# Energy Efficient Task Graph Execution Using Compute Unit Masking in GPUs

Marcus Chow
*Department of Computer Science*
*University of California Riverside*
CA, USA
mchow009@ucr.edu

Kiran Ranganath
*Department of Electrical and Computer Engineering*
*University of California Riverside*
CA, USA
krang006@ucr.edu

Robert Lerias Jr.
*Department of Computer Science*
*University of California Riverside*
CA, USA
rleri001@ucr.edu

Mika Shanela Carodan
*Department of Computer Science*
*University of California Riverside*
CA, USA
mcaro008@ucr.edu

Daniel Wong
*Department of Electrical and Computer Engineering*
*University of California Riverside*
CA, USA
danwong@ucr.edu

***Abstract*—The frontiers of Supercomputers are pushed by novel discrete accelerators. Accelerators such as GPUs are employed to enable faster execution of Machine Learning, Scientific and High-Performance Computing applications. However, it has been harder to gain increased parallelism in traditional workloads. This is why more focus has been into Task Graphs. AMD's Directed Acyclic Graph Execution Engine (DAGEE) allows the programmer to define a workload in fine-grained tasks, and the system handles the dependencies at the lower-level. We evaluate DAGEE with the Winograd-Strassen Matrix Multiplication algorithm and show that DAGEE achieves on average 15.3% speed up over the traditional matrix multiplication algorithm.**

**While using DAGEE this may increase the contention among kernels due to the increased amount of parallelism. However, AMD allows the programmer to set the number of active Compute Unit (CU) by masking. This fine-grain scaling allows the system software to enable only the required number of Computation Units within a GPU. Using this mechanism we develop a Runtime that masks CU's for each task during a task graph execution and partitions each task into their separate CU's, reducing overall contention and energy consumption. We show that our CU Masking runtime on average reduces energy by 18%.**

***Index Terms*—DAGEE, Windograd-Strassen, Task Graph Execution, Compute Unit Masking, Resource Partitioning, Energy Efficient**

## I. Introduction

The process of executing kernels on a GPU have largely remained the same since the beginning of general purpose execution. Device kernels are comprised of a grid of Thread Blocks and each thread block is scheduled to a Compute Unit (CU) on the GPU. The host produces and enqueues a kernel in a stream, while the gpu consumes and dequeues from the same stream, one after the other. Improvements have been made, such as, concurrent execution of kernels from separate streams and device side kernel launches, but the limitations in host to device streams remain unaddressed. This means, if a programmer wants to increase parallelism across kernels, they must ensure dependencies are met across streams, increasing the complexity of their code and burdens the programmer to know all the nitty-gritty details of architecture and system stack.

AMD's Direct Acyclic Graph Execution Engine (DAGEE) [1] offers a novel programming paradigm, through tasked based execution. Here the programmer only need to specify the nodes and edges of a task graph, and the library enforces dependencies in the driver level queues. However, with increased kernel concurrency also affects hardware utilization. This is due to Thread blocks of various kernels contending for the same CU. This may potentially be alleviated through Compute unit Masking thereby improving energy efficiency of the system. CU Masking is a technique that allows a programmer to mark off which CUs a kernel is able to execute on.

This work aims to show DAGEE is a viable execution paradigm and combined with CU Masking, can lead to increased utilization and energy efficiency of the GPU. Our paper makes the following contributions:

1) Implementation and performance analysis of Winograd-Strassen Matrix Multiplication algorithm using DAGEE.
2) Power and Energy Characterization of Resource Partition through Compute Unit Masking Policies.
3) Implement Compute Unit Masking at Runtime for Task Graph Executions

## II. Task Graphs

Works like [2] popularized dataflow programming models. A workload is implemented as an acyclic graph as shown in Figure 1. In this implementation, the nodes are the computation operations such as CPU functions and GPU kernels. The edges represent the dependencies among the tasks. The number of required active threadblocks will depend on number of tasks that are have their dependencies successfully met. More recent works like [1], [3] proposed novel data-flow programming paradigms to exploit heterogenous architectures like CPUs

and GPUs in systems such as Summit [4], and Exascale Supercomputers such as Frontier [5] and El Capitan [6].

**Benefits of task graphs:** By dividing the workload into taskgraphs, we can effectively control the granularity of necessary compute requirements. Thereby assist in making attuned decisions on allocation to improve speedup and energy efficiency of the system. In the next section, we shall discuss Compute Masking could help in saving energy while executing tasks of different compute intensities.

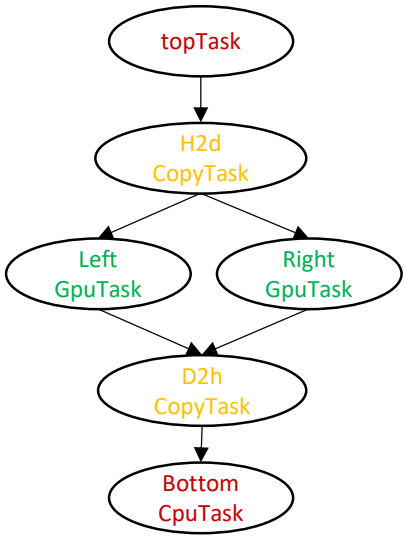



Fig. 1: Sample DAGEE Task Graph

## III. Directed Acyclic Graph Execution Engine (DAGEE)

DAGEE [1] is a C++ library that provides a simplified interface to implement applications as task graphs as described in Section section II. The nodes in the task graph can be a computation or a memory movement, while the edges represent the dependencies between tasks.

DAGEE is high-level Programmer API that utilizes AMD's C-runtime library called Asynchronous Task and Memory Interface (ATMI) runtime under the hood. ATMI is internally calls ROCm stack. ROCm has all the low-level device drivers, queues, and Data Structures necessary to launch and execute computation on AMD GPUs and CPUs.

Let us assume that we we have an application which has two CPU, two GPU and two memory copy tasks respectively. The dependencies among all these tasks is as shown in Figure 1. Figure 2 illustrates the implementation of Figure 1 in DAGEE. We observe that the implementation involves four stages.

1) **Initialize executors:** In this step, we instantiate the resources that are necessary to successfully execute the task graph. The resources can include CPU, GPU, and Memory Engines.
2) **Register GPU Kernels and CPU functions:** We register the function and kernel pointers with the respective CPU or GPU executor.

```
void dagee_example()
{
  // Initialize Executors
  dagee::CpuExecutorAtmi cpuEx;
  dagee::GpuExecutorAtmi gpuEx;
  dagee::MemCopyExecutorAtmi memEx;
  auto dagEx = dagee::makeMixedDagExecutor(cpuEx, gpuEx, memEx);
  auto *dag = dagEx.makeDAG();

  // Register CPU functions and GPU kernels
  auto initCpu = cpuEx.registerKernel<...>(&initFuncCpu);
  auto comptueGpu = gpuEx.registerKernel<...>(&computeKernelGpu);
  auto finalCpu = cpuEx.registerKernel<...>(&finalFuncCpu);

  // Create Task Nodes
  auto topCpuTask = dag->addNode(cpuEx.makeTask(initCpu,...));
  auto h2dCopyTask = dag->addNode(memEx.makeTask(src, dest, size));
  auto leftGpuTask = dag->addNode(gpuEx.makeTask(comptueGpu,...));
  auto rightGpuTask = dag->addNode(gpuEx.makeTask(comptueGpu,...));
  auto d2hCopyTask = dag->addNode(memEx.makeTask(src, dest, size));
  auto bottomCpuTask = dag->addNode(cpuEx.makeTask(finalCpu,...));

  // Specify dependency
  dag->addEdge(topCpuTask, h2dCopyTask);
  dag->addFanOutEdges(h2dCopyTask, {rightGpuTask, leftGpuTask});
  dag->addFanInEdges({rightGpuTask, leftGpuTask}, d2hCopyTask);
  dag->addEdge(d2hCopyTask, bottomCpuTask);
  dagEx.execute(dag);
}
```

Fig. 2: Sample DAGEE program

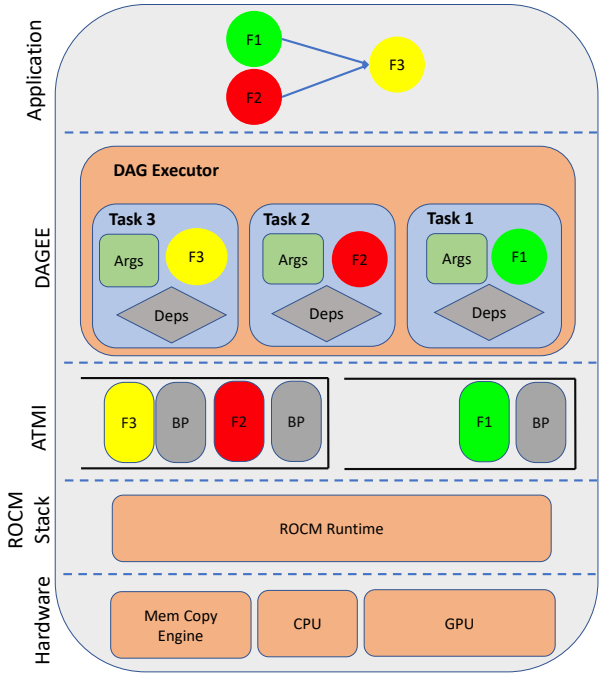



Fig. 3: Software stack for implementing DAGs in AMD GPUs. The application defines the task graph using DAGEE's API, which then wraps the individual functions with their arguments in a task. The tasks and their dependencies get passed to ATMI which puts them into their respective queues,along with their barrier packets to enforce dependencies, that the rocm stack cam dispatch to the hardware.

3) **Create Task Nodes:** Each task will include information regarding the registered kernel and necessary function

arguments to successfully execute the task including kernel launch parameters.

4) **Specify Task dependency:** The dependency will stall a task node from execution until all the dependencies in the previous step is met. These dependency information is essential since it provides the insight into the required parallelism and hardware requirements to make informed decisions to improve performance and energy efficiency before the task is dispatched on to the system.

### A. System Stack

The high-level overview of DAGEE Software Stack is shown in Figure 3. The application is implemented as a task graph. The tasks can be CPU, GPU or Memory copy task. We know from Figure 2 that each node in the task graph is created with a DAG executor. Each task will consist of a Function pointer along with the required function arguments and required task dependencies. DAGEE internally uses Asynchronous Task and Memory Interface (ATMI) to effectively manage task queues and launch tasks. ATMI dispatches the ready tasks onto the AMD hardware resources through ROCm [7] as the dependencies are met. ATMI uses Barrier Packets (BP) to effectively enforce dependencies that are set by DAGEE. Once the kernel and barrier packets are enquired in their respective queues. The ROCm Runtime dispatches the packets to the underlining hardware, either being the Memory Copy Engine, CPU, or GPU.

## IV. Compute Unit Masking

A compute unit (CU) is the GPU's core hardware unit. AMD allows programmers to control which CU's are active for thread block scheduling for each specific queue. This is done through the HSA runtime API $hsa\_amd\_queue\_cu\_set\_mask(queue, size, mask)$ This function call sets the active CU's all kernels that are dispatch to this queue. Each

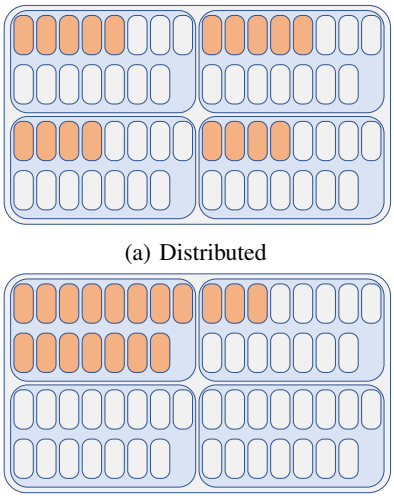
Compute Unites are then grouped together into a cluster called a Shader Engine (SE). These clusters are hidden from the programmer, they impact performance and power usage depending on how they are activated as discussed in subsection IV-A. This leads to two masking policies; *Distributed* and *Packed*

**Distributed**: This policy aims to balance active CU across Shader Engines. This allows for minimal contention between thread blocks within a single kernel.

**Packed**: Activates CUs within a single SE before activated another SE. The goal of this packed is to keep as many SE engines unoccupied as possible, leaving space reducing contention between concurrent kernels.

### A. Power and Energy Characterization

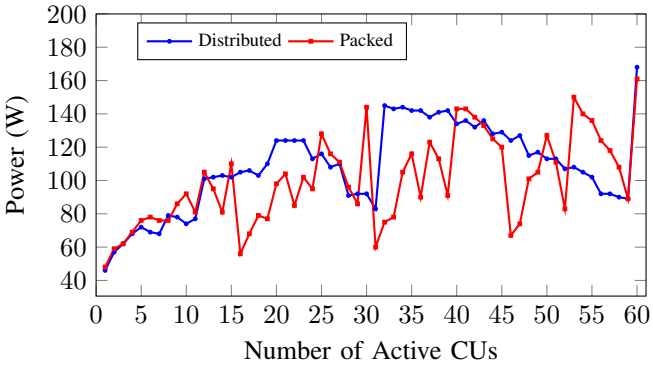
To characterize power and energy consumption of different number of active CUs, we run a simple matrix multiplication kernel that uses 128 thread blocks, to make sure that the full GPU is occupied, while measuring the execution time and average power consumption.



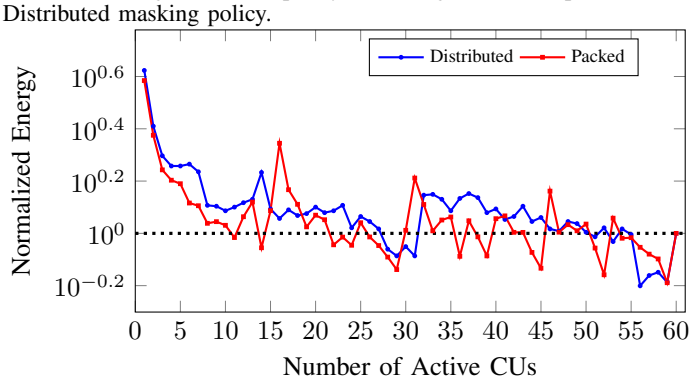
(a) Distributed

(b) Packed

Fig. 4: AMD MI 50 Has 4 Shader Engines with 15 CUs each, for a total of 60 CUs. Figure shows 18 active CUs for Distributed and Packed Policies




(a) Power Consumption. The difference between power curves indicated that there is some form of power gating involved within each Shader Engine. Packed policy, on average, used less power than Distributed masking policy.




(b) Normalized Energy to 60 active CUs. This shows that reducing the number of CUs reduces the overall energy in both Distributed and Packed Policies. Indicated by all points below the dotted line.

Fig. 5: CU Masking Power and Energy analysis

Figure 5a Shows the power consumption for distributed and

packed polices. For Distributed we see large dips at 31 and 59 active CUs. At 31 CUs, each shader engine has half of its CUs active. We believe this dip in power is due to power gating within the shader engine. Where the first and second half's are on a separate power gate. We see the same dips in the packed policy at 7, 22, 37, and 52 active CUs. This shows that it is possible to isolate concurrent kernels to individual SEs and, as long as they each use half of the SE, will still reduce total power consumption. However, for a single kernel it is still more efficient to use packed policy. We also notice that the larger power savings come from the higher number of active CUs, at lower active CUs the power savings diminish. Indicating that we should not have only a few CUs active for the best efficiency.

Although CU Masking may reduce power consumption, it also impacts the execution time due to limited compute resources. Therefore, In Figure 5b, we show the normalized energy of active CUs with respect to all 60 active CUs. We highlight the normal energy usage by the dashed line. Here, we see reducing the number of active CUs has the potenial to save energy overall, with energy savings up to 46%. Saving diminish exponentially as the number of active CUs decreases, due to the performance impact of reduced compute resources being greater than the power savings. However, even at 15 active CUs, energy is reduced by 12% in packed policy. This allows us to separate each SE into its own "logical" GPU, in which we explore in the following section.

## V. Compute Unit Masking during Task Graph Execution

The last section we described the mechanism that allows us to Mask CUs for a single kernel. However, when using DAGEE, multiple kernels are allowed to be queued up in various queues as long as their dependencies are met and dependencies are enforced using barrier packets. Therefore, CU masking is unable to work at the application level because the user has no control over when a kernel is dispatched.

### A. Compute Unit Runtime

For our Compute Unit Masking Runtime, we modified ATMI so that when a kernel packet is dispatched to a queue, we dispatch two extra barrier packets in front to the kernel packet to enforce CU Masks. In the first barrier packet, we assign a callback function that executes once that packet is consumed by the runtime. The runtime then allocates the mask for the upcoming kernel packet. We use a round robin scheduling for both packed and distributed and packed policies. The second barrier packet is assigned a dependency that enforces the kernel to wait until the cu mask callback function is finished executing, which it signals using $hsa\_signal\_store\_relaxed(signal, 0)$. Figure 6 describes our runtime and CU partitioning for four tasks.

For our Round Robin scheduler, we split the GPU into four groups for both packed and distributed masking policy. We decided to use four groups because the MI50 has four shader engines which lowers the complexity. We leave evaluation of more complex schedulers to future work.

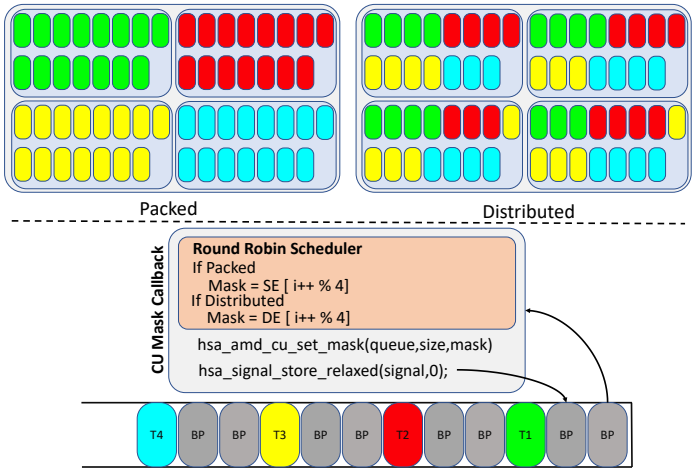



Fig. 6: (Below) Two barrier packets are required to set cu masks per kernel packet. The first packet is assigned a call back handler that sets the cu mask on its callback function. The second is make sure the kernel packet is not dispatched before the cu mask is set. (Above) We show how CUs are partition among four separate tasks for packed and distributed policies

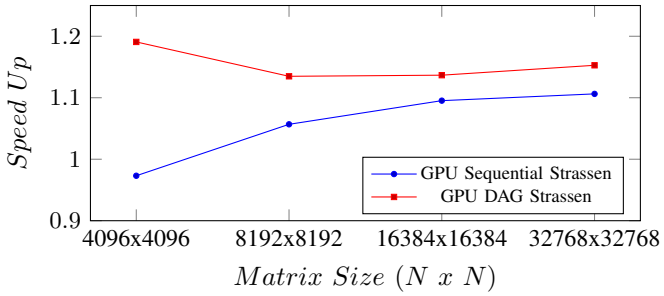



Fig. 7: Speed Up observed with different sizes of Matrix Multiplication operation. The results are normalized against standard GPU Matrix Multiplication.

## VI. Evaluation

For our evaluation methodology we use the compute unit masking runtime described in Figure 6 on a system equipped with AMD EPYC 7302 and MI50 GPUs, using ROCM v4.1.1. We breakdown our evaluation into two parts, Performance and Energy Efficiency.

### A. Winograd Strassen Algorithm

Winograd-Strassen is an combinatorial matrix multiplication algorithm that breaks down a matrix into a series of steps. Each step computing on a portion of the final matrix. These steps reduce the overall complexity from $O(n^3)$ to $O(n^{2.81})$. Not only does it reduce the number of operations, there is an increase of addition operations and a decrease in multiplication ones.

This means in practice, the algorithm is faster due to the short latency of addition operations compared to multiplication.

We use 3 workloads to evaluate this work - standard Matrix Multiply algorithm, Sequential Strassen, and DAG Strassen. Sequential Strassen, performs the algorithm sequentially while DAG Strassen exploits the parallelism in the dag in Figure 8. We run experiments for matrix sizes 4096x4096 to 32768x32768. We do not show the results for matrix sizes smaller than 4096x4096 since there isn't enough parallelism that is necessary, the initialization overhauls the execution time.

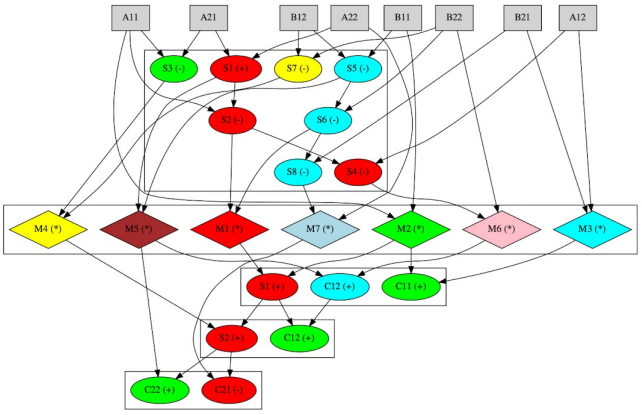


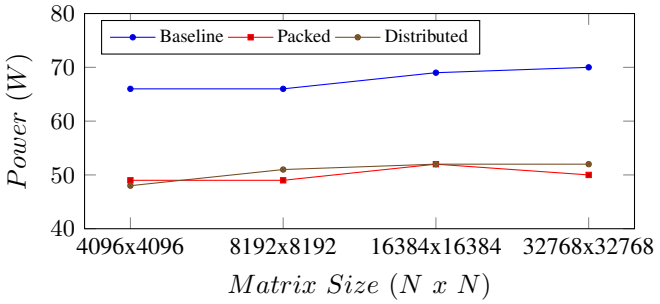

Fig. 8: Figure from [8]. This dependency graph is used as a reference to implement Strassen Matrix Multiply in DAGEE.

### B. DAGEE Performance

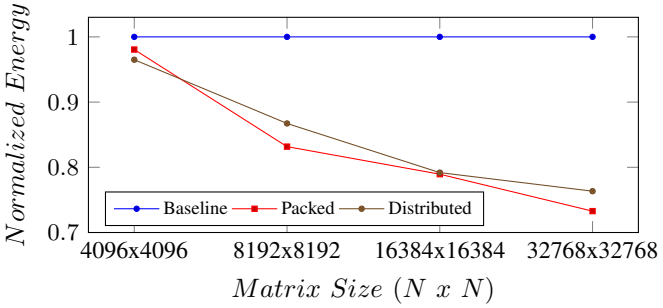
On average we see a speed up of 5.8% for sequential Strassen and 15.3% speed up for DAG Strassen as shown in Figure 7, without our CU Masking Runtime. This shows the benefit of implementing algorithms in DAGEE to exploit their task level parallelism. However, for both Sequential and DAG Strassen, we do not see the performance benefits for smaller sizes. This is due to the difference in initialization costs in sequential and DAGEE implementations, therefore we do not show those results. Sequential Strassen still sees a benefit due to its lower complexity and use of more $addition$ operations than $multiplaction$ operations.

### C. CU Masking Runtime Energy Analysis

Next, we evaluate our CU Masking runtime and Round Robin Scheduling for power and energy savings in Figure 9. On average the Baseline uses 67W while both packed and distributed use only 50W of power. The power remains constant for all sizes. This is due to the fact that we only show sizes that fully utilize the GPU. CU Masking uses much less energy because each kernel has its own set of CUs, reducing the amount of contention across kernels. This reduction in power is one of the main reasons for the energy savings seen in Figure 9b. On average, Packed has a 18% reduction of energy compared to 16% for distributed. Again, it is more efficient to pack the same kernel within a single CU if possible. Also, larger matrices benefit more from CU Masking,




(a) Power consumed during Task-based Strassen Matrix Multiply. Using CU Masking Policies greatly reduces the average power consumption for larger matrices.




(b) Normalized Energy to 60 active CUs. Overall, Packed policy uses less energy than Distributed, this is due to less contention within a Shader Engine.

Fig. 9: CU Masking Power and Energy analysis.

## VII. Related Works

There has been previous works exploring dependency graph execution and resource partition on GPUs. An overview of these works is presented here.

**Execution Graphs on GPUs:** Recent works [1], [3], [9] have proposed programming models and runtimes to implement GPU-based execution graphs. LC-MEMENTO [9] proposed novel memory models to efficient synchronization of data blocks and task placement on heterogeneous systems with CPUs and GPUs. DAGEE [1] is newly released task-based runtime, we have not been able to find work using DAGEE as the programming model. However, Nvidia has a graph execution library called Cuda Graphs[10], which is only supported on Nvidia GPUs. We evaluate our work with DAGEE on AMD GPUs.

**Winograd-Strassen Algorithm:** Previous work have implemented parallel versions of winograd-strassen on Nvidia GPUs using concurrent streams.[8][11] Our work implements the algorithm using a DAG, showcasing the performance benefit of tasked based execution on AMD GPUs.

**GPU Resource Partitioning:** Resource partitioning using AMD's CU Masking has been shown to increase performance for multi-tenancy in GPUs [12], [13]. We aim to showcase that

CU Masking can be a viable technique for energy savings in tasked based applications.

**Efficient GPU scheduling:** Coarse-grain efficient scheduling of multiple GPUs was demonstrated to be beneficial in [14], [15]. Similarly, there are works that improved performance within a single GPU with effective scheduling of kernels and threadblocks [16], [17], [18], [19], [20], [21]. However, these works do not consider the need to enforce effective implementation methodologies to enable effective execution of workloads in System stack. Although LC-MEMENTO [9] does propose novel programming models to support heterogenous task-graph applications, they do not consider the effects of effective fine-grain utilization of Compute Units.

## VIII. Conclusion & Future Works

In this work, we demonstrated the benefit of using Task Graphs to improve Performance while using CU Masking for resource partitioning to improve energy efficiency using the Winograd-Strassen Matrix Multiplication algorithm. We showed that by allowing concurrent execution of tasks, we could increase the speedup of Matrix multiply by 15.3%. Additionally, we were able to save up to 18% energy by using CU-masking techniques to activate only the Compute Units required to fulfill the 'ready' tasks.

### A. DAGGEE with Larger workloads

This paper uses Matrix Multiplication as the only workload. However, it is necessary to identify adoption of DAGEE and CU-masking techniques in other workloads such as Machine Learning (ML). We plan on integrating DAGEE along with DAG Matrix Multiplication, DAG Pooling, DAG Reduction, DAG convolution, and other popular linear algebraic functional layers into tensorflow and pytorch to evaluate performance and energy efficiency of future ML systems.

### B. Task speculation

Furthermore, The task graph-based DAGEE programming model can also be used to speculate critical execution path and memory footprint to improve caching, paging, and task placement.

### C. CU Partitioning Policies

In this work, we only evaluate a naive round robin CU Masking Partitioning scheme. This was to show that it is possible to save energy during task graph execution. However, it may be possible to increase energy efficiency through a more complex approach. Our method equally partitioned the GPU into four equal groups. However, it is possible to make each group have a different number of active CUs. We can base the number of CUs given to a task on number of thread blocks launched, type of function being run, and number of available CUs. These methods may improve the performance and energy efficiency of task graph execution.

## References


[1] A. M. D. (AMD), "Directed acyclic graph execution engine," 2021. [Online]. Available: https://github.com/AMDResearch/DAGEE

[2] J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-flow Processor," in *Proceedings of the 2Nd Annual Symposium on Computer Architecture*, ser. ISCA '75. New York, NY, USA: ACM, 1975, pp. 126–132. [Online]. Available: http://doi.acm.org/10.1145/642089.642111

[3] P. N. N. L. (PNNL), "Abstract runtime systems," 2021. [Online]. Available: https://github.com/pnnl/ARTS

[4] J. Hines, "Stepping up to summit," *Computing in Science & Engineering*, vol. 20, no. 2, pp. 78–82, 2018.

[5] O. N. Labs, "Fronteir exascale supercomputer," 2021. [Online]. Available: https://www.olcf.ornl.gov/frontier/

[6] L. L. N. Labs, "El capitan supercomputer," 2021. [Online]. Available: https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer

[7] A. M. D. (AMD), "Radeon open-compute software platform (rocm)," 2021. [Online]. Available: https://github.com/RadeonOpenCompute/ROCm

[8] P.-W. Lai, H. Arafat, V. Elango, and P. Sadayappan, "Accelerating strassen-winograd's matrix multiplication algorithm on gpus," in *20th Annual International Conference on High Performance Computing*. IEEE, 2013, pp. 139–148.

[9] K. Ranganath, J. Firoz, J. D. Suetterlein, J. B. Manzano, A. Marquez, M. Raugas, and D. Wong, "Lc-memento: A memory model for accelerated architectures," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2021.

[10] Nvidia, "Cuda graphs," 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs

[11] J. Huang, C. D. Yu, and R. A. van de Geijn, "Implementing strassen's algorithm with cutlass on nvidia volta gpus," *arXiv preprint arXiv:1808.07984*, 2018.

[12] N. Otterness and J. H. Anderson, "Amd gpus as an alternative to nvidia for supporting real-time workloads," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[13] ——, "Exploring amd gpu scheduling details by experimenting with "worst practices"," in *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, 2021.

[14] K. Ranganath, A. Abdolrashidi, S. L. Song, and D. Wong, "Speeding up collective communications through inter-gpu re-routing," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 128–131, 2019.

[15] K. Ranganath, J. D. Suetterlein, J. B. Manzano, S. L. Song, and D. Wong, "Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2021.

[16] A. Duţu, M. D. Sinclair, B. M. Beckmann, D. A. Wood, and M. Chow, "Independent forward progress of work-groups," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 1022–1035.

[17] A. Abdolrashidi, H. A. Esfeden, A. Jahanshahi, K. Singh, N. Abu-Ghazaleh, and D. Wong, "Blockmaestro: Enabling programmer-transparent task-based execution in gpu systems," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 333–346.

[18] A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong, "Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 600–611.

[19] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 3, pp. 1–26, 2021.

[20] D. Tripathy, A. Abdolrashidi, Q. Fan, D. Wong, and M. Satpathy, "Localityguru: A ptx analyzer for extracting thread block-level locality in gpgpus," *Proceedings of the 15th IEEE/ACM International Conference on Networking, Architecture, and Storage*, 2021 (To appear).

[21] D. Tripathy, H. Zamani, D. Sahoo, L. N. Bhuyan, and M. Satpathy, "Slumber: static-power management for gpgpu register files," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 109–114.