Performance of CUDA Unified Memory in CMS Heterogeneous Pixel Reconstruction

Matti J. Kortelainen[1,*] and *Martin* Kwok[1,**] on behalf of the CMS Collaboration

[1]Fermi National Accelerator Laboratory, Batavia, IL, USA

Abstract. The management of separate memory spaces of CPUs and GPUs brings an additional burden to the development of software for GPUs. To help with this, CUDA unified memory provides a single address space that can be accessed from both CPU and GPU. The automatic data transfer mechanism is based on page faults generated by the memory accesses. This mechanism has a performance cost, that can be with explicit memory prefetch requests. Various hints on the inteded usage of the memory regions can also be given to further improve the performance. The overall effect of unified memory compared to an explicit memory management can depend heavily on the application. In this paper we evaluate the performance impact of CUDA unified memory using the heterogeneous pixel reconstruction code from the CMS experiment as a realistic use case of a GPU-targeting HEP reconstruction software. We also compare the programming model using CUDA unified memory to the explicit management of separate CPU and GPU memory spaces.

1 Introduction

Graphics Processing Units (GPUs) are commonly used to accelerate scientific computing because of their cost and power efficiency in solving many data-parallel problems. Their programming model introduces a concept of separate memory spaces between the host (CPU) and devices (GPUs). Traditionally the data in these memory spaces have to be managed explicitly, for example calling a function to copy the bytes from host to device (or vice versa), and possibly allocating a specific memory buffer in pinned host memory for asynchronous memory transfers and then copying data from regular host memory to pinned host memory (or vice versa). Data structures that use pointers are especially tedious to transfer because of the need to rewrite the pointers from host memory addresses to device memory addresses.

CUDA [1] 6.0 introduced unified memory to simplify the programming model by providing a single memory space that the runtime and the hardware manage between the host and devices according to demand. The automation comes with a runtime cost from tracking the memory accesses through page faults. The runtime penalty can be mitigated with explicit prefetch calls to initiate the data transfers earlier. The performance impact of unified memory with or without prefetching depends, however, on application and the exact memory access patterns and device kernel computation times.

[*]e-mail: matti@fnal.gov
[**]e-mail: kkwok@fnal.gov

In this work, we evaluate the performance impact of the CUDA unified memory compared to manage the separate host and device memory spaces explicitly. We use the Patatrack heterogeneous pixel reconstruction workflow [2] from the CMS experiment [3] at the CERN LHC [4] as a use case for a set of realistic HEP reconstruction algorithms that are able to effectively utilize a GPU. Even if of this study being specific to CUDA, we believe the conclusions largely hold on other technologies as well, except for a case where the host and the device share the same physical memory. In addition, some approaches for portable code between CPU and GPUs rely on unified memory or equivalent, for example in NVIDIA HPC Compiler support for C++ parallel algorithms [5].

This paper is organized as follows. Technical aspects of the Patatrack pixel reconstruction are described in Section 2. The use of CUDA unified memory in the Patatrack code is discussed in Section 3. Performance measurements and their results are shown in Section 4, and conclusions are given in Section 5.

2 Structure of the pixel reconstruction application

The Patatrack pixel reconstruction pioneered offloading algorithms to NVIDIA GPUs with direct CUDA programming within the CMS data processing software (CMSSW) [6]. The offloaded chain of reconstruction algorithms takes the raw data of the CMS pixel detector as an input, along with the beamspot parameters and necessary calibration data, and produces pixel tracks and vertices as an output. CMSSW schedules algorithms as units that are called *modules*. The algorithms are organized in five CMSSW framework modules, depicted in Figure 1 as a directed acyclic graph (DAG) by their data dependencies, that communicate the intermediate data in the device memory through the CMSSW event data. The BeamSpot module only transfers the beamspot data to the device memory. The Clusters module transfers the raw data to the device memory, unpacks them, calibrates the individual pixels, and clusters the pixels on each detector module. The RecHits module estimates the 3D position of each cluster and forms hits. The Tracks module forms n-tuplets from the hits and fits the hit n-tuplets to obtain track parameters, and the Vertices module forms vertices from these tracks. There are further modules that optionally transfer the tracks and vertices to the CPU, and convert the Structure-of-Array (SoA) data structures to the data formats used by downstream algorithms in CMSSW, but those are not considered in this work and therefore not shown in Figure 1.

The CUDA code of the Patatrack pixel reconstruction was extracted into a standalone program [7] mainly to explore performance portability technologies. The separation from CMSSW gives freedom e.g. for compilers, build rules, external libraries, and code organization that would be more laborious to achieve in the full CMSSW software stack. These

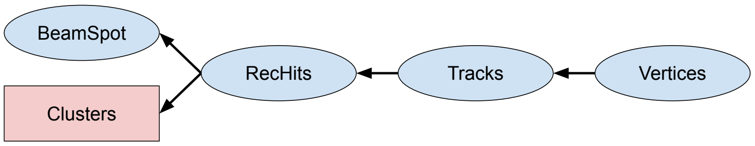

Figure 1. Directed acyclic graph of the framework modules in the Patatrack pixel reconstruction. The arrows denote the data dependencies of the modules, e.g. RecHits module depends on BeamSpot and Clusters modules. The Clusters module (red rectangle) is the only one that transfers data from the device to the host and uses the ExternalWork synchronization mechanism, while the other modules (blue oval) do not.

benefits can be useful for other technology exploration than only portability tools, like the impact of using CUDA unified memory as in this work.

The standalone program was crafted to mimic several aspects of the CMSSW, including similar organization of code into shared libraries, plugin libraries that are loaded dynamically based on run-time information, and a simple framework that uses Intel Threading Building Blocks (TBB) [8] for multi-threading. Borrowing from the CMSSW framework concurrency features [9–11], this simple framework implements only an event loop based on the TBB tasks, processing multiple events concurrently, and processing independent modules concurrently for the same event. There is only a single module type of each module having a separate instance for each concurrent event, and the *External Worker* concept [12] is included in order to use the host threads to do other work while the device is running the offloaded work.

The CMSSW tools to use the CUDA runtime directly from framework modules [12] are also included. On the device, the events are processed concurrently with CUDA streams. Each parallel branch in the data dependence DAG gets its own CUDA stream dynamically. In the case of Figure 1, this strategy means that the BeamSpot and Clusters modules get separate CUDA streams, the remaining modules queue their work in the same CUDA stream as the Clusters module, and RecHits module synchronizes the work of BeamSpot and Clusters modules with `cudaStreamWaitEvent()` before queueing its work.

The standalone setup includes a binary data file that contains raw pixel detector data from 1000 simulated top quark pair production events from CMS Open Data [13], with an average of 50 superimposed pileup collisions with a center-of-mass energy of 13 TeV, using conditions corresponding to the design 2018 CMS detector. All of the data, about 250 MB, are read into the host memory at the job startup to exclude I/O from the throughput measurement. The necessary pixel detector conditions data are also stored in binary files, and are read into the host memory at the job startup time. The data processing throughput is measured by measuring the time spent in the event processing, and dividing the number of processed events with that time. This event processing time includes the time taken to copy the raw data of each event from the pre-read memory buffer into an object in the event data.

3 Use of CUDA Unified Memory

An important ingredient for the achieved performance of the Patatrack pixel tracking is a memory pool to amortize the costs of raw CUDA memory allocations. Therefore we first ported the memory pool, based on the `CachingDeviceAllocator` from the CUB library [14], to the semantics of `cudaMallocManaged()`. This development was fairly smooth, and provided a memory allocation API similar to the current code enabling a straightforward migration path for the code.

We migrated the code to use the unified memory component by component. The first migrated component was all the conditions data that are transferred to each device once during the job at their first use. In this use case, the unified memory allowed a significantly simpler code, depicted in Figure 2, compared to explicit memory management, shown in Figure 3. The explicit memory management approach has to deal with the complexity of allocating device memory for each device, transferring the data to each device, and keeping alive the pinned host memory at least until all transfers all complete. With unified memory it is sufficient to just allocate the memory, optionally advise on the usage of the memory region, and optionally prefetch the data.

The following components were migrated to use unified memory: the BeamSpot, Clusters, and RecHits modules (see Figure 1). At the time of writing, the Tracks and Vertices modules are still to be migrated. For these components we introduced preprocessor macros to switch between explicit and unified memory for each component separately to be able to

```
struct SiPixelFedCablingMapGPU; // definition omitted for brevity

class CablingWrapper {
public:
  explicit CablingWrapper(SiPixelFedCablingMapGPU const& cablingMap) {
    cudaMallocManaged(&cablingMap_, sizeof(SiPixelFedCablingMapGPU));
    *cablingMap_ = cablingMap;
    int ndev;
    cudaGetDeviceCount(ndev);
    for (int device = 0; device < ndev; ++device) {
      cudaMemAdvise(cablingMap_, sizeof(SiPixelFedCablingMapGPU),
                    cudaMemAdviseSetReadMostly, device);
      cudaSetDevice(device);
      auto stream = cms::cuda::getStreamCache().get();
      cudaMemPrefetchAsync(cablingMap_, sizeof(SiPixelFedCablingMapGPU),
                           device, stream.get());
  }
  ~CablingWrapper() {
    cudaFree(cablingMap_);
  }

  const SiPixelFedCablingMapGPU* get() const {
    return cablingMap_;
  }

private:
  SiPixelFedCablingMapGPU *cablingMap_;
};
```

Figure 2. A simplified example of the conditions data distribution to all devices with the unified memory management. Caching of CUDA streams is visible with the call to `cms::cuda::getStreamCache().get();`. That function returns an `std::shared_ptr` holding a CUDA stream object that gets returned to the cache upon the `shared_ptr` destruction.

test their impact on performance. In the use cases of event data, we found the programming model with unified memory to be only a little bit simpler than with explicit memory management.

We are using preprocessor macros also to enable prefetch calls (`cudaMemPrefetchAsync()`), and to advise the runtime that specific memory ranges are read only with `cudaMemAdvise()` and `cudaMemAdviseSetReadMostly` attribute. The programming model with prefetching is similar to explicit memory transfers. Advising the usage of a memory range, on the other hand, is more complicated in conjunction with a memory pool, because of having to unset the advise before freeing the memory range to the memory pool. Therefore, if either of these calls would be needed to gain performance, much of the simplicity in the programming model would be lost. We identify two use cases where programming would nevertheless be simpler than with explicit memory: for data to be transferred to many devices, and for data structures that heavily use pointers to refer to other locations in the unified memory space. Such data structures are tedious to manage explicitly because of having to re-write the pointers from host memory addresses to device memory addresses. The Patatrack pixel tracking code has only a few such data structures, and even they have only one level of pointer indirection. Therefore, migrating to unified memory does not bring obvious simplification to the event processing code.

```
struct SiPixelFedCablingMapGPU; // definition omitted for brevity

class CablingWrapper {
public:
  explicit CablingWrapper(SiPixelFedCablingMapGPU const& cablingMap) {
    cudaMallocHost(&cablingMapHost, sizeof(SiPixelFedCablingMapGPU));
    *cablingMapHost_ = cablingMap;
  }
  ~CablingWrapper() {
    cudaFreeHost(cablingMapHost_);
  }

  const SiPixelFedCablingMapGPU* getAsync(cudaStream_t cudaStream) const {
    const auto& data = gpuData_.dataForCurrentDeviceAsync(cudaStream,
        [this](GPUData& data, cudaStream_t stream) {

      cudaMalloc(&data.cablingMapDevice, sizeof(SiPixelFedCablingMapGPU)));
      cudaMemcpyAsync(data.cablingMapDevice, this->cablingMapHost,
              sizeof(SiPixelFedCablingMapGPU), cudaMemcpyDefault, stream));
    });
    return data.cablingMapDevice;
  }

private:
  SiPixelFedCablingMapGPU *cablingMapHost_;  // pointer to struct in CPU

  struct GPUData {
    ~GPUData() {
      cudaFree(cablingMapDevice);
    }
    SiPixelFedCablingMapGPU *cablingMapDevice;  // pointer to struct in GPU
  };
  cms::cuda::ESProduct<GPUData> gpuData_; // distributes data to all devices
};
```

Figure 3. A simplified example of the conditions data distribution to all devices with the explicit memory management. Implementation of the helper class `cms::cuda::ESProduct<T>` is omitted for brevity. Its complexity can be summarized as 61 lines of code containing carefully crafted synchronization logic. Checks on the return values of the CUDA API calls are also omitted for brevity.

4 Performance measurements and results

The performance tests were done on GPU nodes of the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC). A Cori GPU node has two sockets with Intel Xeon Gold 6148 ("Skylake") processors, each with 20 cores and 2 threads per core, and eight NVIDIA V100 GPUs. For this work we used only one CPU socket, to avoid NUMA effects, and one GPU. In all tests, the threads were pinned to a single socket, and the node was free from other activity. Each job was run for approximately 5 minutes, processing the set of 1000 individual events for some integer factor times, and repeated 8 times on random nodes of the GPU cluster. The code was compiled with GCC 8.3.0, and `nvcc` from CUDA 11.1.

Using unified memory for only the conditions data is expected to have a smaller impact on the performance than using it for both the conditions and event data, because the conditions data need to be updated orders of magnitude less frequently; in the case of the standalone Patatrack pixel tracking program exactly once at the beginning of the job. This difference is shown in Figure 4, which depicts the event processing throughput as a function of events

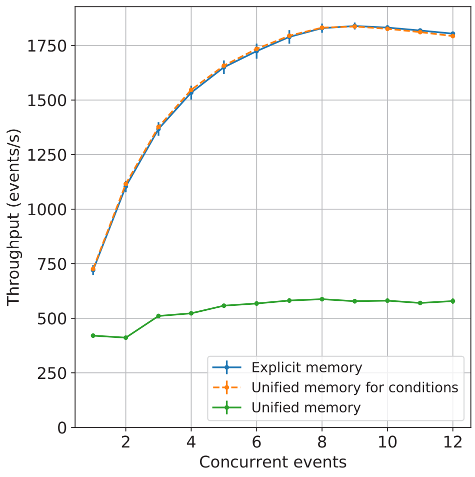

Figure 4. Event processing throughput of the standalone Patatrack pixel tracking using explicit memory (blue), CUDA unified memory applied to the management of conditions data (orange) and unified memory applied to all modules up to the RecHits module (green) as a function of the number of concurrent events. The program uses the same number of threads as concurrent events except for the case of 1 concurrent event, for which 2 threads are used. For the full unified memory case, the highest throughput was achieved by advising read-only memory regions with `cudaMemAdvise()` and without prefetching data. Using unified memory for conditions data shows an impact of less than 1 % on the overall throughput with respect to explicit memory. The quoted uncertainties correspond to the sample standard deviation of 8 trials.

being processed concurrently using unified memory only for conditions data, and for conditions and event data for the modules that have been migrated so far, and compares those to the throughput of managing the host and device memory explicitly. The throughput of using unified memory only for the conditions data is observed to be within 1 % to that of the explicit memory management throughout the tested range of concurrent events. Using unified memory for the event data, however, results in a significant degradation in the throughput, to 30–55 % with respect to the explicit memory management.

The throughput measurement for the unified memory shown in Figure 4 is the best one from the set of with or without data prefetching with `cudaMemPrefetchAsync()`, and with or without advising read-only memory regions with `cudaMemAdvise()`. The throughput as a function of concurrent events for all these four cases are shown in Figure 5. Even though both options are intended improve the performance of unified memory, we found that in the Patatrack pixel tracking application the highest throughput is obtained with advising read-only memory regions, but *without* prefetching data. With respect to this configuration, enabling prefetching (orange line in Figure 5) leads to around 20 % decrease in throughput, which was not expected. Without the read-only memory advice, the decrease in throughput brought by prefetching the data is smaller, around a few to 10 %. Conversely, without the data prefetch, advising the read-only memory regions improves the throughput by 10–15 %.

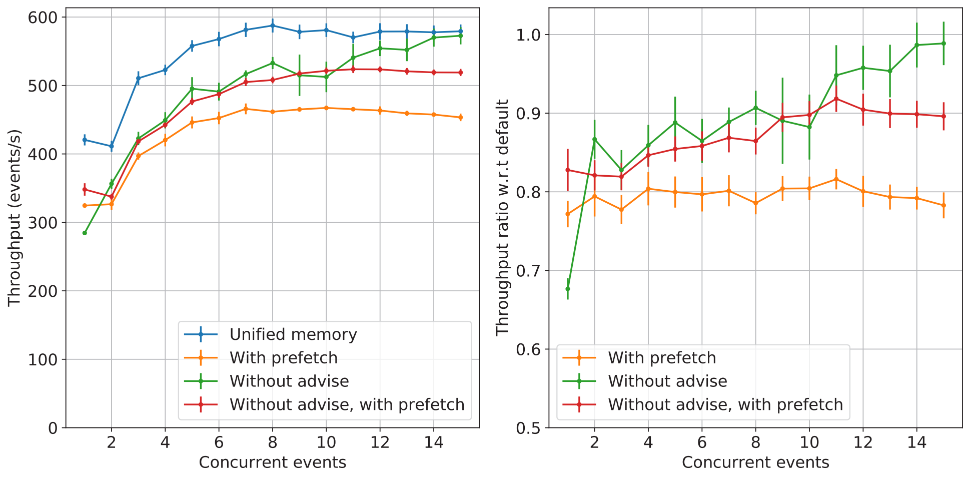

Figure 5. *Left:* Event processing throughput of the Patatrack pixel tracking using different unified memory options as a function of the number of concurrent events. The program uses the same number of threads as concurrent events except for the case of 1 concurrent event, for which 2 threads are used. The highest throughput is achieved using without `cudaMemPrefetchAsync()` and with `cudaMemAdvise()` (blue) and is chosen as the default option. Other combinations of prefetch and advise options varied with respect to the default options are shown. The quoted uncertainty corresponds to sample standard deviation of 8 trials.
Right: Ratio of throughput using unified memory with respect to the default unified memory options as a function of number of threads.

5 Conclusions

We have ported parts of the Patatrack heterogeneous pixel reconstruction code from explicit CUDA host and device memory management to use CUDA unified memory in a standalone setup, and measured the impact on the event processing throughput with NVIDIA V100 GPU. We observed that using unified memory for infrequently changing conditions data gave performance within 1 % of the throughput achieved with the explicit memory management. In this use case the unified memory provided clearly simpler programming model. For the event data, however, we observed that already partial use of the unified memory incurred a significant decrease in the maximum throughput, from 1840 ± 20 events/s to 588 ± 10 events/s. We achieved the best performance with unified memory by advising that applicable memory regions are read only, but without prefetching the event data from host to device. The prefetching decreasing the performance was unexpected, we presume this effect could be related to lock contention of the global mutex within CUDA runtime.

Acknowledgements

This work was supported by the U.S. Department of Energy, Office of Science, Office of High Energy Physics, High Energy Physics Center for Computational Excellence (HEP-CCE) at Fermi National Accelerator Laboratory, under B&R KA2401045. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

References

[1] NVIDIA, *CUDA C++ Programming Guide, version 11.2* (2021)

[2] A. Bocci, V. Innocente, M. Kortelainen, F. Pantaleo, M. Rovere, Front. Big. Data **3**, 601728 (2020), `2008.13461`

[3] CMS Collaboration, JINST **3**, S08004 (2008)

[4] L. Evans, P. Bryant, JINST **3**, S08001 (2008)

[5] NVIDIA, *NVIDIA HPC Compilers, C++ parallel algorithms* (2021)

[6] C.D. Jones, M. Paterno, J. Kowalkowski, L. Sexton-Kennedy, W. Tanenbaum, *The New CMS Event Data Model and Framework*, in *Proceedings of International Conference on Computing in High Energy and Nuclear Physics (CHEP06)* (2006)

[7] *Standalone Patatrack pixel tracking*, https://github.com/cms-patatrack/pixeltrack-standalone/ (2021), accessed: 2021-02-07

[8] *oneAPI Threading Building Blocks*, https://github.com/oneapi-src/oneTBB (2021), accessed: 2021-02-07

[9] C.D. Jones, E. Sexton-Kennedy, J. Phys.: Conf. Series **513**, 022034 (2014)

[10] C.D. Jones, L. Contreras, P. Gartung, D. Hufnagel, L. Sexton-Kennedy, J. Phys.: Conf. Series **664**, 072026 (2015)

[11] C.D. Jones, J. Phys.: Conf. Series **898**, 042008 (2017)

[12] A. Bocci, D. Dagenhart, V. Innocente, C. Jones, M. Kortelainen, F. Pantaleo, M. Rovere, EPJ Web Conf. **245**, 05009 (2020)

[13] CMS Collaboration, *TTToHadronic_TuneCP5_13TeV-powheg-pythia8 in FEVT-DEBUGHLT format for 2018 collision data. CERN Open Data Portal.*, doi:10.7483/OPENDATA.CMS.GOB0.0LEW (2019)

[14] *CUB*, https://nvlabs.github.io/cub/ (2021), accessed: 2021-02-07