CASE: A Compiler-Assisted SchEduling Framework for Multi-GPU Systems

Chao Chen*[†]
chachenz@amazon.com
Amazon Web Service
Santa Clara, CA, USA

Chris Porter*
porter@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Santosh Pande
santosh.pande@cc.gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Abstract

Modern computing platforms tend to deploy multiple GPUs on a single node to boost performance. GPUs have large computing capacities and are an expensive resource. Increasing their utilization without causing performance degradation of individual workloads is an important and challenging problem. Although services such as NVIDIA's MPS allow multiple *cooperative* kernels to simultaneously run on a single device, they do not solve the co-execution problem for *uncooperative*, independent kernels on such a multi-GPU system. To tackle this problem, we propose **CASE** — a *fully automated compiler-assisted scheduling framework*. During the compilation of an application, **CASE** constructs GPU tasks from CUDA programs and instruments the code with a probe before each one. At runtime, each probe conveys information about its task's resource requirements such as memory and the number of streaming multiprocessor (SMs) needed to a user-level scheduler. The scheduler then places each task onto a suitable device by employing a policy appropriate to the system. In our prototype, a throughput-oriented scheduling policy is implemented to evaluate our resource-aware scheduling framework. The Rodinia benchmark suite and the Darknet neural network framework were used in our evaluation. The results show that, as compared to existing state-of-the-art methods, **CASE** improves throughput by up to 2.5× for Rodinia, and up to 2.7× for Darknet on modern NVIDIA GPU platforms, mainly due to the fact that it improves the average system utilization by up to 3.36× and the job turnaround time by up to 4.9×. Meanwhile, it limits individual kernel performance degradation within 2.5%. **CASE** achieved peak system utilization of 78% for Rodinia and 80% for Darknet on a 4×V100 system.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; **Software performance**; *Scheduling*; **Massively parallel systems**; **Compilers**.

Keywords: GPU, Scheduling, Compiler, High-performance computing

*Both authors contributed equally to the paper.
[†]This work was done when the author was a PhD student at Georgia Tech

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9204-4/22/04…$15.00
https://doi.org/10.1145/3503221.3508423

1 Introduction

General-purpose graphics processing units (GPGPUs) have become essential components in modern data centers and high-performance computing (HPC) systems. They provide the massive computing capacity required by modern machine learning and data analytics applications, or needed by large-scale high-fidelity scientific simulations. In the latest release (Jun 2021) of the TOP500 list [22], 60% of HPC systems are equipped with high-end GPU devices to deliver high-peak system performance. For many of these systems, multiple GPU devices are deployed in each computing node. An example is the Summit supercomputer, in which each compute node has 6 NVIDIA Tesla V100 GPUs.

However, how to efficiently utilize these high-power GPU resources remains an open research problem in many contexts. While certain performance-critical workloads may require dedicated GPUs and are able to fully saturate these high-end devices, many others do not utilize these resources continuously to their maximum capacities [25]. Per a discussion with scientists from Los Alamos National Laboratory, each of their scientific workloads typically only uses $\sim$ 30% of GPU resources because of “sequential-parallel" computing patterns and varying kernel sizes, leaving the majority of computing resources under-utilized and wasted. This trend was also observed in machine learning workloads in data centers [26]. One of the major reasons for under-utilization of resources is that over the generations, GPUs have tremendously grown in terms of their memory capacities and amount of compute resources (SMs). For example, in two recent generations, Nvidia V100 GPUs had 16/32GB amount of memory and 5,120 CUDA cores whereas A100s have 40/80GB memory and 6,912 CUDA cores. The problem is exacerbated because new generations of GPUs are expensive, both in terms of cost and power consumption. A high-end NVIDIA GPU device could cost as much as 2 to 5×

that of a high-end Intel Xeon CPU, and, in data centers, a GPU virtual machine instance could be 10× more expensive than a regular one. These practical observations demonstrate the necessity of efficient mechanisms to share GPUs among different workloads [6–8, 20, 23, 25, 26], thereby increasing utilization, saving on energy consumption, and improving the cost-efficiency as well as throughput for these systems.

In recent generations of GPUs (e.g. Volta), Multi-Process Service (MPS) helps mitigate this issue. It can facilitate co-execution of kernels from different processes. However, it is designed for *cooperative* applications (e.g. MPI jobs) on a single device. It also cannot schedule kernels across different GPU devices according to devices' statuses. Programmers' knowledge and efforts are required to schedule kernels from different MPI ranks and explicitly manage kernel-device pairings to avoid device overloading. Thus, even for a cooperative multi-process application, programmers have to explicitly designate the device for each kernel launch and its related CUDA operations via `cudaSetDevice`. If there is no such call in the application, the CUDA runtime will bind every CUDA operation to *device0* by default.

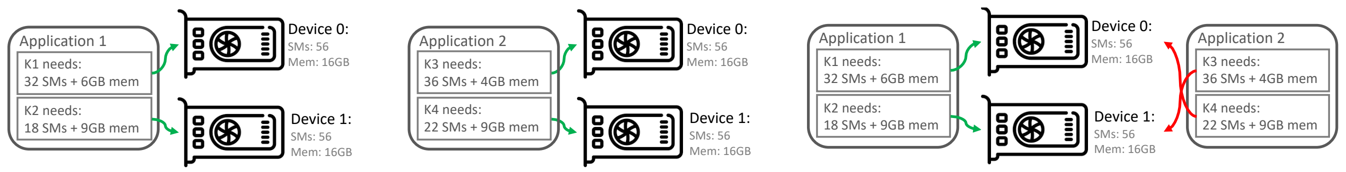
Making these scheduling decisions pre-runtime is not even a viable solution when applications from different users share the devices. Such *uncooperative* jobs have no knowledge of the resource requirements and dynamic concurrency of other executing processes. In such scenarios, some GPU devices could be significantly overloaded while others are idle. This could slow down the execution of individual applications, even though the overall system is quite under-utilized. Worse, if the memory capacity is exceeded (which MPS doesn't protect against), it could cause processes to crash due to "out-of-memory" (OOM) errors, which is disastrous for long-running applications. Thus, dynamic sharing of *multiple GPUs* (that reside inside a single, high-performance node) among *uncooperative and independent* workloads in an efficient and memory safe manner remains an unsolved problem.

1.1 Motivating Example

Figure 1 illustrates the issue on a 2-GPU system, where each GPU has 56 SMs and 16GB DRAM. It assumes there are 2 applications; each has 2 CUDA kernels that can be executed in parallel; and each kernel needs different GPU resources. If the system is dedicated to each application, it is easy to achieve good performance by simply mapping $k1$ to device 0 and $k2$ to device 1 for $application1$, and mapping $k3$ to device 0 and $k4$ to device 1 for $application2$. But each device is under-utilized under such an allocation. By closely examining the resource requirements for each kernel, one can see that it is possible to share the system between these two applications without performance degradation, since their total resource requirements are within the system capacity. However, the previous statically determined schedule (mapping) will not work in this shared scenario, because the total SM requirements of $k1$ and $k3$, and the total memory requirements of $k2$ and $k4$, exceed the capacity of a device. Though overloading SM resources could cause performance interference and degradation, overloading memory will lead to OOM errors and application failures. A good solution is to co-locate $k1$ with $k4$ on one device and $k2$ with $k3$ on another device. However, it is impossible to make such a decision statically, and a dynamic, runtime solution is proposed in this work. The proposed method manages GPU resources uniformly and allocates them at each kernel launch per request. Then the kernel will be scheduled to an appropriate device based on its resource requirements and the status of each device to ensure memory safety and minimize the performance interference among workloads.

1.2 The Proposed Solution

The above example and the limitations of MPS imply a need for system-level mechanisms to coordinate the execution of kernels from *uncooperative* processes (e.g. processes from different users) on a set of GPU devices, thereby increasing the resource utilization, saving on energy consumption, and improving cost-efficiency — all while incurring negligible performance interference for individual workloads. Our approach to this challenging problem is **CASE**, a compiler-assisted scheduling framework which uniformly manages and schedules GPU resources among uncooperative workloads. **CASE** is fully automated and works without any manual effort or changes to application source code. It leverages the compiler to construct GPU tasks, which are basic scheduling units for the runtime system. Briefly, a GPU task contains one or more kernel launches, as well as related GPU operations, e.g. memory allocations and initialization, that are required to execute the underlying kernel(s). A GPU task is generated through both static analysis by the compiler and a lazy runtime by bundling together all the kernels that share underlying memory or exhibit memory dependencies. Obviously, each GPU task contains a complete set of GPU operations required to finish a GPU computation, and thus it can be scheduled and executed on any GPU device without breaking its correctness. For each GPU task, a probe is statically inserted into its host-side code to gather and convey the task's resource requirements (such as memory footprints and number of SMs) to a user-level scheduler at runtime before the task is executed. The scheduler then dynamically places the GPU task on an appropriate device based on the task's resource requirements and the device status in terms of available memory and SMs. Different scheduling policies can be deployed in the proposed framework to target different computing environments. In this paper, we will focus on the design of the framework itself, applying it to a throughput scheduling policy. To the best of our knowledge, this is the first work that aims for a fully automated, efficient scheduling framework of a *multi-GPU* system among *uncooperative* applications.



(a) The system is dedicated to App1 **(b)** The system is dedicated to App2 **(c)** The system is shared by App1 & App2

Figure 1. Two uncooperative processes can be scheduled independently (a and b), but to boost system utilization and cost-efficiency, they require more than naive scheduling when sharing devices (c).

1.3 Contributions

In particular, this work makes the following contributions:

1. We propose *a GPU scheduling framework* to uniformly and transparently manage GPU resources for applications. It enables independent, uncooperative applications from different users to simultaneously execute on a set of shared devices safely (with no OOM errors) and with almost no performance degradation to each individual workload.
2. We devise a compiler-assisted method to construct GPU tasks, analyze their resource requirements (e.g. global memory and SMs), and insert probes to convey this information to the scheduler at runtime. The constructed GPU tasks can be dynamically bound to any GPU device at runtime, yielding fully automated task scheduling among devices.
3. We implement a prototype of **CASE** on top of NVIDIA GPUs, the CUDA library and the LLVM framework, and evaluate it in a throughput-oriented computing environment. An efficient and fast throughput-oriented scheduling policy is implemented to quickly schedule a GPU task on an appropriate device. By taking advantages of **CASE**-furnished details about tasks' resource requirements and the devices' statuses, the scheduler guarantees the task to be executed efficiently and safely (without OOM errors) and without overloading devices. A net result is that it improves the system throughput by over 2×, and system utilization by 1.59 ∼ 3.36×.

The rest of paper is organized as follows: Section 2 discusses the related state-of-the-art in terms of research. Section 3 and Section 4 present the design and prototype of **CASE**. Evaluation results are then presented in Section 5. Finally, Section 6 and Section 7 discuss the directions of our future work and conclude the work.

2 Related Work

The importance of GPU sharing is widely recognized in recent research. To the best of our knowledge, this work is the first to tackle this problem for generic workloads on multi-GPU systems in a fully automated manner with no user intervention, no OS or system changes.

Several frameworks [2, 14, 19, 23, 25, 29] are proposed to enable preemption on GPUs through kernel slicing. FLEP [25] is an example. It slices long-running kernels into multiple short-running sub-kernels. Thus GPU applications can be preempted when sub-kernel invocations are finished. However, these frameworks are designed to solve a different problem that occurs in a different system setting. They are mainly designed for single-GPU systems and tackle the problem of how to do effective preemption to regain resources (e.g. SMs) for scheduling higher priority processes. In contrast, **CASE** solves the problem of how to pack GPUs effectively across all devices in a multi-tenancy, multi-GPU system, which is not addressed by these systems. However, the idea of preemption proposed in FLEP can be coupled with our work to tackle latency-critical and QoS-sensitive applications.

Gdev [11] and PTask [17] are two OS-based approaches. They design a set of OS abstractions to integrate GPU runtime support into the OS and provide first-class GPU resource management schemes for multitasking systems. Similar to FLEP, they are designed for QoS-critical workloads, and only target a single device, which is a different setup as targeted by **CASE**. In addition, these methods would require significant changes to basic system software stacks, which is a major barrier to the adoption in production systems. PTask even needs a new programming model, implying a need of significant code changes to existing applications. In contrast, **CASE** offers full automation with no changes to an application or any part of the GPU software stack by providing a user-level scheduler providing a basis for a practical system.

SchedGPU [15] enables the sharing of a GPU device among independent workloads. It avoids OOM errors by interpreting memory requirements of workloads. SchedGPU differs from **CASE** in several ways. First, it requires the programmer to add library calls that pass the applications' memory needs. This process can not only be daunting but is also error-prone. Second, it takes into account only memory, which as we will show in the neural network experiments, can cause slowdowns if compute resources are not properly managed both within and across GPUs. Lastly and most importantly, it is again designed for a single-device environment, and only has the capability of suspending or continuing a CUDA operation. It has no ability to schedule tasks among devices to balance the device utilization, which is provided by **CASE**.

On a multi-GPU system, straight-forward device mapping is the widely utilized method to allocate GPU devices among applications. Slurm [27] is an example. It manages job queues and ensures that when an independent job runs on a node, the node is provisioned with a sufficient number of GPUs for that job. Kubernetes employs a similar approach. Going beyond this, Marble [9] attempts to find an optimal number of GPUs for each Deep Learning (DL) workload, based on their profiled scalability. Essentially, in these frameworks, each device is dedicated to only one workload. They do not address the problem of sharing devices among *independent* and *uncooperative* workloads. Gandiva [26] is a cluster-wide GPU scheduler for DL training workloads. It enables two different training workloads to run on the same set of devices without performance interference to each other. However, similar to other machine learning frameworks, e.g. mxnet [5], Gandiva heavily relies on DL properties, and cannot be applied to generic workloads such as Rodinia. **CASE** not only supports DL tasks but generic workloads, as well.

VirtCL [28] introduces a new abstraction layer atop OpenCL. For any applications that may share the system, VirtCL requires programmers to rewrite them using 5 abstractions. Each shared and non-shared object must be passed through an API which could be error-prone. **CASE**, on the other hand, is a fully automatic framework that works with off-the-shelf CUDA code with no changes. The two are also focused on different problems. VirtCL attempts to solve memory inconsistency with distributed shared memory (DSM) and device contention with a history-based scheduler. It implements DSM for OpenCL with a 6% overhead. CASE targets throughput (~2.5× gain) and device utilization (up to 3.36× gain) with negligible kernel slowdown (2-2.5%).

Finally, NVIDIA's multi-instance GPU (MIG) [13] is a new hardware feature in A100s to partition a large GPU into multiple physically isolated small GPUs. Each partition can be considered a small GPU device and can be assigned to an application. **CASE** can inter-operate with small changes with MIG especially in terms of packing jobs inside each of the MIG managed partitions, which is our future work. This is so since the key aspects that **CASE** relies on are the CUDA compute and memory needs generated by the probes, and the scheduling schemes can be adapted accordingly. Furthermore, where isolation among jobs is not a major requirement, under certain scheduling scenarios, **CASE** offers more flexibility and perhaps better *packing* possibility than MIG since there are no restrictions in terms of partitions that dictate co-execution of jobs. For example, on an A100 GPU (40GB), one can pack 13 jobs under MPS if each job needs 3GB, whereas it can only provide at most 7 partitions under MIG.

3 The Design of CASE

As shown in Figure 2, **CASE** consists of three main components: a compiler pass, lazy runtime, and scheduler. The

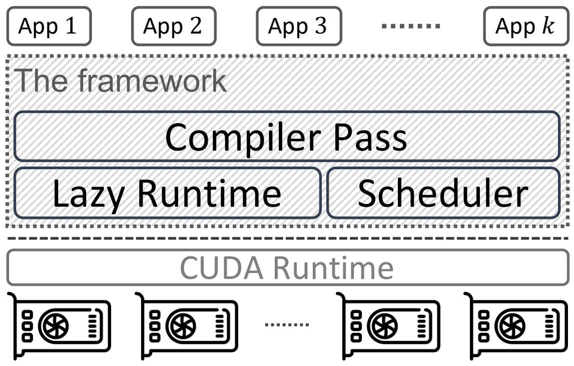


Figure 2. High-level framework design.

compiler pass, coupled with the lazy runtime, constructs GPU tasks and instruments applications with one probe per task. At runtime, the probes convey tasks' resource requirements to the scheduler before they execute. The scheduler employs a scheduling policy to assign GPU tasks to appropriate devices based on device statuses and probe data.

3.1 GPU Tasks

The "GPU task" is the basic scheduling unit in **CASE**. It is a collection of GPU operations, containing one or more kernel launches as well as a set of preamble and epilogue operations that are necessary to facilitate the correct execution context for the kernel(s). The preamble operations allocate (e.g. `cudaMalloc`) and initialize (e.g. `cudaMemset`) the memory space on the target device, and the epilogue operations save the computing results (e.g. `cudaMemcpy`) and free the allocated resources (e.g. `cudaFree`). All of these related GPU operations should be issued to the same device, therefore forming a GPU task. An example task is shown in Figure 3, in which the code from lines 22 ∼ 39 belongs to a GPU task for adding two vectors. The preamble operations (lines 22 ∼ 28) first allocate memory and prepare the data; then the kernel is launched (line 32) to do actual computations; and finally the epilogue operations (lines 35 ∼ 39) save the results and release the occupied resources.

3.1.1 Task Construction.

CASE leverages a compiler pass, coupled with the lazy runtime, to construct GPU tasks and gather their resource requirements. It works on the LLVM IR of applications, and therefore can support applications programmed with various programming languages supported by LLVM. Essentially, **CASE** builds GPU tasks by searching for kernel launches and related GPU operations leveraging the def-use chain information provided by the compiler. It first searches for kernel launches. In LLVM IR, they are heuristically implied by calls to `_cudaPushCallConfiguration`, followed by calls to host stub functions of kernels. Figure 4 (lines 6 ∼ 10) shows an example, which corresponds to the `VecAdd` at line 32 in Figure 3. For each kernel launch, the grid and block dimensions can be retrieved

```
// VecAdd is a CUDA kernel executed on GPU
__global__ void VecAdd(int *A, int *B, int *C) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  C[i] = A[i] + B[i];
}

// main is sequential code running on CPU
int main(int argc, char **argv) {
  int tid, A[N], B[N], C[N], *dA, *dB *dC;

  // initialize the vectors
  for (int i = 0; i < N; i++) {
      A[i] = cos(i);
      B[i] = sin(i);
      C[i] = 0;
  }

  // the instrumented probe
  tid = task_begin(N*3, 128, N/128);

  // allocate device memory
  cudaMalloc(&dA, N); // an input vector
  cudaMalloc(&dB, N); // an input vector
  cudaMalloc(&dC, N); // for storing result

  // initialize the device memory
  cudaMemcpy(dA, A, N, cudaMemcpyHostToDevice);
  cudaMemcpy(dB, B, N, cudaMemcpyHostToDevice);

  // launch the kernel on device
  dim3 T(128), B(N/128);
  VAdd<<<B, T>>>(d_A, d_B, d_C);

  // retrieve the result
  cudaMemcpy(C, dC, N, cudaMemcpyDeviceToHost);

  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
  task_free(tid);
}
```

Figure 3. An example GPU task, which consists of a kernel launch and related GPU memory operations.

```
...
%d_A = alloca float*, align
...
call cudaMalloc(float** %d_A, i64 %8)
...
%call = call _cudaPushCallConfiguration(i64 %g1,↩
     i32 %g2, i64 %b1, i32 %b2, i64 0, %struct.↩
    CUstream_st* null)
%a = load float*, float** %d_A, align 8
%b = load float*, float** %d_B, align 8
%c = load float*, float** %d_C, align 8
call VecAdd(float* %a, float* %b, float* %c)
...
```

Figure 4. The kernel launch in LLVM IR for VecAdd (simplified for better reading).

by examining the first 4 parameters of `_cudaPushCallConfiguration`. Then the compiler pass identifies involved GPU memory objects, which are pointer variables used by `cudaMalloc` calls, by walking backward up the def-use chain of each parameter of the kernel's host-side function, until it meets a terminating instruction, e.g. `alloca`. As an example, in Figure 4, the pass will visit `d_A` via `a`, and determine that `d_A` represents a GPU memory object since it is used in a call to `cudaMalloc`. Finally, the related preamble operations (e.g. `cudaMalloc`, `cudaMemcpy`) and epilogue operations (e.g. `cudaFree`) can be easily identified based on the def-use chain of pointers of memory objects, since they are taken as parameters to the calls of these runtime APIs. These steps are depicted by `constructGPUUnitTasks` in Alg. 1. It returns a set of unit tasks (represented by `GPUUnitTask`), with each unit task comprising exactly one kernel launch. Considering the fact that `cudaMalloc` operations always dominates other GPU operations on them same memory object, only `cudaMalloc` (indicated by `allocs`) are considered in Alg. 1, and their locations will used to derive the entry point of a GPU task.

In addition, many GPU tasks could share a set of memory objects. A typical example is a process executing two successive GPU kernels, $k1$ and $k2$, where the output of $k1$ (say, array `C`) is an input to $k2$. If $k1$ and $k2$ are scheduled onto two different devices, the data for `C` needs to be copied to the device running $k2$. To avoid the cost of such data movement, the framework schedules these two kernel launches on the same device by packing them into one GPU task. Therefore, further merge operations are applied to tasks that share memory objects (`constructGPUTasks` in Alg. 1). These unit tasks are combined into a large one (represented as `GPUTask` in Alg. 1). For convenience, all independent `GPUUnitTask`s will simply be converted to `GPUTask`s to have a unified task representation. Logically, the (minimal) code region containing all operations in a `GPUTask` is considered as a GPU task. **CASE** identifies the entry point and the end point of the region using the dominator information. Particularly, the lowest position in the control-flow-graph (CFG) of the program that dominates all operations in a `GPUTask` is selected as the entry point, and the highest point in CFG of the program that post-dominates all operations in the `GPUTask` is treated as the end point.

Finally, for a `GPUTask`, its memory and computing resource requirements are analyzed by examining every memory allocation operation (`cudaMalloc`) and kernel launch operation (e.g. `_cudaPushCallConfiguration`) inside the task. All of the analyzed information is presented in the form of symbols, and a probe is inserted at a program point which post-dominates all of these symbol definitions but dominates the entry point of the task. The probe takes these symbols as parameters and will interpret them at runtime to get actual resource requirements for each GPU task and convey them to the user-level scheduler. It summarizes memory sizes to get total memory requirements and utilizes the max grid and block dimensions as computing resources (the grid and block dimensions of the first kernel will be utilized if others are not available).

3.1.2 Lazy Runtime. Many applications encapsulate kernel launches and other GPU operations in separate functions, e.g. allocating GPU memory in `init()` and launching kernels in `execute()`. Static analysis cannot establish such def-use chains and domination relationships interprocedurally among GPU operations. To address this, the compiler first runs an inlining pass to minimize such cases, and then intra-procedurally analyses are performed to validate its effects. If the issue still exists, **CASE** will then defer the bindings of the memory operations to the lazy runtime.

The statically unbound operations are marked for lazy binding by the compiler. This enables the lazy runtime to record all such GPU operations and delay their bindings (executions) until a kernel launch. For example, a call to `cudaMalloc` will be replaced by the compiler with the `lazyMalloc`, which will simply assign a unique pseudo address for representing the memory object to be allocated, instead of performing the actual allocation. Thus, the subsequent CUDA operations on the memory object will see the pseudo address (and in fact all those CUDA operations are replaced with corresponding lazy runtime operations, as well). **CASE** leverages the pseudo address to track operations performed on each memory object. Specially, for each memory object (represented as a pseudo address), a queue is maintained by the lazy runtime to record GPU operations applied on it (e.g., [`cudaMalloc`, `cudaMemcpy`]) in execution order. Just before every kernel launch operation (e.g. `__cudaPushCallConfiguration`), a specific lazy runtime API `kernelLaunchPrepare` is inserted by the compiler. It will interpret the memory objects needed by the kernel, replay the recorded GPU operations for each of them, and replace their pseudo addresses with the real ones to ensure the kernel can be executed successfully. It also collects the resource requirements of the kernel launch by associating (or binding) them to the CUDA task being launched and conveys them to the scheduler. Note that these are the same operations as before, just with value substitutions during a short queue walk; thus there is negligible overhead to the kernel launch. Such an approach, coupled with the above static program analysis, binds full resource needs to a kernel, thereby converting it into a device-independent entity for the scheduler. The scheduler can then assign the task dynamically to a device and allocate the required resources recorded in the probes.

3.1.3 On-device Dynamic Allocation. In addition to global memory allocations, dynamic memory allocation from inside a kernel also needs to be considered. While it could be difficult to get accurate memory resources that will be allocated inside a kernel, it is easy to get the upper bound based on current GPU runtime and architecture design. The on-device heap size defaults to 8MB for the NVIDIA devices we tested. Applications can increase this limit by adjusting the `cudaLimitMallocHeapSize` via a call to `cudaDeviceSetLimit`; and this call must be placed before launching the kernel. Thus, the maximum heap memory size used by dynamic memory allocations inside a GPU is either statically bound to a CUDA task or dynamically intercepted and bound by the lazy runtime by analyzing the call to `cudaDeviceSetLimit`.

Algorithm 1 The pseudo code of constructing GPU tasks using static program analysis.

```
function buildGPUTasks($IR$)
    vector⟨GPUTask⟩ $Tasks$
    vector⟨GPUUnitTask⟩ $UnitTasks$
    $UnitTasks \leftarrow$ constructGPUUnitTasks($IR$)
    $Tasks \leftarrow$ constructGPUTasks($UnitTasks$) return $Tasks$
end function

function constructGPUUnitTasks($IR$)
    vector⟨GPUUnitTask⟩ $UnitTasks$
    for each kernel launch $l$ in $IR$ do
        $memObjs \leftarrow$ getMemArgs($l$)
        $allocs \leftarrow$ getAllocOps($memObjs$)
        $blocks \leftarrow$ getGridDims($l$)
        $threads \leftarrow$ getBlockDims($l$)
        $UnitTasks$.push($blocks$, $threads$, $allocs$, $l$)
    end for
return $UnitTasks$
end function

function constructGPUTasks($UnitTasks$)
    vector⟨GPUTask⟩ $Tasks$
    for each unvisited unit task $u1$ in $UnitTasks$ do
        set⟨CUDAUnitTask⟩ $Union$;
        $visited[u1] \leftarrow$ true
        for each unvisited unit task $u2$ in $UnitTasks$ do
            if $u1.memobjs \cap u2.memobjs \neq \emptyset$ then
                $Union$.insert($u1$, $u2$)
                $visited[u2] \leftarrow$ true
            end if
        end for
        if $Union$.size == 0 then
            $Tasks$.push($u1$)
        else
            $Tasks$.push(merge($Union$))
        end if
    end for
    return $Tasks$
end function
```

3.2 The Scheduling Framework

A user-level scheduler is deployed to place GPU tasks on appropriate devices based on their resource requirements (such as memory, CUDA cores, shared memory and execution time of a kernel). For applications, the scheduler exposes a simple API, `task_begin`, to indicate the beginning of a task. It is a synchronized API that blocks the process until it returns. The aforementioned compiler pass will automatically insert it at the beginning of each GPU task, and feed it with appropriate parameters, which contain the details about the resources required by the task, including the number of blocks, the threads per block, the total memory size [1], and the ID which is used by the runtime to uniquely identify the task. `task_begin` conveys this information to the scheduler, and then waits for the response from the scheduler. In return, the scheduler feeds this information along with the devices' status to the deployed scheduling policy, which finds an appropriate device for the task. The device ID is returned to the application, and then the `task_begin` directs the following GPU task to that device via specific mechanisms provided by the GPU runtime system. The task is suspended if no device has enough resources to host it. The corresponding end call is `task_free`, which takes the original ID as its parameter, and frees the resources required by the task. For `GPUTasks` constructed by the lazy runtime, these two APIs are called by the lazy runtime API accordingly. Based on this scheduling framework, different scheduling policies are designed and implemented for different computing environments. In this paper, we mainly focus on design of resource-aware scheduling framework, demonstrating its benefits with a throughput-oriented scheduling policy.

4 Prototype Implementation

We implemented a prototype of the proposed scheduling framework based on NVIDIA GPUs, CUDA-10.2 and LLVM-9.0. After the scheduler returns the ID of the device where the task will be executed, `task_begin` calls `cudaSetDevice` to map the task to the target device. For each GPU device, MPS is enabled so that **CASE** can schedule kernels from different processes to run on the same device, as long as the device has enough resources. If there is no device with enough resources for the requesting task, the prototype scheduler will put the task ID into a queue and not respond to the request until the needed resources are released. Since `task_begin` is a synchronized API, it will automatically suspend the application if it does not hear back from the scheduler. All communication between applications and the scheduler is implemented over shared memory.

A throughput-oriented scheduling policy that packs jobs onto devices geared towards batch jobs is implemented in our prototype. This policy demonstrates the advantages of **CASE** in terms of boosting device utilization as well as the system throughput. We choose a throughput-oriented scheduling policy because it demonstrates a dominant usage among workloads, such as ML training, data classification/analytics and linear algebra, which are very popular in modern HPC and clouds systems [1, 24]. Important analytics, data mining or ML is carried out by such workloads on data. For batch workloads, the fairness and QoS are not as important as the throughput of the system. Such systems aim to finish as many jobs as possible, therefore improving the system utilization and cost efficiency.

Algorithm 2 The pseudo code to select a GPU for a process, emulating the way hardware tracks SM usage.

```
function SCHED($task, GPUs$)
    $TargetG \leftarrow None$
    for $G$ in $GPUs$ do
        $TBs \leftarrow task.ThreadBlocks$
        if $task.MemReq > G.FreeMem$ continue
        while $TBs > 0$ do
            $availSM \leftarrow G.$GETNEXTSM($task$)
            if $!availSM$ break
            $availSM.$ADD($TB$); $TBs--$
        end while
        if $TBs == 0$ then
            $G.$COMMITAVAILSMCHANGES()
            $TargetG \leftarrow G$
            break
        end if
    end for
    return $TargetG$
end function
```

The prototype scheduling policy makes a decision based on a vector of metrics including the availability of global memory and SMs. Such a multi-resource oriented scheduling problem is NP-hard. In this paper, we look at two scheduling algorithms that are tailored specifically to the problem at hand. Alg. 2 emulates hardware's round-robin approach for placing a task's thread blocks across a GPU's SMs. It tracks exactly how many thread blocks and warps on each SM are available (taking into account the device's max thread blocks and warps per SM). It also ensures that the memory required by a task is available on the selected GPU. Both memory and compute are hard constraints in this algorithm. In contrast, Alg. 3 is simpler. It treats memory as a hard constraint, but it treats compute as a soft constraint (because it can impact performance but will not lead to a crash). It works by cycling over the GPU devices, checking two criteria as it goes. First, it checks if the memory requirement of an incoming task can be met on a particular GPU device. If it can, it checks if that device has the least compute load that it has come across so far. Compute load is in terms of number of warps

[1]The compiler pass automatically instruments the application with the code to compute total memory requirements by adding sizes of all memory objects accessed by the kernel.

Algorithm 3 The pseudo code to select a GPU for a process, with memory safety and quick placement based on max available warps.

```
function SCHED($task, GPUs$)
    $TargetG \leftarrow None$
    $MinWarps \leftarrow \infty$
    for $G$ in $GPUs$ do
        if $task.MemReq < G.FreeMem$ then
            if $G.InUseWarps < MinWarps$ then
                $MinWarps \leftarrow G.InUseWarps$
                $TargetG \leftarrow G$
            end if
        end if
    end for
    if $TargetG$ then
        $TargetG$.ADD($task$)
    end if
    return $TargetG$
end function
```

currently scheduled on the device. If both criteria hold, then it updates that current GPU to be the target for its task. In other words, it tracks in-use memory and active warps on each GPU, and picks the GPU with available memory and the least compute load. In terms of resource tracking, it may not be as accurate as Alg. 2, but it can take advantage of the soft compute constraint and clear the job queue much faster (because of its simplicity it has lower dynamic schedule calculation overhead) when Alg. 2 might have held back a job. Once a GPU is selected for a task, both the available memory and warp capacity of the GPU are updated. In Alg. 2, this is done with `G.CommitAvailSMChanges`, which is just a struct assignment (of changes to the `availSM` values and the task's memory needs to the actual state of GPU `G`); in Alg. 3, this is done with `TargetG.Add(task)`, which just adds the task's resource requirements to the state of GPU `TargetG`. Both scheduling algorithms are deliberately designed to be very simple to minimize the runtime overheads and to keep them dynamically reactive to short GPU jobs.

4.1 Currently Supported CUDA Features

Our prototype currently targets applications programmed based on core CUDA APIs, such as benchmarks evaluated in the paper. Advanced CUDA features, such as Unified Memory and Streams, are not currently supported. We think support for many of these features can be easily integrated which is our future work. Unified Memory requires users to utilize `cudaMallocManaged` instead of `cudaMalloc` to allocate GPU memory, such that CUDA's driver and hardware can automatically manage the data transfer between the GPU and system memory via page-fault handling at the cost of high performance overheads. There are two potential options for **CASE** to support the Unified Memory: 1) making the compiler pass and the lazy runtime library recognize calls to `cudaMallocManaged`. It would be similar to what we have done with `cudaMalloc`. In addition, a new flag would be added to the scheduling framework interface indicating that the tasks are using Unified Memory and that the memory "overflow" can be allowed; 2) designing and implementing a new compiler pass to automatically replace calls to `cudaMallocManaged` with ones to `cudaMalloc`. Appropriate calls to `cudaMemcpy` would also be instrumented into the application to ensure the compiled code is functionality equivalent to the original source code.

In addition, our current prototype also assumes it can dispatch each *GPUTask* onto any available devices. However, in some applications, programmers may choose to statically dispatch their kernels to a specific device (via `cudaSetDevice`) due to some user-specific reasons. **CASE** may assign the task onto other devices, which would be unexpected by such users. Our benchmarks do not cover such workloads, and they are not evaluated. In our future work, we will perform in-depth evaluations on such workloads, and improve the schedulers in **CASE** accordingly.

5 Evaluation

We evaluated **CASE** on two independent high-performance servers: Chameleon[2] and Amazon AWS. The Chameleon node consists of an Intel Xeon E5-2670 CPU, 128GB DRAM and 2 NVIDIA P100s. The AWS node (p3.8xlarge instance) is equipped with an Intel Xeon E5-2686 CPU, 244GB DRAM and 4 NVIDIA V100s. Each P100 has 16GB RAM, 3584 cores; each V100 has 16GB RAM, 5120 cores.

5.1 Methodology

For comparison, we implemented three other scheduling policies: *single-assignment (SA) scheduling*, *Core-to-GPU (CG) scheduling*, and *SchedGPU* which was introduced in [15]. We chose these three policies since they are representatives about how current systems utilize multi-GPU devices. The details for each of them are described in below:

SA shares the same scheduling strategy as provided in Slurm [27] and Kubernetes which are widely utilized in modern HPC systems and data centers. It distributes workloads among GPUs at process-level granularity. When a CUDA application begins, *SA* maps it to the first available GPU device. Each application has dedicated access to the assigned device during its lifetime. Each device has no more than one job at a time (assuring memory safety and avoiding performance interference), and no device sits idle once a request is made.

CG allows more than one process to share a GPU device via NVIDIA MPS, considering that a device could be extremely

[2]This is an experimental test-bed for computer science funded by the NSF Future Cloud program.

underutilized in *SA*. It behaves more closely to an MPI application, in which kernels from a group of MPI ranks can be scheduled on a device by a programmer. Programmers have to statically control the size of the group to assure memory safety and minimize the performance interference, based on their knowledge about applications. *CG* also attempts to statically control the maximum number of jobs per GPU through a pre-determined cpu-core-to-gpu ratio, but lacks knowledge about the uncooperative processes. It derives this ratio heuristically based on system configurations. For example, in a system with 12 CPU cores and 2 GPUs, each GPU device might serve kernels from no more than 6 cores (with 1 process per core), producing a ratio of 6:1. In our experiment, we examined multiple such ratios. At runtime, *CG* visits the GPU task queue in a round robin manner and maps the tasks to GPU devices until the ratio is met (in the above example, 6 tasks will be mapped per GPU device). Obviously, *CG* stands the risk of "out-of-memory" errors and crashes, since it has no knowledge of the memory requirements of the tasks.

SchedGPU is described in Section 2. It packs as many jobs as possible on the device, as long as the device has enough memory. It shares some similarity to **CASE** in terms of tracking memory requirements of each request, and suspends the process if the device does not have enough memory. Since *SchedGPU* is not open-source, we prototyped it according to the paper. We manually identified resource requirements of each kernel and modified each benchmark's source code to pass them to the daemon.

These schedulers are derived from either current practical systems or a state-of-the-art study. They are throughput-oriented and provide a good baseline for analyzing **CASE** to demonstrate the advantages of leveraging applications' knowledge of their resource needs. We do not compare **CASE** to others, e.g. FLEP [25], because they target QoS-sensitive workloads, do not handle the multi-GPU case or are not open-sourced. As pointed out earlier, they target a different problem and thus a comparison will be apples to oranges.

Our evaluation seeks to answer the following questions:

1) What are the throughput and system utilization improvements achieved by **CASE**?
2) How did **CASE** achieve these improvements from the perspective of job turn-around time (interval between the job arrival time and its completion time in the queue) and the memory-safe assurance?
3) What is the negative effect of **CASE** on an individual kernel's execution speed?

5.2 Results with Rodinia Benchmark Suite

In this section, we present the evaluation performed with Rodinia Benchmark Suite. We compared **CASE** with *SA* and *CG*. To emulate the job diversity in a throughput-oriented environment, we created a set of job mixes leveraging the CUDA benchmarks in the Rodinia suite v3.1 [3, 4]. In summary, 8 different job mixes were created from 7 Rodinia benchmarks as representative of modern workloads. These benchmarks include backprop (pattern recognition), srad-v1 and srad-v2 (image processing), lavaMD (molecular dynamics), needle (bioinformatics), dwt2d (image/video compression), and bfs (graph).

Various problem sizes were used for each benchmark in each job mix. Our mixes favor larger problem sizes to mimic realistic, heavy GPU kernels. In our setting, these benchmarks typically consume 1 ∼ 13GB memory footprints. We mark benchmarks with kernels that have over a 4GB memory requirement as "large". Those between 1 and 4 GB are considered small. Our mixes are created based on the ratio of large:small jobs. We have four different mixes: 1:1, 2:1, 3:1, and 5:1. Every mix matches one of these ratios with either 16 or 32 total number of jobs. The jobs are randomly chosen from their respective sets (large or small). Table 1 shows the arguments passed for each Rodinia benchmark. The table is ordered by increasing max kernel size of each benchmark. All Rodinia workloads are randomly generated from this initial list. Table 2 shows the details of the randomly generated job mixes; each job mix executes around 5 (for 16-job mixes) or 10 (for 32-job mixes) minutes.

We treat each job mix as a batch. All jobs from a job mix arrive at the same time, as opposed to arriving at predetermined or random times. The scheduler will dequeue a job from the batch and schedule it until all jobs in the batch get scheduled or all devices are fully occupied.

5.2.1 CASE Scheduling Algorithms: Comparison. In this subsection, we compare Alg. 2 and Alg. 3 introduced in Section 4. Figure 5 shows their throughput evaluated with 8 job mixes in a 4×V100 environment. On average, the throughput for Alg. 3 is 1.21× higher. (Note, for the absolute throughput numbers (jobs/sec) for this experiment and others in the evaluation, please refer to Tables 7 and 8 at the end of the section). We also scaled our experiments to 32-, 64-, and 128-job mixes, and observed similar improvements. Alg. 3 outperforms Alg. 2 mainly because of the extra time jobs wait for a GPU under Alg. 2. We observed a 30% increase in Alg. 2 in terms of job wait times. Alg. 2 ensures there is sufficient compute available before running each job, whereas Alg. 3 schedules jobs optimistically and sooner, taking advantage of fast completing jobs, even when compute is stressed. This confirms our intuition that it is better to design an imprecise but lightweight scheduler that dispatches a job quickly to a device than one which is precise but relatively slower to analyze job queue. The remaining evaluation is focused on Alg. 3, because it has better performance.

5.2.2 Throughput. Figure 6 compares the throughput of the schedulers. The results are normalized to *SA*. As compared to *SA*, **CASE** improved system throughput by 1.8 ∼ 2.5× (on average 2.2×) on P100s and 1.4 ∼ 2.5× (on average 2×) on V100s. This is mainly because **CASE** allows

Table 1. Rodinia benchmarks and their command line arguments, in order of increasing kernel size.

Benchmark	Command line arguments
backprop	8388608
bfs	data/bfs/inputGen/graph32M.txt
srad_v2/srad	8192 8192 0 127 0 127 0.5 2
dwt2d	data/dwt2d/rgb.bmp -d 8192x8192 -f -5 -l 3
needle	16384 10
backprop	16777216
srad_v1/srad	100 0.5 11000 11000
backprop	33554432
srad_v2/srad	16384 16384 0 127 0 127 0.5 2
srad_v1/srad	100 0.5 15000 15000
lavaMD	-boxes1d 100
dwt2d	data/dwt2d/rgb.bmp -d 16384x16384 -f -5 -l 3
needle	32768 10
backprop	67108864
lavaMD	-boxes1d 110
srad_v1/srad	100 0.5 20000 20000
lavaMD	-boxes1d 120

Table 2. Rodinia workload mixes.

Workload	Mix	Workload	Mix
W1	16-job,1:1-mix	W2	16-job,2:1-mix
W3	16-job,3:1-mix	W4	16-job,5:1-mix
W5	32-job,1:1-mix	W6	32-job,2:1-mix
W7	32-job,3:1-mix	W8	32-job,5:1-mix

Table 3. Percentage of crashed jobs for *CG* (P100s/V100s).

# workers	1:1 mix	2:1	3:1	5:1
3/6	0/0	3%/17%	8%/17%	0/0
4/8	14%/13%	6%/19%	6%/25%	9%/13%
5/10	13%/15%	13%/25%	20%/20%	22%/25%
6/12	16%/33%	17%/29%	16%/38%	16%/50%

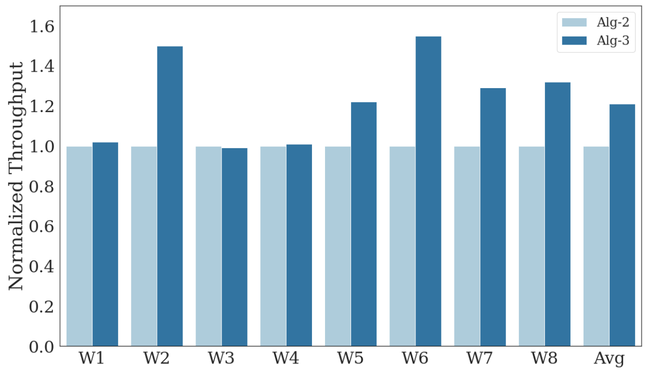
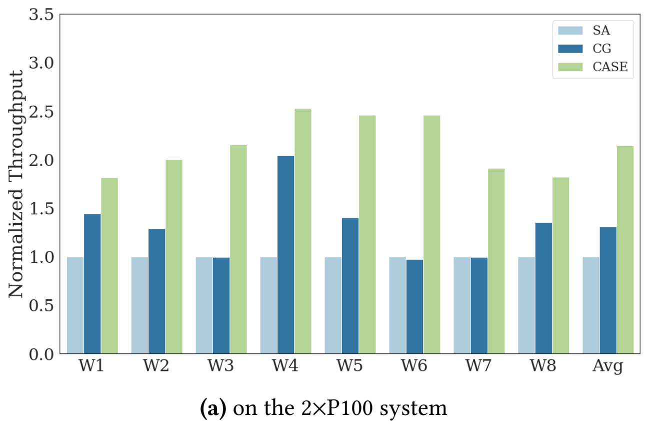


Figure 5. Throughput for Alg. 2 and Alg. 3 on a 4×V100 system (normalized to Alg. 2 for easy comparison, and the absolute throughput (jobs/second) for the baseline is in Table 7).

multiple kernels from different processes to be concurrently executed on the same device. Although *CG* also allows co-execution of kernels from different processes, **CASE** still outperformed *CG* throughput by an average of 64% on P100s and 41% on V100s. This is mainly because *CG* has no knowledge about the memory or SM requirements of workloads; therefore it could overload GPU devices and cause some jobs to crash due to memory safety violations. As shown in Table 3, the crash behavior of the *CG* scheduler was erratic. The expected trend (which is borne out in the table) is that as the number of workers increases, the chance for crashes under *CG* should increase; but there could be some exceptions due to many uncontrollable factors (various job sizes among large jobs, randomness of job arrival time, etc.). For example, in the 6-worker, 5:1 mix on V100s, the first 4 jobs happen to have 7.8GB, 10GB, 7.8GB, and 8.3GB footprints, which are spread across the 4 GPUs (each with 16GB memory); the next 2 jobs (assigned to the first 2 GPUs by *CG*) have 2.0GB and 4.7GB footprints. Thus, there is no OOM error, and the remaining schedule similarly allows all jobs to finish successfully. Nevertheless, for job mixes with large jobs, the percentage of crashes due to *CG* is alarming, ranging from 13% to 50% on V100s. Because of this, *CG* achieved similar or even lower throughput than *SA* for W6 and W7 in P100s and W3 and W6 in V100s. W1 in V100s is an exceptional case where *CG* happened to run efficiently without crashing, leading to higher throughput than **CASE**. This is because W1 has a 1:1 ratio of large:small jobs, and on this workload, *CG* was (coincidentally) able to pack the jobs without crashing but with lower runtime overheads than **CASE**.



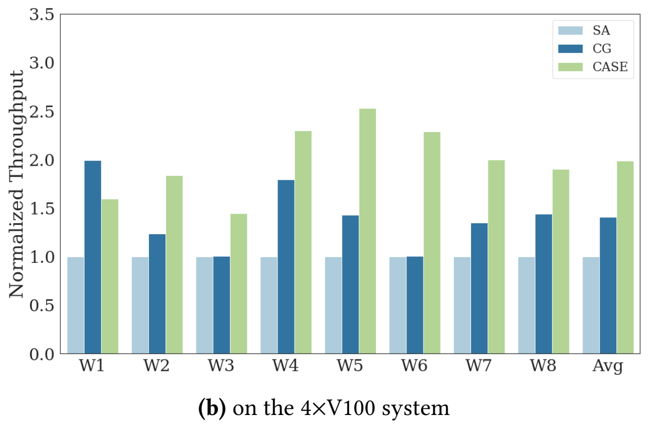


Figure 6. Throughput for *SA*, *CG*, and **CASE** (normalized to *SA*, and the absolute throughput (jobs/second) for the baseline is in Table 7).

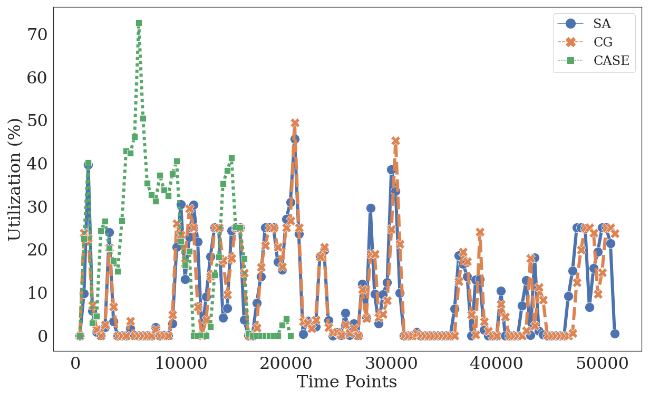


Figure 7. Utilization comparison among **CASE**, *SA* and *CG* using W7 from Rodinia on the 4×V100 system.

5.2.3 System Utilization Improvement. Figure 7 shows the system utilization of **CASE**, *SA* and *CG* on the AWS platform with 4×V100s for the W7 job mix. The NVML library is used to sample the device status every 1ms. The figure plots the average device (SM) utilization across all 4 V100 GPUs. As shown in the figure, **CASE** achieved significantly higher device utilization. The peak utilization for **CASE** is 78%, and for *SA* and *CG* it is 48%. The average utilization (across lifetime of the workload) is 23.9% for **CASE** vs. 9.5% for *SA* and 9.3% for *CG*.

5.2.4 Turnaround Time Speedup. Since the workload involves batch processing, the experiment begins with a queue already full of jobs. We view these jobs as requests, and measure the turnaround time for each job. While some degree of slowdown can happen when a particular job is co-executing with others, the turnaround time (time interval between the job arrival time and completion time) can be boosted by improving the throughput and reducing the time these requests sit in the queue. Table 4 shows the turnaround time speedups over *SA* for all mixes and workload sizes on both the P100s and V100s. We observed an average of 3.7× for the P100s and 2.8× for the V100s, and a maximum of almost 5× in some cases. The absolute job completion times average 236s and 122s for the P100 and V100s, respectively.

Table 4. Average job turnaround speedup for **CASE**.

GPUs	# of jobs	1:1 mix	2:1	3:1	5:1
2×P100s	16 jobs	4.9×	2.3×	4.9×	4.3×
2×P100s	32 jobs	4.6×	3.2×	3.6×	2.0×
4×V100s	16 jobs	2.4×	2.0×	3.5×	2.6×
4×V100s	32 jobs	3.8×	2.9×	2.9×	2.6×

5.3 Results with the Darknet Benchmarks

Due to the prevalence of deep learning, in this section, we present a study solely based on these types of jobs. In this evaluation, only the throughput, job turn-around time and the device utilization are examined. The off-the-shelf learning framework, Darknet [16], is utilized as the benchmark. Darknet provides several machine learning models, such as YOLO and RNN for both training and inference tasks. Its pre-trained models for image classification are competitive with popular networks like ResNet-50 [10] and VGG-16 [21] (in terms of top-1 and top-5 accuracy, GPU timing, and size); and as a framework it is also effective for creating other types of neural network tasks (such as RNN text generation). We ran 4 types of jobs: neural network training and prediction for image classification (CNN), real-time object detection (CNN), and text generation (RNN). For prediction, we used the pre-trained Darknet53-448x448 architecture and weights for the 1000-class ImageNet competition [18]; for training, we used the small architecture provided by Darknet for CIFAR-10 [12]; for real-time object detection, we used the pre-trained yolov3-tiny architecture and weights on the provided images; for RNN-based text generation, we used the pre-trained network based on Shakespeare's complete works.

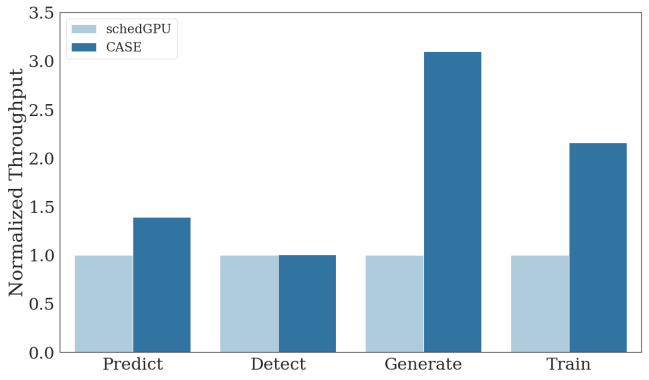


Figure 8. Throughput on homogeneous, parallel 8-job neural network workloads on 4×V100s (normalized to *SchedGPU*, and the absolute throughput (jobs/second) for the baseline is in Table 8).

Table 5 shows the command used for each Darknet task. All Darknet workloads in the evaluation are eight homogeneous jobs for a given task.

We ran two neural network experiments. In the first we compared **CASE** against *SchedGPU* [15], a state-of-the-art work for intra-node scheduling. We ran 4 homogeneous workloads (1 for each type of task described above), with 8 jobs in each workload. The memory size of each neural network is between 0.5-1.5GB, so 8 jobs can always fit within a single V100's memory. Since *SchedGPU* uses memory capacity as the only resource criterion in scheduling, this setting faithfully ensures that the *SchedGPU* can schedule all jobs to run on one device at a time for its best performance, since the memory capacity is not exceeded. Note that the Rodinia workloads (intentionally) exercise multiple GPUs, which *SchedGPU* cannot effectively handle (see Section 2 and the MPS discussion in Section 1). The Darknet experiments are designed, however, to allow *SchedGPU* to run each workload without queuing it, which results in a fair comparison with **CASE**.

Figure 8 presents the throughput of *SchedGPU* and **CASE** (normalized to *SchedGPU*). It shows that *SchedGPU* significantly under-performs on modern neural network loads at least those in these experiments, which have a high compute resource need in terms of warps or thread blocks. Because *SchedGPU* does not account for this resource, it is unable to spread work across GPUs and can easily oversaturate a GPU device. **CASE** achieves throughput speedups of 1.4×, 2.2×, and 3.1× over schedGPU for the predict, train, and generate tasks. For detection, the frameworks have similar results. This is because the real-time object detection network used in our evaluation utilized 25% or less GPU resources, so the compute units are not saturated in this case. The key takeaway is that single-GPU performance, even when it satisfies the simultaneous memory requirements of all running jobs, can and will suffer under common, modern machine learning

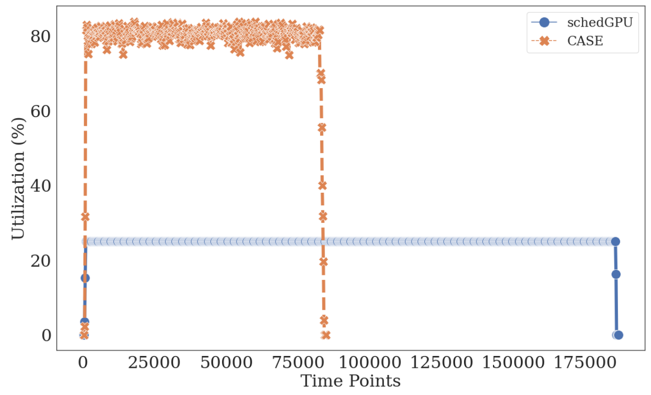


Figure 9. Utilization comparison between **CASE** and *SchedGPU* using 8 Darknet jobs on the 4×V100 system.

tasks. And memory requirements alone are not good enough to be used in a scheduler for better system performance. We also ran one large-scale experiment in a manner similar to our Rodinia setup, in order to verify that **CASE** is effective on large mixes of these neural network jobs. We ran a 128-job, random mix of the 4 tasks. **CASE** completed the jobs 2.7× faster than single-assignment, which is comparable to the results we see for Rodinia. We attribute such a huge improvement of **CASE** to its ability of balancing workloads among devices. As shown in Figure 9, on a 4-device system, **CASE** averaged ~80% device utilization, while *SchedGPU* only has 23%. This implies that, in *SchedGPU*, one of the devices is extremely overloaded with almost 100% utilization, while the other 3 devices are idle and wasted.

5.4 Kernel Slowdown

We looked at the amount of extra time needed to run a given kernel on the GPU with this framework. We compared the two scheduling algorithms to single-assignment on the 8 Rodinia workloads (Table 2) on the 4×V100 system. As we see in Table 6, Algorithm 2 averaged 1.8% slowdown; Algorithm 3 averaged 2.5%. The averages are over each workload's slowdown. Note that the "speedups" in workload 1 are noise. Standard deviations for the kernel slowdowns are ~5% and 3% for Algorithms 2 and 3, respectively, for this workload. Thus, both algorithms cause negligible slowdowns to the kernels themselves; and compared with each other, the difference is less than 1%. The absolute times for these kernels range from less than 1ms to over 20s.

6 Future Work

In our future work, we will improve **CASE** in the following aspects. 1) Process isolation: Currently, **CASE** is mainly designed to target HPC applications and assumes that process isolation is treated elsewhere. This assumption is usually easy to meet in HPC settings, because all applications running in supercomputers are only from authorized users and

Table 5. Darknet tasks and their corresponding commands.

Task	Command
Predict	cat images-large.txt | darknet classifier predict imagenet1k.data darknet53_448.cfg darknet53_448.weights
Detect	cat images-medium.txt | darknet detect cfg/yolov3-tiny.cfg weights/yolov3-tiny.weights
Generate	darknet rnn generate cfg/rnn.cfg weights/shakespeare.weights -len 100000
Train	darknet classifier train cfg/cifar.data cfg/cifar_small.cfg

Table 6. Kernel slowdowns for Algorithms 2 and 3 for Rodinia on 4×V100s, expressed as a percentage of SA.

Sched	W1	2	3	4	5	6	7	8	Avg
Alg2	-0.3	1.0	0.3	4.1	2.9	5.1	1.1	0.6	1.8
Alg3	-0.7	0.8	7.0	3.1	2.2	4.1	0.4	2.9	2.5

Table 7. Absolute jobs/sec throughput across workloads for each of the baselines for Rodinia (Alg2-V100 is the baseline for Figure 5, SA-P100 is the baseline for Figure 6a, SA-V100 is the baseline for Figure 6b).

WL	Alg2-V100	SA-P100	SA-V100
1	0.16	0.073	0.139
2	0.13	0.068	0.123
3	0.26	0.083	0.17
4	0.45	0.108	0.189
5	0.28	0.088	0.174
6	0.27	0.099	0.184
7	0.27	0.107	0.182
8	0.2	0.07	0.143

Table 8. Absolute jobs/sec throughput across workloads for the baseline for Darknet (SchedGPU is the baseline for Figure 8).

WL	SchedGPU
Predict	0.042
Detect	0.093
Generate	0.037
Train	0.013

are already deemed safe and trustworthy. It may not hold for some other environments, however, where independent processes cannot be trusted, which could have implications to both fairness and security. In terms of fairness, a "greedy" process may request and hold large resources (e.g. the majority of a GPU's memory), which can negatively impact other processes. In terms of security, allowing independent jobs to run simultaneously on a GPU with no oversight clearly violates best practice. Existing techniques can aid in both cases and be combined with **CASE**. For example, existing preemption techniques can be integrated to mitigate the fairness issue (see our discussion in Section 2). Similarly, NVIDIA's MIG feature can help both fairness and security by segmenting hardware resources and preventing interference. 2) Robustness: **CASE** currently also assumes that each GPU application is well programmed and will not crash when running independently on the system. We will relax this assumption to support cases when a GPU task crashes unexpectedly due to bugs or other failures. **CASE**'s runtime system will have to capture such crashes with customized signal handlers, which would allow it to accurately track device statuses even in these scenarios.

7 Conclusion

In this paper, we present a fully automated GPU scheduling framework to uniformly and transparently manage GPU resources. It constructs CUDA tasks via static program analysis and a lazy runtime, and schedules CUDA tasks from independent workloads (uncooperative processes) onto GPU devices. With the knowledge of resource requirements for each CUDA task, it guarantees zero OOM errors among co-executing tasks from independent processes. We evaluated the system on the Rodinia benchmark suite on two different GPU families, Pascal and Volta. On average, on a 2-GPU P100 system, the framework improves system throughput by 2.2× over a memory-safe scheduler, and by 64% over a memory-unsafe scheduler that has a crash frequency of 11%. On a 4-GPU V100 system, the throughput improves by an average of 2× over a memory-safe scheduler, and by 41% over a memory-unsafe scheduler with a crash frequency of 20%. We evaluated the 4-GPU system on neural network workloads and measured similar results (2.7× throughput improvement over competing state-of-the-art [15]). Individual kernel execution speeds degrade by 1.8% to 2.5% under the framework. **CASE** achieves a peak utilization of 78% for Rodinia with an average of 23.9% and 83% peak and average utilization for ML workloads on a 4 V100 GPU system. Supported by this empirical evaluation, we believe that such an automated solution is a practical way of solving the GPU sharing problem to boost throughput and device utilization.

The artifact of this work is available (refer to Appendix A for details).

Acknowledgments

We thank anonymous reviewers and the shepherd for their invaluable comments. They helped to improve the paper significantly.

Appendix

A CASE Artifact

The artifact for **CASE** is hosted by Zenodo at https://zenodo.org/record/5787410. Its digital object identifier (DOI) is 10.5281/zenodo.5787410. A README with instructions for building and running **CASE** can be found in the zipped archive within GPU-Sched-master.zip.

References

[1] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. July 11-13, 2018. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, Boston, MA, USA, 533–546.

[2] C. Basaran and K. Kang. 2012. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *2012 24th Euromicro Conference on Real-Time Systems*. IEEE, Pisa, Italy.

[3] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*. IEEE Computer Society, Austin, TX, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[4] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the 2010 IEEE International Symposium on Workload Characterization, IISWC 2010, Atlanta, GA, USA, December 2-4, 2010*. IEEE Computer Society, Atlanta, GA, 1–11. https://doi.org/10.1109/IISWC.2010.5650274

[5] T. Chen, Mu Li, Y. Li, M. Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, B. Xu, C. Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv* abs/1512.01274 (2015).

[6] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing Simulation*. IEEE, Caen, France, 224–231.

[7] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. 2009. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, Vol. 9. Citeseer.

[8] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2011. Pegasus: Coordinated Scheduling for Virtualized Accelerator-Based Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Portland, OR). USENIX Association, USA, 3.

[9] Jingoo Han, M. Mustafa Rafique, Luna Xu, Ali Raza Butt, Seung-Hwan Lim, and Sudharshan S. Vazhkudai. 2020. MARBLE: A Multi-GPU Aware Job Scheduler for Deep Learning on HPC Systems. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*. IEEE, 272–281. https://doi.org/10.1109/CCGrid49817.2020.00-66

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[11] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. 2012. Gdev: First-Class GPU Resource Management in the Operating System. In *Preceedings of 2012 USENIX Annual Technical Conference*. USENIX, Boston, MA, 401–412.

[12] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. [n.d.]. CIFAR-10 (Canadian Institute for Advanced Research). ([n. d.]). http://www.cs.toronto.edu/~kriz/cifar.html

[13] NVIDIA. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. Technical Report V1.0. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[14] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey). ACM, 593–606. https://doi.org/10.1145/2694344.2694346

[15] Carlos Reaño, Federico Silla, Dimitrios S. Nikolopoulos, and Blesson Varghese. 2018. Intra-Node Memory Safe GPU Co-Scheduling. *IEEE Trans. Parallel Distributed Syst.* 29, 5 (2018), 1089–1102. https://doi.org/10.1109/TPDS.2017.2784428

[16] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[17] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal). ACM, 233–248. https://doi.org/10.1145/2043556.2043579

[18] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[19] Kittisak Sajjapongse, Xiang Wang, and Michela Becchi. 2013. A Preemption-Based Runtime to Efficiently Schedule Multi-Process Applications on Heterogeneous Clusters with GPUs. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing* (New York, New York, USA) *(HPDC '13)*. Association for Computing Machinery, New York, NY, USA, 179–190. https://doi.org/10.1145/2493123.2462911

[20] L. Shi, H. Chen, J. Sun, and K. Li. 2012. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Trans. Comput.* 61, 6 (2012), 804–816.

[21] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.1556

[22] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. [n.d.]. *Top500 The List*. https://www.top500.org/

[23] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*.

[24] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, Laurent Réveillère, Tim Harris, and Maurice Herlihy (Eds.). ACM, 18:1–18:17. https://doi.org/10.1145/2741948.2741964

[25] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 483–496. https://doi.org/10.1145/3037697.3037742

[26] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and*

Implementation (Carlsbad, CA, USA). USENIX, 595–610.

[27] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing, 9th International Workshop*, Vol. 2862. Springer, 44. https://doi.org/10.1007/10968987_3

[28] Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. 2015. VirtCL: a framework for OpenCL device abstraction and management. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, Albert Cohen and David Grove (Eds.). ACM, 161–172. https://doi.org/10.1145/2688500.2688505

[29] H. Zhou, G. Tong, and C. Liu. 2015. GPES: a preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 87–97.