheduling from the central host. The fine-grained subtask specifications and local ASCS schedulers ensure better use of the local hardware resources, such as the local scratchpad memory. The amortized data communication latency also makes it possible for the global coarse-grained scheduler to better use available accelerator devices. Unlike other hardware off-loading of task scheduling work—such as in Arnold et al. [3] and Morais et al. [20]—the main novelty of RANGER is the combination of two techniques by developing hardware-assisted hierarchical scheduling to address the conflicting dichotomy between programming portability and the resource requirement.

3 RANGER Architecture and Implementation

This section describes the RANGER architecture design, accelerator devices, and overall hierarchical task-scheduling method.

3.1 Baseline Accelerator Architecture

The baseline heterogeneous computing platform is a heterogeneous computing platform, as shown in Fig. 2. The host, an array of heterogeneous accelerator devices, and the main memory are connected by a shared bus. A top-level run-time runs on the host, which is responsible for communicating with the applications, determining the task schedules and assignments, and dispatching the scheduling decisions to each device. Once a task is dispatched to a device, each device would communicate with the global memory through its DMA channel, fetch the needed data from global memory to scratchpad memory on the device, launch the computation, and transfer the results back to global main memory after the task has completed. The global main memory usually has the characteristics of higher density and lower cost per gigabyte but longer access time (e.g., DRAM memory). On the other hand, the local scratchpad memory has much faster access time but at a lower density (e.g, SRAM memory). First-in-first-out (FIFO) logic circuits are inserted at various interfaces. The global bus can be implemented with different switching fabrics. This study uses the AXI bus [2], the industry standard for on-chip interconnect, although it is also possible to use various network-on-chip (NoC) switching fabrics.

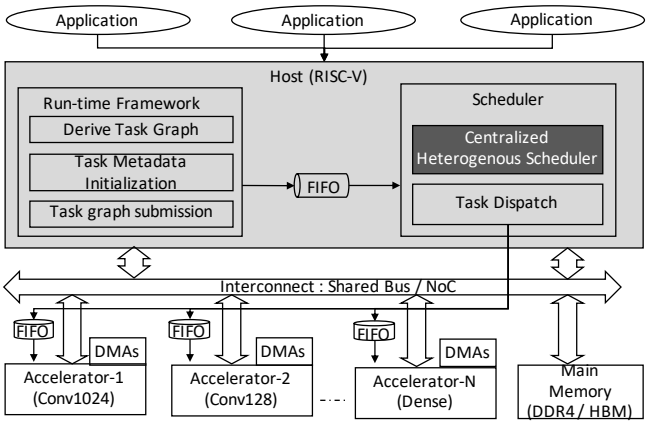


Fig. 2: Baseline architecture design of the heterogeneous accelerator platform.

3.2 RANGER Architecture and Memory-Mapped IO Interface

The RANGER architecture is shown in Fig. 3. Compared with the baseline architecture design shown in Fig. 2, each device is augmented with a RISC-V processor, which

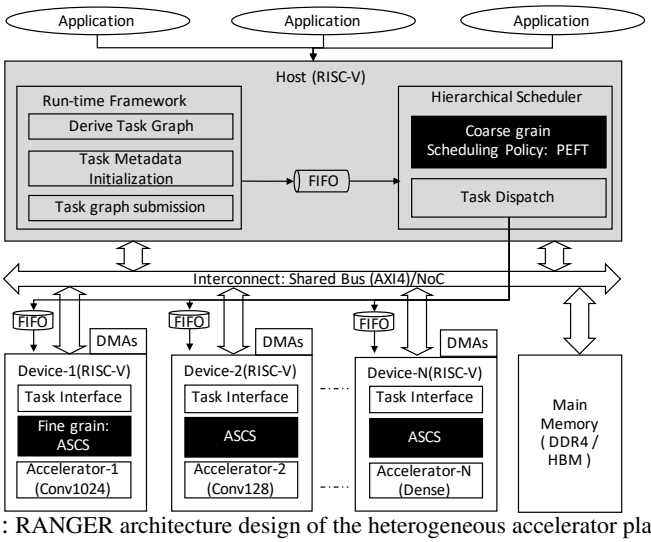


Fig. 3: RANGER architecture design of the heterogeneous accelerator platform.

takes control of the interface to the global bus, as well as the scratchpad memory and accelerator.

A more detailed view of each device is shown in Fig. 4. The accelerator on each device interfaces with the RISC-V core through memory-mapped I/O. This is facilitated by the added MEMCTRL logic, which can determine whether the requested memory operation (LOAD/STORE) is intended for the accelerator, the DMA-In or DMA-Out channel to the local scratchpad memory, or the main memory (DRAM). Furthermore, each DMA channel and the accelerator on the device also have their own unique memory-mapped registers.

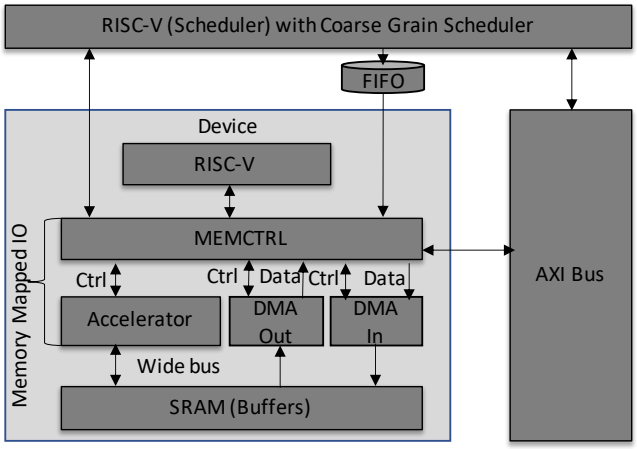


Fig. 4: RANGER Device: RISC-V, accelerator and DMA channels with memory-mapped I/O.

As an illustration, a memory-mapped register for a convolution accelerator (CONV) is shown in Tab. 1. The DMA memory-mapped I/O configuration enables the transferring of any tile representing a 3D array in the DRAM memory to the local scratchpad SRAM by specifying offsets in the (x, y, z) plane. The number of banks in the SRAM is designed to be twice the number of inputs and outputs required for the accelerator to ensure that the SRAM design is equivalent to a bank conflict-free design. The assignment of the SRAM bank to the input buffer is determined by the controller kernel, which runs on the embedded RISC-V core.

Table 1: Memory-mapped IO registers of a convolution accelerator in RANGER

Register	MEMIO Offset	Range of Values
ACC_SRC_ADDR	0x00	32-bit SRAM address
ACC_DST_ADDR	0x04	32-bit SRAM address
ACC_IN_XYZ_OFFSET	0x08	3D-(X,Y,Z) offsets, each 20bit
ACC_OUT_XYZ_OFFSET	0x10	3D-(X,Y,Z) offsets, each 20bit
ACC_OUT_XYZ_SIZE	0x18	64-bit (X,Y,Z) size, each 20bit
ACC_KERNEL_STRIDE	0x20	32-bit (Kernel-size, Stride-size)
ACC_CTRL	0xF8	0: DO NOTHING 1: START Computation
ACC_STATUS	0xFC	Returns status 0: Busy Running 1: Free

3.3 Top-Level Scheduler

The top-level, coarse-grained global scheduler runs on the host shown in Fig. 3. Theoretically, any task scheduler from a rich body of research can be used as the top-level scheduler. In this study, PEFT [1] was implemented as the global scheduler to make it easier to compare RANGER performance with other solutions. To populate the required CCM of PEFT, each device—RISC-V, the added control logic, the DMA interface, and the accelerator—was implemented in GEM5. Each task was profiled to generate the corresponding entry in the CCM. It is also possible to use other performance prediction techniques, such as Johnston et al. [13] or Liu et al. [18]. The run-time range of each task in this study is on the order of milliseconds. Hence, extra effort was taken to optimize the coarse-grained scheduler to ensure that each scheduling decision can be completed within 1 ms on the customized RISC-V core. The computed scheduling decisions are formatted as the mapping of tasks to the available devices in which each task specification contains a set of commands and input/output memory locations. These commands are pushed into the FIFO queue of the corresponding device, as shown in Fig. 3.

3.4 Low-Level Scheduler

The low-level scheduler is responsible for further partitioning the given task into finer granularity. The partitioning and sequencing of the subtasks and their dependencies can vary from one kernel accelerator to the other. For example, a convolution kernel requires

two sets of input and generates one set of output, whereas a batch normalization (BN) kernel has only one set of input and one set of output. Furthermore, each of these two kernels has different computational density. To maintain the flexibility and ensure the optimality, the authors developed ASCS. Based on the sizes of the inputs and outputs of a given task, its computational density, and the amount of available scratchpad SRAM memory, an ASCS scheduler generates subtasks, each of which also contains the instructions on how to configure the accelerator/DMA-channel-specific memory-mapped addresses. By doing so, ASCS provides more fine-grained control of the DMA channels and its interface to the global DRAM.

Because of their application-specific nature, the ASCS schedulers are specially tailored for each kernel accelerator as a part of the accelerator development process. They are also parameterized so that when the accelerator hardware specification (e.g., the size of the accelerator, the amount of scratchpad memory) changes, the ASCS schedulers can be easily updated.

For example, a high-level description of the ASCS scheduler for the convolution accelerator is shown in Algorithm 1. The functionality of a convolution operation is to compute a stream of output by convoluting a stream of input with a set of given weights. The convolution is carried out by the unit of "tile." Depending on the available scratchpad memory, different tile size configurations can lead to different decisions on whether to leverage weight reuse, input reuse, or output reuse. In Algorithm 1, the subtasks pipeline is created and initialized in lines 1 and 2. Depending on the tile configuration, single or double buffers are allocated, as shown in line 3. After initialization, the four nested loops shown from lines 4 to 7 iterate in order of tile height, tile weight, the channels, and the inputs. The pipelined subtasks are identified by their individual timestamps and maintained in a circular queue.

Within the innermost loop, the DMA process to fetch the weights, inputs, and outputs will be activated, as required, based on the data reuse leveraged by the kernel. Lines 11, 13, and 16 indicate the configuration of DMA channels for the weights, input, and output tiles. Line 17 is the computation of the scheduled subtask. Line 18 performs the configuration of the DMA channels and accelerator for the next subtask in the queue. All three routines are nonblocking and thus are executed concurrently.

The ASCS routines are designed and developed to be lightweight and are executed on the RISC-V processor on each accelerator device. Combined with the added peripheral logic, they provide configurability and flexibility to the accelerator. By considering the available computational resources (e.g., the number of Multiplier-Accumulator or MAC units), computational density of the kernel, and size of the available scratchpad memory, the ASCS schedulers in RANGER ensure that the optimal tile size is used so that the execution time of the subtasks, DMA transferring time, and computation time of the subtasks are balanced to achieve maximal throughput for the given task, as illustrated in Fig. 5.

3.5 Implementation Details of Accelerator Kernels

This study implemented three types of accelerators: 2D CONV, BN, and fully connected dense layer (DENSE). Multiple flavors of each type are implemented by changing the number of MAC units and the scratchpad memory size. For example, five fla-

```
// Input feature map: (OH, OW, OD),      Output feature map: (IH, IW, ID)
// Weights : (OD, ID, KH, KW)
// Input Tile: (oth, otw, otd),          Output Tile: (ith, itw, itd)

(weights_task, ifmap_task, kernel_task, ofmap_task) = CreateSubTasksForCommands(task);
pipeline = new Pipeline ( [weights_task, ifmap_task, kernel_task, ofmap_task ] )
AllocateStreamDoubleBuffers(pipeline)
for oh: 0 to OH step oth do
  for ow: 0 to OW step otw do
    for od: 0 to OD step otd do
      for id: 0 to ID. step itd do
        ts = pipeline->GetCurrentTimeStamp();
        next_ts = pipeline->GetNextTimeStamp();
        if weight_dma_to_be_activated:
          pipeline->ConfigureDMA(next_ts, weights_task, dram_weights_address, sram_weights_address)
        if input_fmap_to_be_activated:
          pipeline->ConfigureDMA(next_ts, ifmap_task, dram_ifmap_address, sram_ifmap_address)
        pipeline->ConfigureKernel(next_ts+1, kernel_task, sram_weights_address, sram_ifmap_address,
                                                          sram_ofmap_address)
        if output_fmap_to_be_activated:
          pipeline->ConfigureDMA(next_ts+2, ofmap_task, sram_ofmap_address, dram_ofmap_address)
        pipeline->RunCurrentTimeStampTasks()
        pipeline->ConfigureNextTimeStampTasks()
        pipeline->WaitForTasks()
      end
    end
  end
end
pipeline->RunAllPrologTasks()
```

Algorithm 1: High-level description of ASCS scheduling algorithm for the convolution kernel.

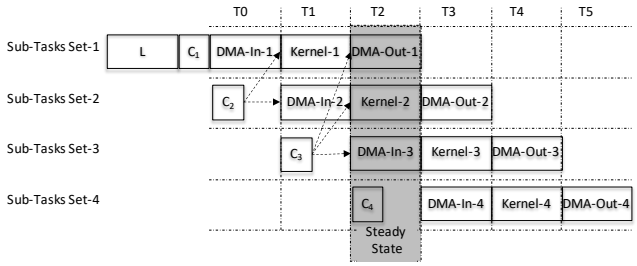


Fig. 5: Illustration of ASCS subtask execution pipeline. The "C" blocks include the configuration of DMA channels and accelerators for each subtask. Starting from the second subtask, they are concurrently executed with the DMA transfer of the previous subtask.

vors of CONV accelerator were implemented: CONV1024, CONV512, CONV256, CONV128, and CONV64. Generally speaking, the bigger accelerators have higher performance but also require larger area. The accelerator statistics are listed in Tab. 2. The BN accelerator requires less scratchpad memory because the nature of the kernel is similar to the inner product operation in the numerical linear algebra. The area estimates are based on the data extracted from various designs fabricated on a TSMC 16 nm CMOS technology [27].

Using these kernel accelerators, we implemented multiple heterogeneous designs of the RANGER architecture, as shown in Tab. 3. The cycle-accurate GEM5 simulator was

Table 2: List of kernel accelerators and their area estimations in a TSMC 16 nm technology.

	Accelerator	Functionality	MAC Units	SRAM Size (KB)	Area (mm^2)
1	CONV1024	2D Convolution	1024	256.0	1.81
2	CONV512	2D Convolution	512	256.0	1.28
3	CONV256	2D Convolution	256	256.0	1.01
4	CONV128	2D Convolution	128	256.0	0.88
5	CONV64	2D Convolution	64	128.0	0.59
6	BN1024	Batch Normilization	1024	8.0	1.09
7	BN512	Batch Normilization	512	4.0	0.55
8	BN256	Batch Normilization	256	2.0	0.28
9	BN128	Batch Normilization	128	1.0	0.14
10	BN64	Batch Normilization	64	0.5	0.08
11	DENSE1024	Dense	1024	128.0	1.30
12	DENSE512	Dense	512	128.0	0.76
13	DENSE256	Dense	256	128.0	0.50
14	DENSE128	Dense	128	128.0	0.37
15	DENSE64	Dense	64	128.0	0.30

extended [5] to include the accelerators, memory-mapped I/O logic, and DMA components. The DMA component was implemented from scratch with the support of tiled 3D data transfer and burst mode use of the DMA channels. For each heterogeneous design, we also implemented a baseline version with only the accelerators and without the RISC-V core and extra control logic. Overall, the implementation is realized by ∼1,800 lines of C++ code and ∼700 lines of Python code for configuration. The statistics and characteristics of these RANGER and baseline designs are tabulated in Tab. 3.

Table 3: Various heterogeneous designs, the number of kernel accelerators, and the estimated area. Design A and B are the aliases of Design 1 and 2, respectively.

	Accelerators																Area mm^2			
	2D Convolution					Batch Normilization					Dense					Total				
	1024	512	256	128	64	1024	512	256	128	64	1024	512	256	128	64		RANGER	Baseline	Overhead
Design																				
Design 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	15	11.290	10.945	3.15 %
Design 2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	30	22.581	21.891	3.15 %
Design 3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	35	28.266	27.461	2.93 %
Design 4	4	4	4	4	4	2	2	2	2	2	2	2	2	2	2	40	33.951	33.031	2.79 %
Design 5	5	5	5	5	5	2	2	2	2	2	2	2	2	2	2	45	39.636	38.601	2.68 %
Design 6	6	6	6	6	6	2	2	2	2	2	2	2	2	2	2	50	45.321	44.171	2.60 %
Design 7	7	7	7	7	7	2	2	2	2	2	2	2	2	2	2	55	51.006	49.741	2.54 %
Design 8	8	8	8	8	8	2	2	2	2	2	2	2	2	2	2	60	56.691	55.311	2.49 %
Design 9	9	9	9	9	9	2	2	2	2	2	2	2	2	2	2	65	62.376	60.881	2.46 %
Design 10	10	10	10	10	10	2	2	2	2	2	2	2	2	2	2	70	68.061	66.451	2.42 %
Design A	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	15	11.290	10.945	3.15 %
Design B	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	30	22.581	21.891	3.15 %
Design C	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	45	33.871	32.836	3.15 %
Design D	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	60	45.161	43.781	3.15 %
Design E	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	75	56.452	54.727	3.15 %
Design F	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	90	67.742	65.672	3.15 %
Design G	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	105	79.032	76.617	3.15 %
Design H	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	120	90.323	87.563	3.15 %
Average																				2.74 %

The area of each RISC-V core is estimated to be 0.023 mm^2 [37] in the TSMC 16 nm technology. Tab. 3 shows that the RANGER architecture only requires ∼2.74% of area overhead compared with the corresponding baseline designs. The largest RANGER core has an estimate area of 90 mm^2. As a comparison, a quad-core Intel Coffee Lake processor has the die area of 126 mm^2 in a comparable technology [12].

4 Experimental Evaluation

To evaluate the performance of RANGER, the authors used a run-time framework running on the host RISC-V core shown in Fig. 3. The task descriptions are specified in JSON format, which is the output of a Python-based converter. Values of CCM of the top-level scheduler are the profiling results of the individual kernels in GEM5. The makespans of tasks and applications are extracted from the performance counters of DMAs from GEM5.

4.1 Application Benchmarks

The inference phase of four widely used deep neural networks (DNNs) were used as the benchmarks: InceptionV3 [32], ResNet-50 [10], UNet [25], and Vgg16 [28]. We would like to point out that RANGER is a general-purpose scheduler and can handle any DAG tasks. In this study DNNs inference applications were the chosen simply because their DAGs are readily available, and they represent increasingly important workloads. The details of these four DNNs are omitted due to space limitation. Each inference application comprises three types of computational kernels: CONV, BN, and fully connected DENSE. For example, InceptionV3 is represented by 189 tasks—94 CONV, 94 BN, and one DENSE—with its task DAG shown in Fig. 6.

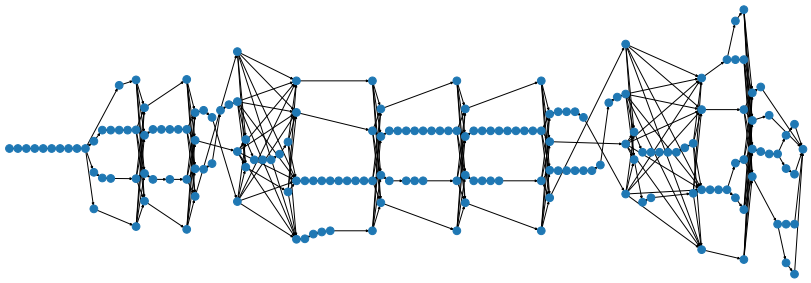


Fig. 6: Task DAG of Inception-v3. The left-most node is the source node of the DAG, and the sink node is at the extreme right.

The execution time of each application is measured by the cycle counts reported by GEM5. For comparison, the same set of applications was also run on the baseline design shown in Fig. 2 with the detailed design specs tabulated in Tab. 3. The results are tabulated in Tab. 4. As shown in the table, across multiple design points, the average speedup of Inception-v3, ResNet, VGG16, and UNet achieved by RANGER are 17.66×, 9.10×, 16.96×, and 6.96× respectively, with the average speedup of 12.7×

Table 4: Comparison of makespans for various RANGER and baseline designs. On average, RANGER achieves a 12.7× speedup.

Model Architecture Design	Makespan Inception-v3 RANGER	Baseline	Speedup	Resnet-50 RANGER	Baseline	Speedup	VGG16 RANGER	Baseline	Speedup	UNet RANGER	Baseline	Speedup
Design 1	94,788,333	762,320,311	8.04×	88,346,488	504,482,689	5.71×	389,202,487	3,193,921,550	8.21×	336,496,490	1,233,341,834	3.67×
Design 2	53,422,061	752,853,720	14.09×	57,531,844	500,829,541	8.71×	209,315,117	3,149,123,579	15.04×	201,002,387	1,177,865,323	5.86×
Design 3	41,881,254	752,544,223	17.97×	52,782,400	500,433,710	9.48×	191,533,321	3,147,951,787	16.44×	173,986,414	1,171,882,097	6.74×
Design 4	38,749,897	752,450,123	19.42×	52,195,186	500,340,314	9.59×	181,632,752	3,147,767,996	17.33×	157,652,345	1,169,892,512	7.42×
Design 5	37,999,017	752,407,810	19.80×	52,191,088	500,295,143	9.59×	180,193,518	3,147,649,984	17.47×	150,250,684	1,169,590,055	7.78×
Design 6	37,988,551	752,393,040	19.81×	52,191,088	500,288,595	9.59×	180,193,518	3,147,724,843	17.47×	150,250,684	1,169,578,215	7.78×
Design 7	37,914,589	752,387,551	19.84×	52,190,081	500,282,221	9.59×	180,193,518	3,147,642,459	17.47×	150,250,684	1,169,575,296	7.78×
Design 8	37,898,912	752,382,660	19.85×	52,190,081	500,275,555	9.59×	180,193,518	3,147,724,163	17.47×	150,250,684	1,169,574,480	7.78×
Design 9	37,891,335	752,382,660	19.86×	52,190,081	500,272,549	9.59×	180,193,518	3,147,669,094	17.47×	150,250,684	1,169,574,480	7.78×
Design 10	37,618,903	752,367,084	20.00×	52,190,081	500,264,005	9.59×	178,630,640	3,147,724,117	17.62×	147,870,807	1,169,574,480	7.91×
Design A	94,788,333	762,320,311	8.04×	88,346,488	504,482,689	5.71×	389,202,487	3,193,921,550	8.21×	336,496,490	1,233,341,834	3.67×
Design B	53,422,061	752,853,720	14.09×	57,531,844	500,829,541	8.71×	209,315,117	3,149,123,579	15.04×	201,002,387	1,177,865,323	5.86×
Design C	42,241,318	752,541,894	17.82×	52,447,643	500,420,375	9.54×	174,801,184	3,144,474,922	17.99×	173,986,414	1,171,816,637	6.74×
Design D	38,570,566	752,444,207	19.51×	51,672,158	500,332,941	9.68×	155,344,163	3,144,196,072	20.24×	157,652,123	1,169,797,015	7.42×
Design E	37,793,805	752,406,134	19.91×	51,362,009	500,254,081	9.74×	153,596,950	3,144,134,148	20.47×	150,247,240	1,169,471,954	7.78×
Design F	37,793,452	752,401,577	19.91×	51,162,596	500,254,081	9.78×	153,596,950	3,144,134,148	20.47×	150,247,240	1,169,470,736	7.78×
Design G	37,708,203	752,386,686	19.95×	51,162,596	500,254,081	9.78×	153,596,950	3,144,134,148	20.47×	150,247,240	1,169,468,346	7.78×
Design H	37,704,900	752,386,686	19.95×	51,161,299	500,254,081	9.78×	153,596,950	3,144,134,148	20.47×	150,247,240	1,169,468,346	7.78×
Average			17.66×			9.10×			16.96×			6.96×

across all four applications. To execute the benchmarks on the accelerator cores by using the baseline architecture, the tasks in each application must be further partitioned based on the amount of scratchpad memory available. The numbers of the fine-grained subtasks are shown in Tab. 5, which also includes the corresponding RANGER task numbers as a comparison. In this case, scheduling these subtasks is computed by the host. Fig. 7 shows RANGER scheduling decisions of 10 Inception-v3 applications running in parallel on Design 1 from Tab. 3.

Table 5: The numbers of tasks that the global host must consider in RANGER and baseline architecture.

Architecture Model	RANGER	Baseline	Increase
Inception-v3	189	20,469	108×
Resnet-50	107	6,824	64×
UNet	17	12,372	728×
VGG16	16	38,064	2,379×

4.2 Scalability Study

This section further investigates the scalability of the RANGER architecture. To saturate the many kernel accelerators in the designs, we increased the repetition of the applications to 10 (i.e., during each experiment,10 identical but independent applications were issued on a given RANGER design). The results are plotted with respect to various RANGER designs in Fig. 8. This study is similar to the strong-scaling study in the traditional HPC applications. The plot clearly shows that the speedup is plateaued to $\sim$2× at Design 3, which has 35 kernel accelerators.

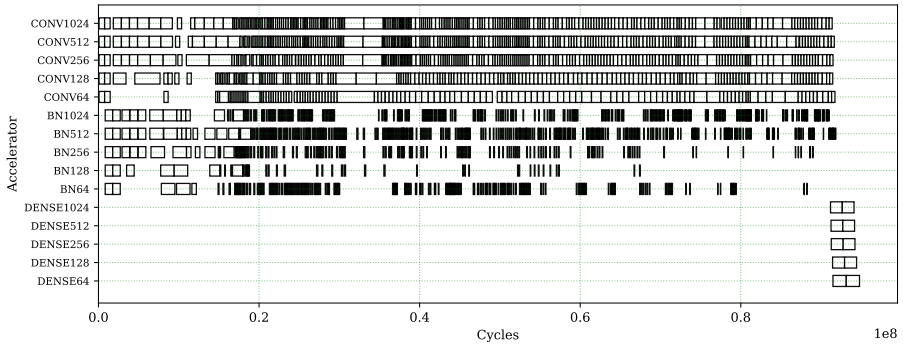


Fig. 7: Scheduling decisions of 10 parallel instances of Inception-v3 computed by RANGER on Design 1. Each box represents a task being scheduled on a particular device.

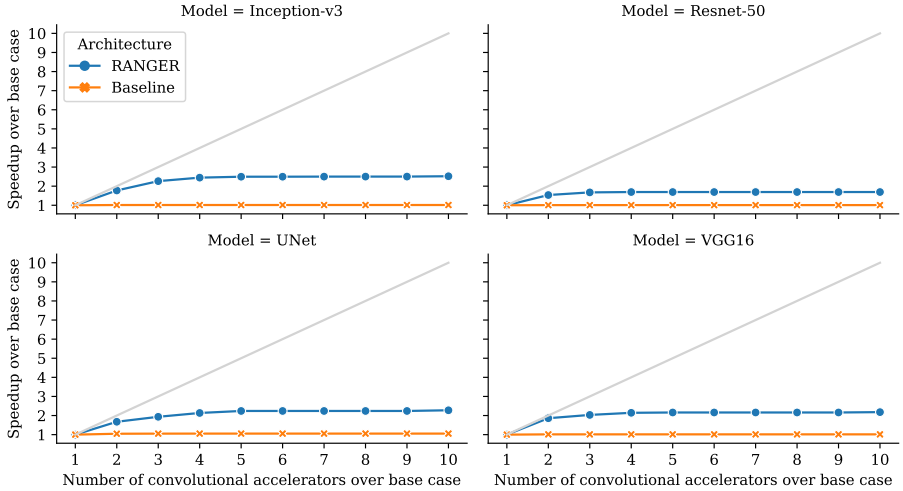


Fig. 8: Measured speedup of RANGER by running 10 parallel instances of each application with respect to Designs 1–10, which contain an increasing number of kernel accelerators. The speedup plateaus at Design 3 due to an insufficient number of tasks for the available kernel accelerators.

In the second study, the experiment was repeated by varying the application repetitions, which is similar to the weak-scaling study of HPC workloads. The results are shown in Fig. 9. With repetition set to 100, the RANGER architecture demonstrates good scalability from Design A, which has 15 kernel accelerators, to Design H, which contains 120 kernel accelerators.

4.3 Overhead of the Local Schedulers

The speedups achieved by RANGER are contributions of the top-level hierarchical scheduling scheme and the implementation of the low-level ASCS. To investigate the performance inefficiency caused by the two-level scheduling scheme, a collection of

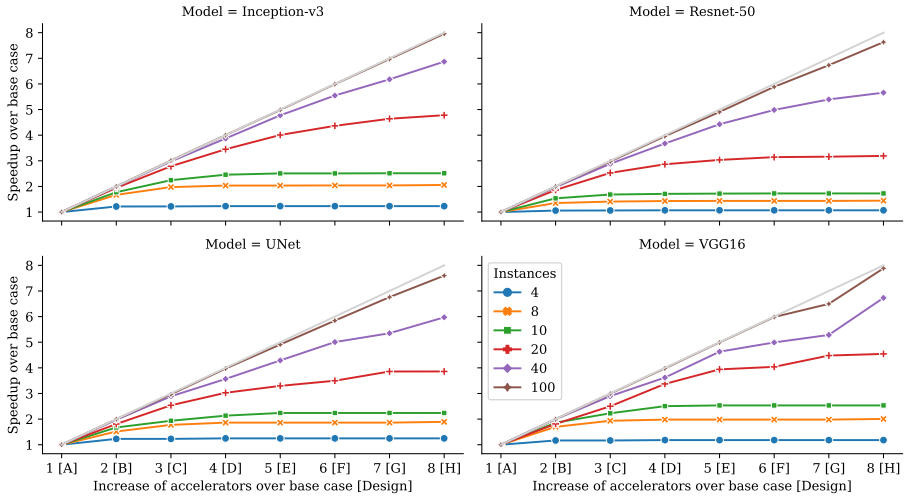


Fig. 9: Measured speedup of RANGER by running an increasing number of instances of the same application on Designs A–H. RANGER demonstrates excellent scalability with 100 instances of application running in parallel.

hypothetical reference designs were designed. Compared with RANGER designs, each reference design has an identical number of kernel accelerators as its RANGER counterpart but with a sufficiently large scratchpad memory to accommodate all needed data. Hence, there is no need to invoke local schedulers because there is no need to further partition each task. Instead, the host processor can directly dispatch the tasks to the accelerators based on the top-level scheduling decisions. With much larger scratchpad memory modules, the estimated areas of the reference designs are listed in Tab. 6. These reference designs cannot be realistically implemented due to their large areas. For instance, nine out of 10 reference designs have the estimate area of over 200 mm^2, which makes them extremely expensive to manufacture. As a reference point, an octa-core Intel Coffee Lake processor on a comparable technology only has a die area of 174 mm^2 [12].

The comparison on the RANGER design makespans and the reference designs are tabulated in Tab. 7. For Inception-v3 and ResNet, RANGER designs clearly have similar makespans with small but consistent improvements. For VGG16 and UNet, RANGER designs show a 31–21% degradation of the makespans. Across all four applications, RANGER shows an average of 10.88% on makespan penalties. The host runs the identical top-level scheduler with identical DAG specifications in both RANGER and reference studies. Therefore, the measured penalty directly indicates the performance overhead of the low-level ASCS scheduler. However, given the impracticality of the reference designs, the authors believe that this magnitude of overhead is completely acceptable.

Table 6: Area estimate of reference designs compared with their RANGER counterparts. Designs of these sizes are extremely expensive to manufacture.

Model Architecture Design	Area mm^2 RANGER	Reference	Difference
Design 1	11	165	14.61×
Design 2	23	275	12.18×
Design 3	28	337	11.93×
Design 4	34	440	12.96×
Design 5	40	445	11.23×
Design 6	45	484	10.67×
Design 7	51	512	10.04×
Design 8	57	546	9.63×
Design 9	62	603	9.67×
Design 10	68	662	9.73×

Table 7: Comparison of makespans for RANGER and reference. On average, RANGER shows only 10.88% of penalty, which is the measurement of performance overhead of the local ASCS.

Model Architecture Design	Makespan Inception-v3 RANGER	Reference	Difference	Resnet-50 RANGER	Reference	Difference	VGG16 RANGER	Reference	Difference	UNet RANGER	Reference	Difference
Design 1	94,788,333	95,707,644	-0.96%	88,346,488	88,621,038	-0.31%	389,202,487	214,684,328	81.29%	336,496,490	216,796,889	55.21%
Design 2	53,422,061	56,538,771	-5.51%	57,531,844	63,717,300	-9.71%	209,315,117	165,634,098	26.37%	201,002,387	156,806,696	28.18%
Design 3	41,881,254	45,388,469	-7.73%	52,782,400	58,538,562	-9.83%	191,533,321	150,559,308	27.21%	173,986,414	145,184,318	19.84%
Design 4	38,749,897	40,891,867	-5.24%	52,195,186	56,076,426	-6.92%	181,632,752	144,136,809	26.01%	157,652,345	134,955,240	16.82%
Design 5	37,999,017	39,601,255	-4.05%	52,191,088	55,193,132	-5.44%	180,193,518	141,111,694	27.70%	150,250,684	129,117,096	16.37%
Design 6	37,988,551	38,921,698	-2.40%	52,191,088	54,871,507	-4.88%	180,193,518	139,536,510	29.14%	150,250,684	128,893,489	16.57%
Design 7	37,914,589	38,412,723	-1.30%	52,190,081	54,847,500	-4.85%	180,193,518	138,354,082	30.24%	150,250,684	127,945,051	17.43%
Design 8	37,898,912	38,260,460	-0.94%	52,190,081	54,561,879	-4.35%	180,193,518	137,473,689	31.07%	150,250,684	127,915,851	17.46%
Design 9	37,891,335	38,260,436	-0.96%	52,190,081	54,561,879	-4.35%	180,193,518	137,473,689	31.07%	150,250,684	127,915,851	17.46%
Design 10	37,618,903	37,782,552	-0.43%	52,190,081	53,711,730	-2.83%	178,630,640	134,278,895	33.03%	147,870,807	121,184,411	22.02%
Average			-2.95%			-5.06%			31.01%			20.53%

5 Conclusion

This paper presents RANGER, a framework and architecture design for hierarchical task scheduling in extremely heterogeneous computing. As a framework, one crucial benefit of hierarchical scheduling is that it only requires coarse-grained task dependency specifications at the top level, whereas more fine-grained, accelerator-specific scheduling can be performed at the lower level. The coarse-grained task specifications make it much easier to maintain programming portability and productivity in heterogeneous computing. Introducing localized low-level schedulers enables the deployment of more sophisticated, accelerator-specific scheduling solutions to better utilize hardware resources. From an architecture perspective, RANGER uses customized RISC-V cores to mitigate the computational overhead of task scheduling. Through extensive experimentation, we demonstrated that RANGER architecture achieves 12.7× performance gains on average in terms of makespan with only a 2.7% area overhead in a 16 nm technology.

In future work, we plan to further improve the global and local schedulers in RANGER. They also plan to integrate RANGER with contemporary parallel run-times to further explore its potential.

Acknowledgments

This material is based upon work supported by the US Department of Energy (DOE) Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC05-00OR22725.

This research was supported in part by the DOE Advanced Scientific Computing Research Program Sawtooth Project and the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle LLC for DOE.

References

1. 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 (2013)
2. ARM Corp.: AMBA: the standard for on-chip communication. https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications, accessed: 2020-12-10
3. Arnold, O., Noethen, B., Fettweis, G.: Instruction set architecture extensions for a dynamic task scheduling unit. In: 2012 IEEE Computer Society Annual Symposium on VLSI. pp. 249–254. IEEE (2012)
4. Asanovic, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., et al.: The Rocket chip generator. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17 (2016)
5. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The GEM5 simulator. ACM SIGARCH computer architecture news **39**(2), 1–7 (2011)
6. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of parallel and distributed computing **37**(1), 55–69 (1996)
7. Canon, L.C., Marchal, L., Simon, B., Vivien, F.: Online scheduling of task graphs on heterogeneous platforms. IEEE Transactions on Parallel and Distributed Systems (2019)
8. Dallou, T., Engelhardt, N., Elhossini, A., Juurlink, B.: Nexus#: A distributed hardware task manager for task-based programming models. In: 2015 IEEE International Parallel and Distributed Processing Symposium. pp. 1129–1138. IEEE (2015)
9. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. pp. 212–223 (1998)
10. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)
11. Huang, T.W., Lin, C.X., Guo, G., Wong, M.: Cpp-taskflow: Fast task-based parallel programming using modern c++. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 974–983. IEEE (2019)

12. Intel Corp.: Coffee lake - microarchitecture - intel. https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake, accessed: 2020-12-10
13. Johnston, B., Milthorpe, J.: AIWC: OpenCL-based architecture-independent workload characterization. In: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). pp. 81–91. IEEE (2018)
14. Kale, L.V., Krishnan, S.: Charm++: Parallel programming with message-driven objects. In: Wilson, G.V., Lu, P. (eds.) Parallel Programming using C++, vol. 1, pp. 175–213. MIT Press, Cambridge, MA (1996)
15. Kaleem, R., Barik, R., Shpeisman, T., Hu, C., Lewis, B.T., Pingali, K.: Adaptive heterogeneous scheduling for integrated gpus. In: 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT). pp. 151–162. IEEE (2014)
16. Khronos Group: OpenCL: The open standard for parallel programming of heterogeneous systems (2019)
17. Kukanov, A., Voss, M.J.: The foundations for scalable multi-core software in Intel Threading Building Blocks. Intel Technology Journal **11**(4) (2007)
18. Liu, F., Miniskar, N.R., Chakraborty, D., Vetter, J.S.: DEFFE: a data-efficient framework for performance characterization in domain-specific computing. In: Proceedings of the 17th ACM International Conference on Computing Frontiers. pp. 182–191 (2020)
19. Ma, Z., Catthoor, F., Vounckx, J.: Hierarchical task scheduler for interleaving subtasks on heterogeneous multiprocessor platforms. In: Proceedings of the 2005 Asia and South Pacific Design Automation Conference. pp. 952–955 (2005)
20. Morais, L., Silva, V., Goldman, A., Alvarez, C., Bosch, J., Frank, M., Araujo, G.: Adding tightly-integrated task scheduling acceleration to a risc-v multi-core processor. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 861–872 (2019)
21. Nickolls, J., Buck, I.: NVIDIA CUDA software and GPU parallel computing architecture. In: Microprocessor Forum (2007)
22. OpenACC: OpenACC: Directives for accelerators (2015)
23. OpenMP: OpenMP reference (1999)
24. Robison, A.D.: Composable parallel patterns with Intel cilk plus. Computing in Science & Engineering **15**(2), 66–71 (2013)
25. Ronneberger, O., Fischer, P., Brox, T.: U-net: Convolutional networks for biomedical image segmentation. In: International Conference on Medical image computing and computer-assisted intervention. pp. 234–241. Springer (2015)
26. Shao, Y.S., Xi, S.L., Srinivasan, V., Wei, G.Y., Brooks, D.: Co-designing accelerators and soc interfaces using gem5-aladdin. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 1–12. IEEE (2016)
27. Sijstermans, F.: The NVIDIA deep learning accelerator. In: Proceedings Hot Chips: a symposium on high performance chips (August 2018)
28. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
29. Sinnen, O.: Task scheduling for parallel systems, vol. 60. John Wiley & Sons (2007)
30. Själander, M., Terechko, A., Duranton, M.: A look-ahead task management unit for embedded multi-core architectures. In: 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools. pp. 149–157. IEEE (2008)
31. Slaughter, E., Wu, W., Fu, Y., Brandenburg, L., Garcia, N., Kautz, W., Marx, E., Morris, K.S., Lee, W., Cao, Q., et al.: Task bench: A parameterized benchmark for evaluating parallel runtime performance pp. 1–30 (2020)
32. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 2818–2826 (2016)

33. 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)
34. Ullman, J.D.: NP-complete scheduling problems. Journal of Computer and System sciences **10**(3), 384–393 (1975)
35. Vetter, J.S., Brightwell, R., et al.: Extreme heterogeneity 2018: DOE ASCR basic research needs workshop on extreme heterogeneity (2018). https://doi.org/10.2172/1473756
36. Waterman, A., Lee, Y., Avizienis, R., Cook, H., Patterson, D.A., Asanovic, K.: The RISC-V instruction set. In: Hot Chips Symposium. p. 1 (2013)
37. Western Digital Corp.: RISC-V: Accelerating next-generation compute requirements. https://www.westerndigital.com/company/innovations/risc-v, accessed: 2020-12-10