Implementation and Evaluation of CUDA-Unified Memory in Numba

Lena Oden and Tarek Saidi
Fernuniversität in Hagen, Lehrstuhl Technische Informatik, Hagen, Germany
lena.oden@fernuni-hagen.de

Abstract. Python as a programming language is increasingly gaining importance, especially in data science, scientific, and parallel programming. With the Numba-CUDA, it is even possible to program GPUs with Python using a CUDA like programming style. However, Numba is missing support for CUDA-unified memory, which can help to simplify programming even more and allows dynamic work distribution between GPUs and CPUs. In this work, we implement and evaluate the support for unified memory in Numba. As expected, the performance of unified memory is worse than using explicit data transfers, but can outperform the performance of the implicit methods provided by Numba. Additionally, using unified memory can help to reduce the Python interpreter overhead and therefore help to improve the performance of small problem sizes. The use of system-wide atomic can help to improve the work distribution between GPU and CPU, but when using more CPU threads the performance suffers under the Python global interpreter lock (GIL).

Keywords: GPU · Python · Unified memory · Numba

1 Introduction

In the early days of information technology, the exorbitant costs of machines eclipsed all other accompanying costs, especially for programming these machines. However, the exponential growth in computing speed and the increasing complexity of software systems ensured that the circumstances were reversed. Many organizations found that their software development costs began to exceed the hardware costs.

A sign of this development is the continuing trend towards the use of dynamic scripting languages. Although these are not compatible with compiled languages and considerably slower, but the development costs can be reduced [14]. One language that is popular in this context is Python.

To compensate for the speed disadvantages of the scripting language, performance critical numerical calculations are written in compiled languages, either by using pre-compiled libraries like numpyp scipy, or by using bridge technologies like cython. However, the desire to carry out mathematical calculations directly in Python is obvious [2]. They enable the rapid development of prototypes and their iterative improvement on an algorithmic level. In fact, the greatest and
most efficiently exploitable optimization potential is usually to be found in the higher-level algorithms of a calculation, whereas low-level optimizations often do not yield a profit in relation to the effort involved [16].

A possibility to create high-performance code for numerical calculations directly in Python is provided by the Python extension module Numba [9]. Numba is a just-in-time compiler that allows the translation of selected computationally intensive Python functions into optimized machine code. The execution speed of these functions is similar to pre-compiled code of other languages [1]. Numba supports the parallelization of Python code and often requires only minor code changes. In addition, Numba-CUDA can also be used to program NVIDIA GPUs.

Since GPUs have their own local memory, data must be exchanged between the system memory and the device memory. Numba-CUDA uses a very simple implicit mechanism by default. It copies all data used before a calculation into the device memory and back after completion of a kernel. If this is done explicitly in the program code, however, the complexity of the programs and the porting effort for existing algorithms increase. If, on the other hand, the data transfer is implicitly performed, the speed can suffer depending on the efficiency of the transfer.

With unified memory, the CUDA platform offers a technology that allows avoiding unnecessary data transfers and explicit memory management [7]. Unified memory provides a uniform virtual address space between system and device memory. On newer GPUs, the data transfer between the physical storage is completely transparent due to a demand paging mechanism implemented at driver and hardware level.

However, unified memory currently is not supported in Numba. In this work, we extend Numba-CUDA to support unified memory, in order to allow efficient implicit memory management. We evaluate our implementation in terms of performance and compared with other memory management provided by Numba-CUDA.

2 Background

In this section, we will give a short overview of the technical background of unified memory and Numba.

2.1 Numba

Numba is a just-in-time compiler working at the level of individual functions in Python. The primarily intention is to accelerate numerical calculations. Unlike other JIT compilers for interpreted languages, Numba is not designed to produce machine code that necessarily works in the same way as the interpreter. Instead, Numba uses knowledge about the internals of data types to compile the Python code into simpler machine code. As a result, Numba only supports a subset of Python that is tailored to the scope of numerical calculations. However, thanks to this, Numba achieves execution speeds similar to compiled C code.
The compilation starts by converting the Python byte code into an immediate representation, called Numba IR. On this, the type interference is performed. If the type of each value can be inferred, the code is lowered to LLVM, which is then used to create the final machine code. In addition to the generation of JIT compiled CPU functions, Numba supports, in similar way, the generation of CUDA kernels for execution on the GPU. Because of the highly parallel programming model of CUDA, CUDA kernels under Numba differ in some aspects of CPU functions. For example, when calling a kernel, the thread grid must be explicitly specified, similar to C-CUDA.

The compilation process for CUDA kernels is essentially the same as for CPU-functions. The generated NVVM-IR is a modification of the of the LLVM IR code [4]. This intermediate code is translated by the LLVM-based proprietary library libNVVM to CUDA PTX-assembly code. When using NumPy arrays for CUDA-kernel input and output, Numba will implicitly transfer data between the host and the device without additional instructions. To allow explicit control over the memory, Numba provides so-called device arrays and functions. This is similar to regular C-CUDA programming using CUDA-copy functions.

### 2.2 Unified Memory in CUDA

CUDA integrates allocated device memory into the virtual address space of the host system, whereby each memory address is unique [3]. However, this alone does not yet allow every processor (or GPU) to access every memory region, but only local memory. One way to overcome this limitation is to use CUDA with mapped memory. This allows the GPU to access areas of host memory directly, but does not allow the CPU to access GPU memory.

With CUDA 6.0 and Kepler-generation GPUs, the CUDA memory model has been extended to include managed memory or unified memory, which allows all processors equal access to all data in a uniform virtual address space, regardless of the physical location. The advantage of this technology is primarily simplified programming, since explicit data transfer between host and device are omitted. On older GPUs, explicit data transfer was handled in the background.

Unified memory on newer GPUs is based on the demand paging mechanism. If a memory page is accessed, which is not currently in main memory, a page fault occurs and the operating system takes the necessary actions to make the page available. The unified memory works in a similar way. If the GPU accesses a memory page that is physically located in the host memory, a page fault occurs, which is handled by the GPU by transferring the respective page to host memory. On Power9 systems, the same mechanism also works in the opposite direction, when the CPU accesses GPU memory. Such CPU-side page faults are hardware technically only possible since the introduction of the Pascal architecture on Power9 system and also require operating system support, which is currently only available on Linux.
On systems that do not support GPU page faults, pages owned by the CPU are instead written to the respective GPU pages when a kernel is started and written back, if the kernel is completed.

With newer GPU-architectures, the system has been further refined so that, for example, pages are not necessarily transferred from one memory to the other during the first access, but only when a certain number of accesses has been exceeded.

3 Related Work

Unified memory was evaluated by different groups. In [10], the speed of unified memory with CUDA 6 and GPUs of the Kepler generation is examined. In almost all cases unified memory leads to significant performance losses. The authors consider the simplifications achieved by unified memory as marginal and only saw a benefit when using more complex data structures.

In [11] a consistent speed disadvantage of around 10% is found in the use of unified memory with GPUs of the Kepler generation. They attribute the performance losses on the one hand to the redundant transfer of data by the unified memory runtime system and on the other hand to the overhead of handling page faults.

Since its introduction with CUDA 6 and the Kepler architecture, unified memory has experienced some improvements. In [8] unified memory was evaluated on modern Pascal- and Volta-GPUs. This work also came to the conclusion that the use of unified memory is generally associated with significant performance losses. However, for memory oversubscribing use cases, unified memory allows a strong reduction of complexity in the memory management.

In [6] a benchmark suite, Chai, designed for heterogeneous computations, is introduced. Using these benchmarks, the workload distribution between CPUs and GPU generally shows a speed gain compared to a CPU-only or GPU-only calculations. Furthermore, it was found that the use of unified memory in conjunction with system-wide atomic operations for synchronisation has a clearly positive effect on performance.

As Python is widely used in scientific programming, different papers deal with the performance for parallel applications written in Python, including the use of GPUs. In [15] the same application is evaluated in Python/Numba and CUDA/Fortran. In [12] a detailed analysis of Numba is provided, but the authors do not provide a detailed comparison to C-CUDA. In [13] we compared the performance of C-CUDA and Numba-CUDA, but mainly focus on the performance of the compute kernels. The memory transfer between GPUs and CPUs was not considered.

4 Implementation of Unified Memory in Numba-Cuda

Unified memory simplifies programming by eliminating the need for explicit memory transfers between different physical memories. This means, the distinction between host arrays (i.e. normal NumPy arrays) and device arrays should be
Implementation and Evaluation of CUDA-Unified Memory in Numba

repealed. Additional, System-wide atomic operations should be made available in Numba, on both sides, CPU and GPU to allow work distribution. The implementation should fit well into the programming interface of Numba and make as few adjustments as possible to port existing programs. The current version of the patch is available as gist on github.¹

4.1 Managed Memory Arrays

The CUDA driver provides the function cuMemAllocManaged() to allocate managed memory. CPython allows a program-wide exchange of the used memory allocation functions through user-defined variants [17]. A simple solution could simply replace any memory request in Python with cuMemAllocManaged(). This would mean that each newly created NumPy array is automatically allocated in managed memory. This approach has a certain elegance at first glance, but also has a large number of problems.

Access to the CUDA driver API is only possible after loading a corresponding extension module, unless you modify the Python interpreter itself. This means, memory blocks from different systems must be managed, which is a time-consuming task. Additionally, this approach is not compatible with non-Linux systems or GPUs of the pre-Pascal era. Therefore, we introduce a new array class, which allocates and stores data in a unified memory block instead of host memory (normal numpy array) or GPU memory (device memory class). This avoids the problems listed above and only used data are stored in managed memory.

To ensure universal usability, we implemented a new array class, called ManagedArray, as a derived class of numpy.ndarray. This allows the usage like a normal NumPy array without further effort. In particular, managed arrays can be used directly in Numba and Numpy CPU functions.

When a new instance is initialized, the required memory for the data is allocated using cuMemAllocManaged() and passed as a buffer object to the superclass. The buffer object automatically frees the memory with cuMemFree() when the last reference to the object is removed. Besides of this modifications, all other functions can be inherited from the original numpy array superclass and no further modifications are needed. In an application, the allocation of a memory array must be replaced with the new class. To prevent Numba from making a automatic copy of the data to the GPU when a kernel is called, the class is marked the with __cuda_memory__. Listing 1.1 shows how the new class can be used to created manged device arrays.

Listing 1.1. Example of unified memory allocation in Numba

```python
numba.cuda.managed_array(shape, dtype=np.float, order='C')
numba.cuda.managed_array_like(ary)
numba.cuda.to_managed(ary)
```

¹ https://gist.github.com/LenaO/7d7cf6ec1822f3375d1b8a6b4ec1914e.
4.2 Global Atomic Functions

Numba supports some atomic operations for CUDA kernels, such as atomic fetch_and_add by using numba.cuda.atomic.add(). The analysis of the IR and PTX (cuda assembly) code generated by Numba shows that these atomic operations have a GPU-wide validity range. This means the atomicity is guaranteed for all other threads of the same GPU, but not with the host or other GPUs. To enable system-wide atomic operations, a PTX command like atom.sys.op.type has to be generated. The current, official version of LLVM supports system-wide atomic operations by special IR commands of the form llvm.nvvm.atom.gen.sys.*. Since Numba uses the non-open source LLVM based library libNVVM for PTX code generation, we investigated the support of these commands. We learned that the LLVM system wide atomics used with CUDA version 10.1 were not recognized.

As an alternative we identified the use of inline assembler expressions in the intermediate code. This means that instead of an NVVM IR command, the appropriate PTX code is embedded directly. The IR code thus loses abstraction, but there are no disadvantages in practical application.

The newly introduced system-wide atomic operations are oriented to the already existing operations. For CPU functions, Numba does not support any user-addressable atomic operations or other synchronization mechanisms at all, as Numba primarily relies on semi to fully automatic parallelization of loops. The implementation of atomic operations for CPU code turned out to be simple. Numba simply needed to be extended to generate the LLVM IR command atomicrmw. The semantics of the CPU functions correspond to those of the GPU variants (see Listing 1.2).

```
Listing 1.2. Example of atomic usage on the CPU and GPU
numba.cuda.atomic.system.add(ary, idx, val)
numba.atomic.add(ary, idx, val)
```

5 Evaluation

In this section, we evaluate the performance of our unified memory implementation in CUDA-Numba. All benchmarks were executed on an NVIDIA Tesla V100 GPU, which is equipped to a node with two IBM POWER9 processors (8 cores per core). We used CUDA 10.1.105. This system has full support for all unified memory features. We evaluate the performance of Unified Memory in Numba and explain the behavior compared to the same methods in C-CUDA applications.

5.1 Micro-Benchmarks

In the first step, we implemented three synthetic benchmarks. These benchmarks do not perform any useful computation, but serve the purpose of transferring data between host and device. The tests emulate the memory access patterns of typical application scenarios.
– *data transfer full* - In this benchmark, we first initialize an array on the host. A GPU kernel performs a simple calculation on this array and writes the result back. The CPU verifies the result. For this test, all data must first be copied into GPU memory and transferred back after the computation is completed for verification. The data volume, that is transferred is independent of the memory management method, as all used data is modified on the GPU.

– *data transfer partial* - This benchmark uses three arrays. Two provide input data and are initialized on the host side. The GPU reads data from input vectors and writes the result to the output vector. The host requires these data to verify the result.

– *data transfer multikernel* - In this test, two GPU kernels are executed successively. It uses one array. The second kernel uses the results of the first as input. The result of the second kernel is written to this array and the result is evaluated on the CPU. A suitable memory management avoids the data transfer between the execution of the two kernels.

For these benchmarks, five different memory management variants implemented and compared for Numba-CUDA

– *Implicit*: Only ordinary NumPy arrays are used. The data transfer is carried out automatically by Numba.

– *Explicit*: Device arrays are used and copy operations between Host and device explicitly triggered. Data is only copied when it is required.

– *Explicit pinned*: Similar to *explicit*, but the host arrays are allocated on pinned memory

– *Smart*: Numba Smart arrays are used, which perform the data transfer automatically as in the implicit case, but should use more intelligent mechanism to avoid unnecessary copying.

– *Managed*: Unified memory arrays are used.

Note that *smart arrays* are deprecated, but we still use them here to compare the performance. We also implemented C-CUDA versions of the benchmarks, using the explicit, pinned and managed memory.

### 5.2 Micro-benchmark-Performance

Figures 1, 2 and 3 show the performance of these benchmarks in Numba. For small array sizes, up to around 64 kByte, the managed array class shows quite good performance compared to the other methods. For the partial and multi-kernel benchmark, the performance is almost comparable to the pinned memory and much better than for smart arrays or the implicit variant.

For larger arrays, however, this changes. For the full data transfer benchmark, our managed array class performs significantly worse than all other variants. For the partial- and multi-kernel benchmark, the performance is almost comparable to the pinned memory and much better than for smart arrays or the implicit variant.

For larger arrays, however, this changes. For the full data transfer benchmark, our managed array class performs significantly worse than all other variants. For the partial- and multi-kernel benchmark, only the implicit variant is worse than the manged class. This is as expected, since for the partial and multi kernel benchmark the implicit version copies unnecessary data.
Fig. 1. Full data transfer Numba performance

Fig. 2. Partial data transfer Numba performance

Fig. 3. Multikernel data transfer Numba performance

Fig. 4. Full data transfer C performance
The performance of the managed class for large arrays is worse than using explicit copies. This is the expected behavior from previous work evaluating managed memory. Still, there are some results that are surprising, especially when comparing the results of the Numba benchmarks with the C-CUDA benchmarks.

The results of the full data transfer benchmark for C-CUDA are shown in Fig. 4. The differences between the managed memory and the explicit variants are larger than for Numba, even for smaller arrays. Furthermore, the runtime of the C-CUDA variants is significantly shorter than the Numba version, especially for small arrays.

It is surprising that the smart array variant performs significantly worse than the explicit variant, since the number of copy operations performed should be the same. We used the Nvidia profiling tool `nvprof` to understand this behavior better. For the partial benchmark, three copy operations are required: The two input vectors are copied to the device and the output vector is copied back. The implicit class requires six copy operations, since always all vectors are copied in and out. The smart class requires four copy operations: all vectors are copied in, but only the result is copied back. The additional copy explains the performance difference for the smart array class. For the multi-kernel benchmark, the same applies: the smart array class copies the data from GPU to host between the two kernel calls, although the data are not required on the host.

![Fig. 5. Timeline of the CUDA-API calls for the different versions of the full-data transfer benchmarks (4 kByte array)](image)

Figure 5 shows the timing of CUDA API-function calls for different versions of the full data transfer benchmark. This allows us to better understand the differences between the Numba version and the C versions. For this size, there is only a small difference between pinned and unpinned memory. Additional, the smart class behaves similarly to the implicit case. Therefore and, due to lack of space, we refrain from the presentation of these. We use CUDA-events to measure the timing, so we show the API calls starting with the recording of the first event and synchronising the second event.

By using implicit or smart data transfer, a `cudaMalloc` operation is required before the kernel is started. Using smart or Numpy arrays, only the host memory is allocated when a new array is created. Numba has a lazy deallocation policy, so these vectors are not freed when the function ends (instead there is a garbage collection that regularly releases all vectors together). Still, there is no reuse of allocated memory beyond the lifetime of an array.
The runtime of the calls to actual CUDA-API are similar for the C-Cuda and Numba versions, and the kernel have a similar runtime. However Numba versions have much more time passing between calls. The C-Cuda version usually has less than a microsecond between two API calls, while the Numba version sometimes requires more than 100 ms.

One reasons for this is the Python interpreter overhead. Although the same compiled CUDA-functions and kernels are called, using python everything in between is handled by the interpreter. Especially with small array sizes this overhead is much higher than the actual runtime of the kernel and the data transfer.

This shows one advantage of unified memory in Numba: Because fewer function calls are needed (no copy functions), the interpreter overhead is much smaller. Therefore we see a comparable performance for the whole benchmark for small sizes compared to the explicit case and a better performance than using implicit or smart copies. For larger arrays, however, this advantage increasingly fades into the background. The Python overhead remains constant, while the copy times and kernel runt times increase. Since the kernel runtime increases significantly when using unified memory (as every access to a new memory page triggers a page fault), the performance of unified memory is also significantly worse in our Numba benchmarks.

5.3 Work Distribution Using Atomic Operations

In order to use the new possibilities for work-sharing between several process units (PU), we transferred a benchmark from the Chai Benchmark Suite [6] to Numba. The Chai benchmarks are particularly suitable for computations based on the division of work in heterogeneous systems.

We selected the RSCD benchmark as an example. It calculates the RANSAC algorithm [5], which is an iterative method for estimating mathematical models on the basis of incorrect measured values. The benchmark distributes the work between PUs by dividing the input data. We implemented two the variants of the benchmark in Python/Numba:

- rscd_d. Uses discrete memory and explicit data transfer. The partitioning of the input data is carried out static, according to a specified CPU-to-GPU ratio $\alpha$.
- rscd_u. All PUs use the same memory for input and output, using manged memory. The workload is distributed dynamically, using atomic operations.

The use of unified memory the elimination of initial data partitioning, multiple keeping of variables and the final merging of results reduces the code complexity. The code of the variant rscd_u is significantly simpler than the variant rscd_d. We increased the size of the problem from the original vector by about factor 10 to generate more computational effort. Figure 6 shows the results. The problem with the Chai benchmark is that it was developed primarily for embedded systems. On our high performance GPUs, splitting the work between GPU
and host always leads to a worse result than if the calculation is done on the GPU only (this is also true for the C-CUDA version of the benchmark). We have used a small alpha of 0.01 for the static distribution. Even with this distribution, in which the CPUs only do 1% of the work, the performance is worse than when the job is only run on the GPU. This is similar to the distribution achieved when using dynamic distribution.

![Graph showing results of the Chai benchmark](image)

**Fig. 6.** Results of the Chai benchmark

The benchmark benefits in most configurations clearly from the use of unified memory. Using only GPU the runtime of variant `rscd_u` is 31% lower than that of the variant `rscd_d`. One of the reasons for this is certainly also that the problem size of the Chai benchmark is comparatively small, and many of the effects explained in the previous chapter play a role here. However, the use of atomic operation reduces the number of kernel invocations and thus the Python interpreter overhead.

The runtime of the unified memory benchmark increases, if more CPUs are used. One reason is CPython is not able to execute Python code in parallel in several threads simultaneously. The Global Interpreter Lock (GIL) ensures that only one thread of Python code is executed at a time. All other threads can execute only external non-Python code at the same time. As a result, the overhead of calling a JIT-compiled Numba function for the CPU threads are summed up, since the GIL is not unlocked until the JIT compiled machine code is called up. This requires further optimization in future work.

6 Conclusion

In this work we have implemented and evaluated the support of unified memory in Numba. Unlike the use of C-CUDA, there are cases where unified memory can help to improve performance by reducing the number of function calls and thus the overhead of the Python interpreter. However, this is only true for small problem sizes and if many context switches between GPU and Python are required. This is especially true for work sharing benchmarks between GPU and CPU, where the use of atomic operations allow a simple work distribution between CPU and GPU.

In a next step we want to evaluate the performance on embedded devices like the Jetson series. Although there is less support for unified memory here, we believe that they benefit even more from unified memory and the reduced
overhead. We also want to implement more features in Numba to improve the use of GPU memory, like the memory hints.

References

1. Anaconda, I., et al.: Numba user manual: overview. http://numba.pydata.org/numba-doc/latest/user/overview.html
2. Cai, X., Langtangen, H.P., Moe, H.: On the performance of the Python programming language for serial and parallel scientific computations. Sci. Program. 13(1), 31–56 (2005)
3. NVIDIA cooperation: CUDA C programming guide. https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf
4. NVIDIA cooperation: NVVM IR specification 1.4: reference guide. https://docs.nvidia.com/pdf/NVVM_IR_Specification.pdf
5. Fischler, M.A., Bolles, R.C.: Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Commun. ACM 24(6), 381–395 (1981)
6. Gómez-Luna, J., et al.: Chai: collaborative heterogeneous applications for integrated-architectures. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 43–54. IEEE (2017)
7. Harris, M.: Unified memory for CUDA beginners. https://devblogs.nvidia.com/unified-memory-cuda-beginners/
8. Knap, M., Czarnul, P.: Performance evaluation of unified memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs. J. Supercomput. 75(11), 7625–7645 (2019). https://doi.org/10.1007/s11227-019-02966-8
9. Lam, S.K., Pitrou, A., Seibert, S.: Numba: a LLVM-based python JIT compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, p. 7. ACM (2015)
10. Landaverde, R., Zhang, T., Coskun, A.K., Herbordt, M.: An investigation of unified memory access performance in CUDA. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE (2014)
11. Li, W., Jin, G., Cui, X., See, S.: An evaluation of unified memory technology on NVIDIA GPUs. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 1092–1098. IEEE (2015)
12. Marowka, A.: Python accelerators for high-performance computing. J. Supercomput. 74(4), 1449–1460 (2017). https://doi.org/10.1007/s11227-017-2213-5
13. Oden, L.: Lessons learned from comparing C-CUDA and Python-Numba for GPU computing. In: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 216–223 (2020)
14. Ousterhout, J.K.: Scripting: higher level programming for the 21st century. Computer 31(3), 23–30 (1998)
15. Pierro, V., Troiano, L., Mejuto, E., Filatrella, G.: Stochastic first passage time accelerated with CUDA. J. Comput. Phys. 361, 136–149 (2018)
16. Smaalders, B.: Performance anti-patterns. Queue 4(1), 44–50 (2006)
17. Van Rossum, G., Drake Jr, F.L.: Python/C API reference manual. Python Software Foundation (2002)