The Journal of Supercomputing (2019) 75:7625–7645
https://doi.org/10.1007/s11227-019-02966-8

Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs

Marcin Knap[1] · Paweł Czarnul[1]

Published online: 20 August 2019

Abstract
The paper presents assessment of Unified Memory performance with data prefetching and memory oversubscription. Several versions of code are used with: standard memory management, standard Unified Memory and optimized Unified Memory with programmer-assisted data prefetching. Evaluation of execution times is provided for four applications: Sobel and image rotation filters, stream image processing and computational fluid dynamic simulation, performed on Pascal and Volta architecture GPUs—NVIDIA GTX 1080 and NVIDIA V100 cards. Furthermore, we evaluate the possibility of allocating more memory than available on GPUs and assess performance of codes using the three aforementioned implementations, including memory oversubscription available in CUDA. Results serve as recommendations and hints for other similar codes regarding expected performance on modern and already widely available GPUs.

Keywords CUDA · Unified Memory · Prefetching · Memory oversubscription

1 Introduction

General-purpose computing on graphic processing unit (GPGPU) has become very popular. Significant advancements, both in hardware and in the CUDA API, have been adopted in recent years. On the programming side, some of the most important features of newer CUDA versions include dynamic parallelism allowing launching a kernel from within a kernel already running on a GPU or Unified Memory (UM) that proposes a programming abstraction of uniform memory space that can be

✉ Paweł Czarnul
pczarnul@eti.pg.gda.pl; pczarnul@eti.pg.edu.pl

Marcin Knap
marcknap@gmail.com

[1] Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Gdansk, Poland

allocated and used without the need for explicit management of data location [21]. In other words, the underlying runtime system migrates memory pages between host's memory and global memory of a GPU according to when the code running on either the CPU or the GPU refers to it. It allows to simplify the programming model considerably and allows reasonably good performance compared to low-level memory management when Unified Memory is not used [28]. One of the recent features of Unified Memory is the possibility to oversubscribe memory, i.e., to allocate more memory than available on a GPU [26]. Performance of this feature is investigated in this paper as well, compared to the standard CUDA implementation without Unified Memory.

2 Related work and motivations

Modern parallel programming for contemporary HPC systems [3] typically involves multithreading for multi- or many-core CPUs and accelerators such as GPUs and efficient communication between cluster nodes. The former can be implemented with, e.g., OpenMP, OpenCL or Pthreads for CPUs and CUDA, OpenCL, OpenACC for GPUs, while the latter can be implemented typically with MPI. Paper [4] presents an exemplary implementation and optimization of parallelization of large vector similarity computations in a hybrid CPU+GPU environment, including load balancing and finding configuration parameters. CUDA-aware MPI implementations allow using CUDA buffers in MPI calls which simplifies implementation.

Before NVIDIA's Unified Memory was introduced, other researchers proposed solutions for making GPU programming easier, especially with respect to easier memory management. Paper [1] presents a compiler approach for automatic scheduling of data transfers from and to accelerators which is aimed at reduction of data transferred between host and device. Only data needed or modified is transferred in identified locations in contrast to a naive approach with all data being copied. Significant gains were shown for selected Rodinia benchmarks such as: breadth-first search, particlefilter, speckle reducing anisotropic diffusion and Needleman–Wunsch. In paper [11], the authors proposed design of region-based software virtual memory (RSVM), a software virtual memory layer for both GPU and CPU. It offered transparent swapping of GPU memory to main memory for multiple kernels and fetching data from host to the GPU. NVIDIA's Unified Memory [9, 19], introduced in CUDA 6, simplifies implementation of a CUDA application since basically it only requires allocation of memory using `cudaMallocManaged()` instead of `cudaMalloc()` and proper synchronization on the host side after invocation of a kernel function before reading results, since a kernel call is asynchronous from the point of view of the host. Unified Memory was further expanded in subsequent versions of CUDA. Specifically, CUDA 8 with Pascal and later GPUs [26] extended UM with 49-bit virtual addressing and on-demand page migration. The new UM allows to use the whole system memory and memory oversubscription. In CUDA 9 and Volta [27], tracking accesses to pages through additional counters has been introduced. Frequency of accesses can impact the driver when deciding on page movements. The driver can migrate pages proactively and perform intelligent eviction [28].

In the Unified Memory version of the code, it is possible to provide hints on where and how data are to be used. It can be done with function `cudaMemPrefetchAsync()` for prefetching data to a GPU before a relevant kernel using that data are launched. Furthermore, it is possible to specify how data in managed memory at the given location with a given size will be used with a call to function `cudaMemAdvise()` [20, 27]. Value `cudaMemAdviseSetReadMostly` advises that it will mostly be read and rarely written to, `cudaMemAdviseSetPreferredLocation` sets the preferred location to the memory of the given device, and `cudaMemAdviseSetAccessedBy` allows to hint that the data will be accessed from a given device and causes the data to be mapped in the processor's page tables. It allows to prevent page faults [31].

In non-UM versions of the code, it is possible to arrange overlapping of computations and communication by launching sequences of copy, kernel execution and copy in various streams such that copy and execute operations in various streams can be overlapped. Streams can also be used with prefetching in the UM enabled code [27].

In paper [14], the authors investigated Unified Memory access performance in CUDA. Performed experiments used custom as well as Rodinia microbenchmarks run on a system with Xeon E5530 CPUs and NVIDIA K20c GPUs. It has been demonstrated that in order to see performance benefits from using UM, kernels should operate on subsets of output data at a time allowing the paging subsystem to come into play. Another advantage of using UM is for very complex data structures. Other scenarios result in better performance for the regular memory management approach. In work [22], the authors assessed relative performance of UM and non-UM versions of code for computation of scalar products. Kepler generation NVIDIA GeForce GTX 680 was used. The UM version resulted in 35% longer execution times. In paper [15], the authors demonstrate an average 10% loss of performance when using versions of benchmarks such as CUDA SDK's Diffusion3D Benchmark, Parboil Benchmark Suite and Matrix Multiplication ported to Unified Memory. Tests were performed on NVIDIA Kepler K40 and the TK1. It seems to be a reasonable performance penalty for potentially easier programming model. In paper [10], we investigated performance of Unified Memory versions versus standard memory implementations with `cudaMalloc()` for three applications with control and data flow characteristic of SPMD, geometric SPMD and divide-and-conquer paradigms. Parallel implementations of verification of Goldbach's conjecture, 2D heat transfer simulation and adaptive numerical integration were used. For the heat simulation, depending on the number of iterations per which data transfer was performed for visualization purposes, UM resulted in worse performance from about 1.5% for 50 iterations up to 8.7% for 10 iterations per visualization. For integration, UM versions resulted in worse performance as well, approximately 30% for integration of 100,000 subranges. Finally, for the implementation of Goldbach's conjecture, performance of UM versions was only up to 2% worse than the standard version.

Unified Memory can also be used in programs written using OpenACC [17, 25]. Article [25] presents a 2D Jacobi iteration code along with Unified Memory. A 7x performance improvement over an OpenMP multicore CPU version running on Intel Xeon E5-2698 version 3 is presented and no performance loss compared to a standard OpenACC version running on NVIDIA K40. For LULESH, OpenACC+Unified Memory offered 3.14x performance increase over OpenMP run on the multicore CPU and 92% performance of a standard OpenACC version. Article [26] shows further performance increase of the OpenACC+Unified Memory code run on NVIDIA Tesla Pascal P100 for 8.57x better than the CPU version.

Since OpenMP version 4.x support for GPU offloading has become available [5]. The authors of paper [18] have investigated usage and impact of Unified Memory when using at the level of OpenMP with latest offloading features. They have modified the OpenMP runtime in the LLVM framework in order for it to allocate data in Unified Memory. Tests were performed on SummitDev which has 54 nodes, each with 2 POWER8 CPUs and 4 Tesla P100 GPUs. Tests were performed using backpropagation, breadth-first search, CFD solver, K-means, kNN and speckle reducing anisotropic diffusion benchmarks. For applications with little data reuse, performance without Unified Memory is slightly better, as opposed to benchmarks with significant data reuse. For the latter, performance with Unified Memory is better to various degrees depending on input data size. Additionally, memory oversubscription is possible as an additional benefit to the standard implementation. Sometimes, though, as for BFS, CFD and SRAD it results in worse performance than for a CPU version. CPU codes for backpropagation and NN offered lower execution times than GPU versions. Usage of Unified Memory with OpenMP 4.5 and technical solutions were described in paper [7].

Support for GPU Unified Memory in the OmpSs model and Nanos runtime is discussed in work [30], along with its design and implementation as well as evaluation using microbenchmarks and Rodinia. For Rodinia benchmarks, speedups up to approximately 0.85 were obtained compared to pure CUDA versions.

Memory oversubscription has been studied in several papers in the literature. Paper [33] investigates low-level implementation of paged GPU memory. It proposes ways to improve performance of such a solution by combining replayable far-faults along with demand prioritized prefetching. The results show results close within 15% to best overlapped communication and execution version. For oversubscription, the authors claim that in general a random eviction algorithm performs very well to more complex strategies, considering overheads of the latter. In paper [13], the authors introduce GPUswap allowing relocation of application data from the GPU to system RAM allowing oversubscription of memory. At the time of the development and comparison with CUDA 6, the latter did not support memory oversubscription. Memory relocation delays for allocation of memory or giving up memory when another application is allocating memory are presented. Allocation delays are presented for various chunk sizes.

Work [32] discusses design and results of running HPGMG (high-performance geometric multigrid methods) on Pascal GPUs, including usage of Unified Memory. Specifically, throughput from using the oversubscription feature is provided using NVIDIA P100 (16 GB memory size) compared to the throughput for configurations fitting within the GPU memory. For an NVLink P100 and a basic version, a performance drop from 160 to approx. 55-75 MDOF/s was observed which was still 2.5x higher than for a 2 socket Intel E5-2630 version 3 system for large memory sizes. After applying optimizations such as data prefetching using `cudaMemPrefetchAsync()` and user hints results from around 175 MDOF/s for fitting within GPU memory to 55-135 MDOF/s were observed for memory sizes exceeding the GPU memory size.

In paper [2], it was shown how using multiple CUDA streams impacts performance of a GPU application that processes a stream of data chunks sent from the host. Specifically, considerable gains are shown for 2 streams compared to 1 stream, around 20–30% for compute intensities below 1 and up to 50% for larger compute intensities, among GPUs tested Tesla K20m, GTX 1060, GeForce 940MX and Tesla V100. Further increase between 3.3 and 4.9% is shown for 4 streams. Such data management optimization techniques can be especially profitable for frameworks for parallel data processing, such as in KernelHive [23], able to process part of data on CPUs and part of data on GPUs within a cluster.

Paper [6] proposes a CRUM (Checkpoint-Restart for Unified Memory) mechanism that allows forked checkpointing for Unified Memory with overlaps writing down a checkpoint during application execution. The work shows little overhead, 6% on average, for running parallel hybrid MPI+CUDA applications such as HPGMG-FV and HYPRE.

In paper [12], the authors assessed execution times with use of UM on NVIDIA Tegra K1. The work analyzed UM-aware versions of benchmarks such as pathfinder, needle, srad v2, Gaussian and lud Rodinia as well as Gauss–Seidel relaxation. As a conclusion, the authors stated that in the case of kernel time percentage lower that 60%, UM exhibited gains on the K1 platform.

The motivation of this work is to assess preferable ways of programming efficient parallel codes problems running on modern GPUs, especially to compare relative performance and ease of programming of standard and Unified Memory-based approaches. What is important, assessment is to be performed also for computations performed on very large data sets which do not fit into the memory size of a single GPU. The relatively new memory oversubscription feature has not yet been assessed thoroughly in the literature. Results of relevant experiments are of high importance because they are applicable to codes from many domains falling into the same processing paradigms as the tested applications.

3 Evaluation of modern Unified Memory features

3.1 Methodology

In order to assess Unified Memory performance, several applications have been implemented and tested on two modern systems with latest GPU architectures: GTX 1080 representing Pascal and V100 representing the Volta generation. This way we can assess whether the same or different behaviors of features can be observed on various architectures, adjusting input data sizes for oversubscription experiments. Various applications operating on data in parallel but with slightly different memory access patterns allow to evaluate performance of both the standard Unified Memory code and

Unified Memory with optimizations compared to the standard memory management approaches. Applications and corresponding memory access patterns include:

- image filters such as Sobel with application of 3 × 3 kernels on an image and image rotation with coalesced reads and non-coalesced writes,
- image stream processing with overlapping of computations and communication,
- fluid dynamic simulation with processing of a 2D space in successive iterations.

3.2 Applications

In order to assess Unified Memory performance, a set of common, representative, parallel applications was chosen. The main reason for such selection was the popularity and flexibility of the applications. Actually, several common parallel problems can be solved using one processing paradigm. Several simulation applications, image processing or math computations can be reduced to the same basic data model: a finite number of cells lined in 2 or 3 dimensions. In this model, adjacent cells need each other's data in order to calculate a final value.

Excellent examples are filters applied on images or specific kinds of simulations. However, the former requires only one or a few iterations (depends on the number of channels and the filter), while the latter is usually computationally demanding, memory consuming and requires at least hundreds of iterations. Of course, the filters can be applied on the videos, and as a consequence, image processing will also use a huge amount of memory and require many iterations. For the sake of efficiency, usually implementation of such processing incorporates usage of CUDA streams, in order to overlap time-consuming memory migration and computations.

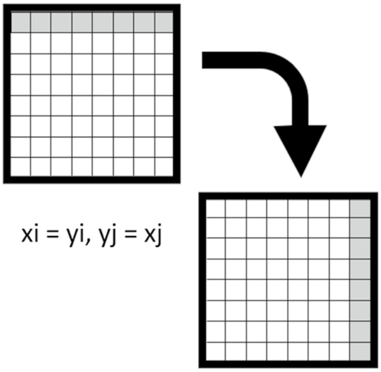
Another type of application is image rotation, which uses the GPU memory in a different manner that requires relocation of data with non-contiguous data access. A different way of reading memory can also affect Unified Memory's efficiency.

3.2.1 Sobel filter

A Sobel filter, sometimes called a Sobel–Feldman operator, is used for image processing, especially in edge detection algorithms. It creates more exposed edges. The authors of [29] presented an idea of a discrete differentiation operator. The principle is that at each point in the image, the result is estimation of the gradient obtained by the vector summation of the 4 possible central gradients in a 3 × 3 kernel. In practice, the value of the final gradient vector creates an image with large density of the edges; therefore, normalization of the vector is used. The basic version of the operator requires two versions of the 3 × 3 kernel: one for horizontal and another for vertical edges in the image:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Fig. 1 Image rotation: 90° to the right



The cells of the above matrix are multiplied by every area of the image, where a current pixel is assigned to the 1, 1 position in the matrix. Accordingly, the other cells correspond to its neighbors. As a consequence, a new value of the pixel is a result of the sum of all preceded multiplications. For the sake of simplicity, it can be reduced to the following formulas:

$$p_x = A[0,0] + 2 * A[0,1] + A[0,2] - A[2,0] - 2 * A[2,1] - A[2,2]$$

and

$$p_y = A[0,0] + 2 * A[1,0] + A[2,0] - A[0,2] - 2 * A[1,2] - A[2,2]$$

where A refers to a part of the image with size 3×3.

Consequently, the value of the pixel requires references to 8 neighbors. In addition, the whole gradient is calculated based on partial ones, as follows:

$$p = \sqrt{p_x^2 + p_y^2}.$$

3.2.2 Image rotation

Although image rotation is a straightforward computational problem, it requires specific access to the memory. Considering a naive solution, there are efficient coalescing reads and non-coalesced writes to the memory, which are not desired in parallel programming.

Figure 1 depicts the main idea of image rotation. In order to rotate an image to the right, position compounds of each pixel have to be reverted. Accordingly, in the case of rotation to the left, the indices should be swapped, with one small difference:

$$x_i = y_j, \quad y_j = n - x_j,$$

where n is the number of pixels in one horizontal row of the image.

Obviously, the algorithm could be more generic to rotate an image with an arbitrary angle with trigonometry equations. Nevertheless, it does not change the manner of accessing memory, but only increases the number of operations that should performed by a single thread to achieve the final result.

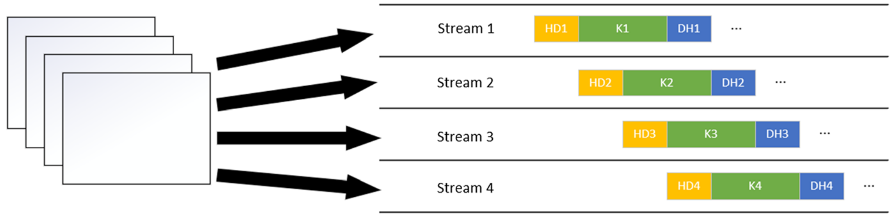


Fig. 2 Images are assigned to the streams

3.2.3 Streamed image processing

It is common to use the CUDA streams mechanism for parallel applications that require considerable amounts of data. The idea is to assign following images to different streams, as the memory migration and computations could be overlapped [16]. Figure 2 shows the general flow with standard copying.

The situation is different with regard to the basic Unified Memory usage. The pages with needed data are copied on demand; therefore, the case that memory migration and kernel execution are overlapped cannot happen. Hence, an advantage of applying streams into the application is not as significant as with standard memory copying. The reason for that should be linked to the overhead of the page faulting mechanism. Introducing prefetching enabled using Unified Memory with overlapping. However, a new overlapping strategy has to be applied. The idea from Fig. 2 cannot be performed with memory managed by the driver due to CPU blocking steps (mainly virtual address map management). Therefore, it is important to assign different streams for data migration and computations. Of course, proper synchronization is needed.

3.2.4 Fluid dynamic simulation

The aim of fluid dynamic simulation is to resolve the process of fluid flow, using numerical methods. Thanks to discretization and numerical solving differential equations, the approximate values of velocity or pressure of the issued fluid are found. In this particular case, the Navier–Stroke equations were used in order to create a model of the simulation. In addition, the FDM[1] method was used to calculate differentials with a GPU. The Navier–Stroke equations are the most common solution to describe physical phenomena, when considering fluids:

$$\frac{\partial u}{\partial t} = -(u * \Delta)u - \frac{1}{\rho}p + \nu\Delta^2 u + F$$

and

[1] Finite Difference Method (FDM) numerical method used for solving differential equations. Discretization method using Taylor series.

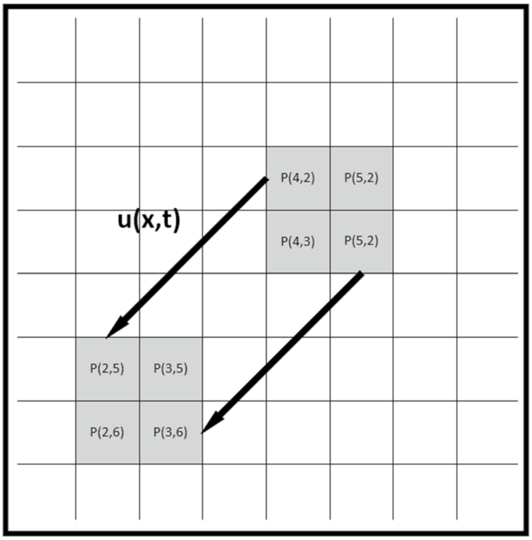


Fig. 3 Fluid fraction migration scheme

$$\varDelta * u = 0$$

where:

ρ fluid density (const.),
v kinematic viscosity,
F external forces (vector quantity),
p pressure,
$\varDelta$ (Nabla operator) three different use means: gradient, divergence and Laplace operator,
u velocity (vector quantity).

The equations describe the fluid flow in time [8]. There are a few assumptions concerning the fluid characteristics: incompressibility and fluid homogeneity. A combination of both attributes means that density is constant in time and space. The following phenomena were implemented: advocation, pressure, diffusion, external forces.

This approach was implemented with a solution based on a mesh. It means that the finite area is used for the simulation, where some fluid with initial values is placed. After that, every cell in the area is processed in order to calculate new positions. Setting a given δt allows to calculate the state of the fluid at arbitrary time (Fig. 3). In practice, next to δt, the number of iterations is also set after which the simulation should be finalized.

Table 1 Specifications of tested platforms

Hardware	Platform 1	Platform 2
CPU	Intel Core i7-8700K (6 cores, 12 threads) 4.7 GHz@3.7 GHZ	Intel Xeon E5-2698 version 4 (20cores, 40 threads) 2.20 GHz @ 2.20 GHz
RAM	16 GB DDR4	256 GB DDR4
GPU	Zotac GTX 1080 mini 8 GB GDDR5	Nvidia Tesla V100 16 GB HBM2
System	Ubuntu 16.04 LTS	Ubuntu 16.04 LTS
CUDA version	9.0	9.0

Table 2 Specifications of tested GPU

	GTX 1080	V100
CUDA capability	6.0	7.1
Core frequency	1600 MHz	1500 MHz
Bus width	256 bits	4096 bits
Memory bandwidth	320 GB/s	900 GB/s
CUDA cores	2560	5120 + 640 tensor cores
Number of SMs	20	80

4 Experiments

4.1 Test platforms

For benchmarking purposes, two different platforms were used with parameters as shown in Tables 1 and 2. Each test was executed 5 times. Average results were presented as gray and black boxes. Moreover, standard deviations of the samples were added as bars for each result in relevant figures. In some cases, deviation reached 7%, but overall it did not influence observations and conclusions.

4.2 Tests

This section presents final results of performed experiments. Each figure depicts execution times for various versions of a given application. Three versions can be distinguished:

- standard implementation uses standard, explicit copying between host and device memories,
- UM implementation uses a basic Unified Memory mechanism,
- UMopt implementation uses Unified Memory with prefetching.

```
void runKernelsWithPrefetching()
{
    //init data
    char* dataIn,dataOut;
    cudaMallocManaged(&dataIn, ...);
    cudaMallocManaged(&dataOut, ...);
    cudaMemAdvise(dataIn, ..., PreferredLocation, CPUDeviceId); //to
        init data on CPU,
    cudaMemAdvise(dataOut,..., PreferredLocation, GPUDeviceId); //
        results on GPU

    initData(dataIn);

    cudaMemPrefetchAsync(...); // host to gpu
    kernel<<<...>>>(dataIn,dataOut,...);
    cudaMemPrefetchAsync(...); // gpu to host
}
```

Fig. 4 Standard prefetching

```
void runKernelsInStreamsWithPrefetching()
{
    // prepare data
    int streamNum = ...;
    createStreamsAndEvents();

    cudaMemPrefetchAsync(..., GPU_ID,streams[1]);
    cudaEventRecord(events[0], streams[1]);

    for(int i = 0; i < KER_NUM; ++i)
    {
        int s_id = i % streamNum;
        int s2_id = (i + 1) % streamNum;

        cudaEventSynchronize(events[s_id]);
        cudaEventSynchronize(events[s2_id]);

        kernel<<<..., streams[s_id]>>>(...);
        cudaEventRecord(events[s_id], streams[s_id]);

        if(i < KER_NUM - 1)
        {
            cudaStreamSynchronize(streams[s2_id]);
            cudaMemPrefetchAsync(..., GPU_ID, streams[s2_id]);
            cudaEventRecord(events[s2_id], streams[s2_id]);
        }
        cudaMemPrefetchAsync(...,CPU_ID, streams[s_id]);
    }
}
```

Fig. 5 Second version with prefetching

Our implementations of prefetching are application specific. The one without explicit streams is shown in Fig. 4.

This prefetching implementation was used for single-image processing tests. In order to overlap data migration with kernel execution, the implemented approach, similarly to [27], is more complex as shown in Fig. 5. Copying data to a GPU and

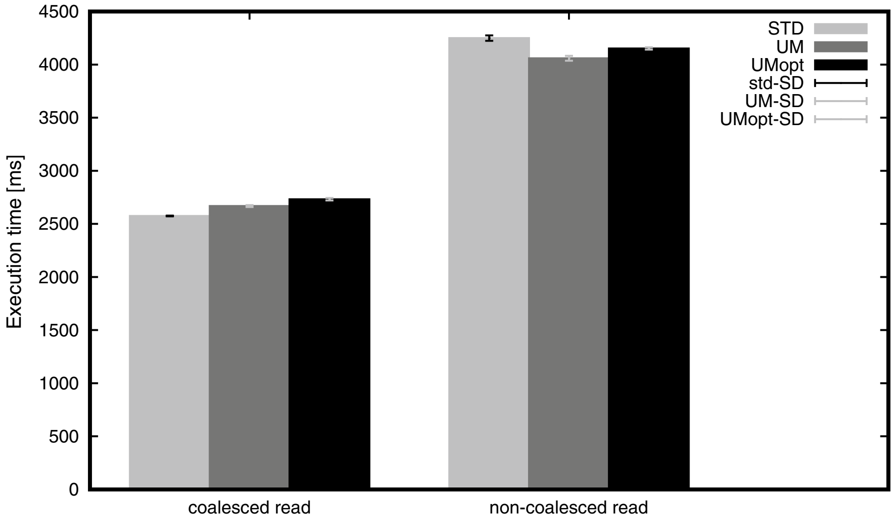


Fig. 6 Sobel filter application on GTX 1080

computations use different streams. Of course, appropriate synchronization between streams is performed to make sure the data were copied before kernel execution. The implementation requires at least two streams: one for data migration to the GPU and another one for kernel execution and data migration to the CPU. Tests have shown that the optimal number of streams is five. Further increasing did not bring any visible improvements. This is in line with observations in [2] where differences between performance of 2 and 4 stream benchmarks were already small. The approach was used for tests with image processing using streams and fluid simulation.

Profiling applications have shown that GPU page faults were decreased by about 50% only in the cases where the streams were used. The standard approach did not bring any visible differences.

4.2.1 UM optimizations: prefetching

Firstly, implementation for processing of a single image was tested with different methods of using memory:

- Sobel filter implementation with coalesced and non-coalesced memory reads (Figs. 6 and 7),
- image rotation (Fig. 8).

Tests used a single image of size: 30,000 × 30,000 × 3 (height × width × number of channels). Results indicate a visible impact of specific architectures on the performance of code with Unified Memory. There is a minor boost or very similar execution time for the GTX 1080. An exception is the rotation image processing using global memory, for which the Unified Memory

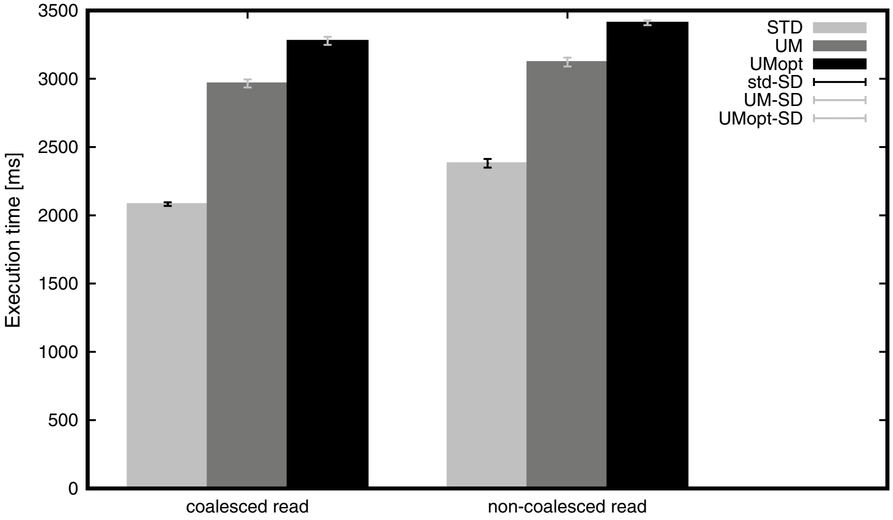


Fig. 7 Sobel filter application on V100

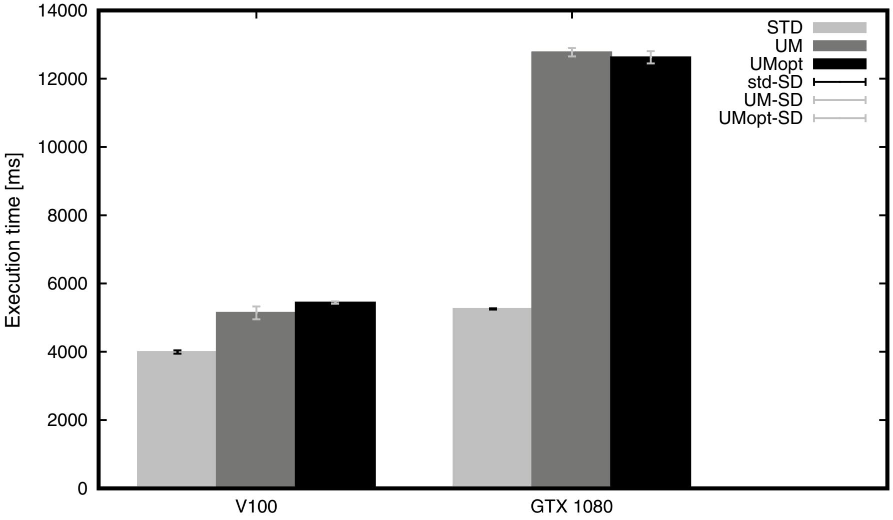


Fig. 8 Image rotation application for both GPUs

implementation is more than 2 times slower. The tests on V100 present performance decrease in each case for standard memory management and no gains from prefetching for UM.

Potential of the newly introduced memory prefetching with streams has been confirmed by the following tests. The Sobel filter kernel was applied on a group of

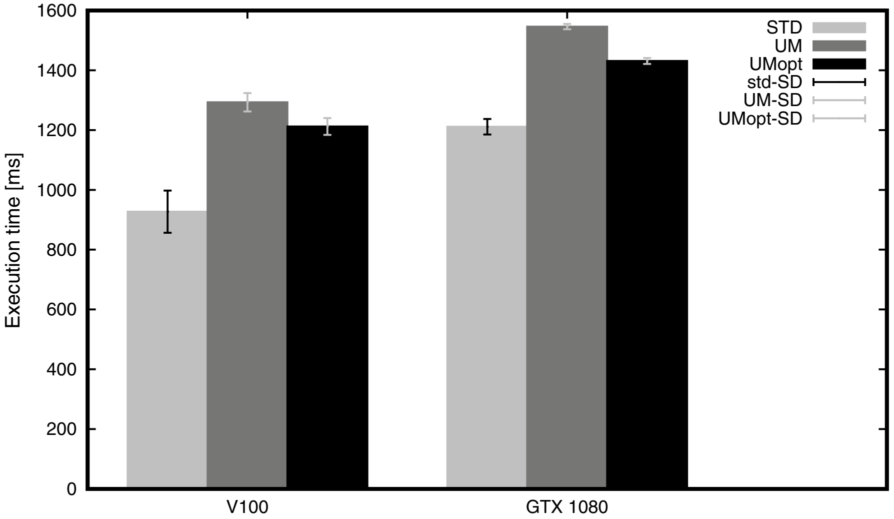


Fig. 9 Image processing using streams

images that were assigned to streams (Fig. 9). Results present a visible advantage of the implementation with prefetching for both GPUs.

The dynamic fluid simulation was performed with 100 iterations for a 3D mesh of size 200 × 200 × 200 using a single-precision float data type. Subsequent tests investigated correlation of balance between computation and data migration, as shown in Figs. 10 and 11. Therefore, a constant number of iterations was mixed with various numbers of memory copies that can be used for data visualization. In each case, prefetching resulted in performance improvement in the Unified Memory implementation. Moreover, the latter was even quicker compared to standard memory copies for small memory size migrations. Improvement in the case of UMopt compared to UM varied between 26% (100 copies for 100 iterations) and 1% (1 memory copy for 100 iterations).

4.2.2 UM oversubscription

Various types of implementation using the oversubscription mechanism were investigated for processing a single image first:

- standard single processing of a single image with standard memory copying. The size of an image was chosen in order to exceed a GPU memory—5 GB/10 GB memory was allocated for an input image and the same size buffer for an output image accordingly for GTX 1080/V100. The implementation includes streams and allocation of all available GPU memory. The solution requires to create a portion of data that fits within the allocated memory, copying to the global memory, processing and fetching to the RAM. The whole process is repeated until all

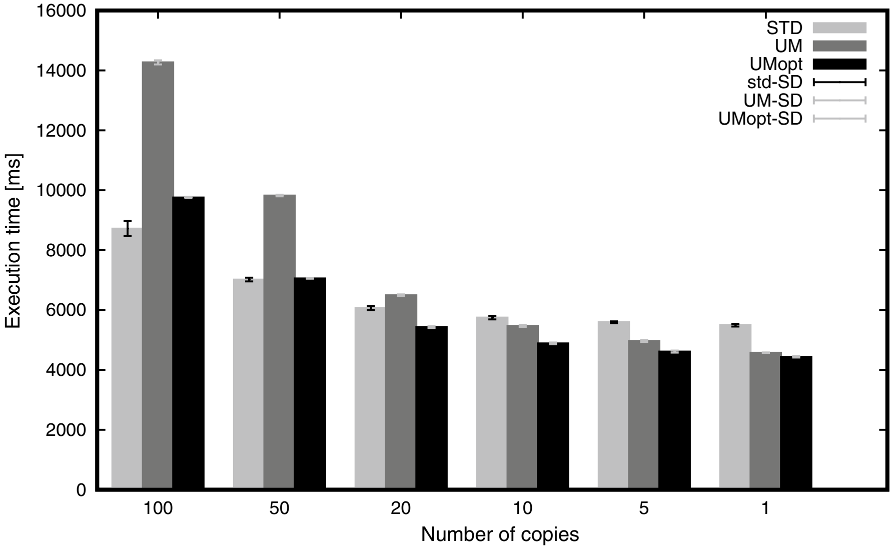


Fig. 10 100 iterations of dynamic fluid simulation on V100

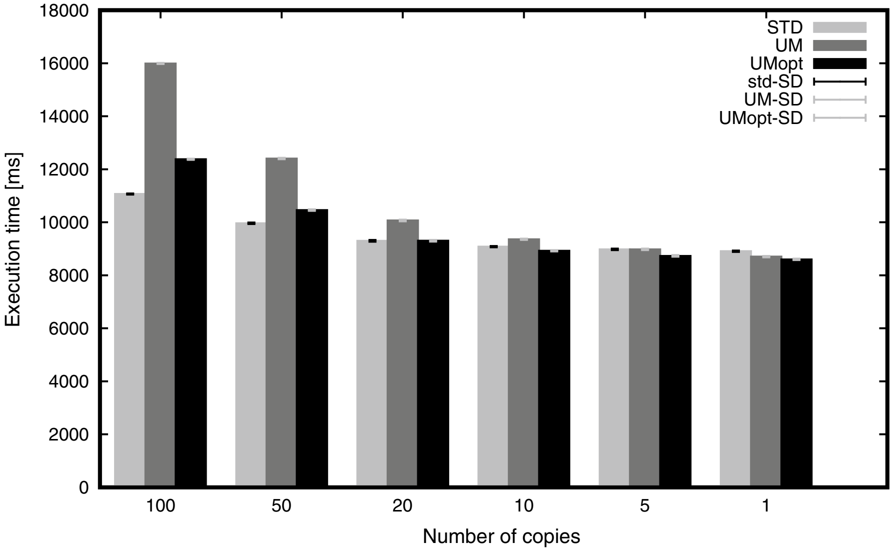


Fig. 11 100 iterations of dynamic fluid simulation on GTX 1080

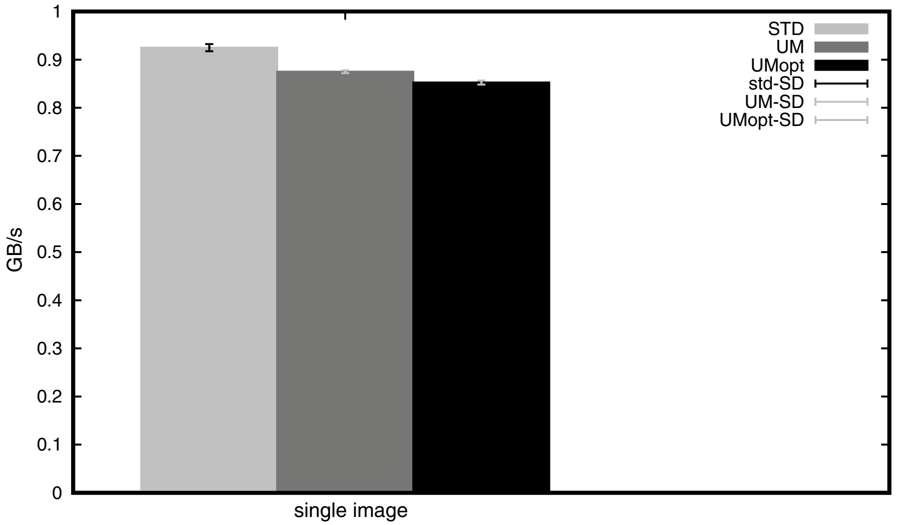


Fig. 12 Oversubscription mechanism for a single image (42, 303 × 42, 303 × 3) with 10 GB allocation on GTX 1080

data have been processed. Programmatically, such an approach is quite complex and error-prone.

- UM single processing of a single image with Unified Memory. The solution simply allocates all needed memory in order to process the image further. Memory migration is performed implicitly by the driver.

We have also investigated performance of stream-aware multi-image processing. The following configurations were tested:

- standard multiple multi-image processing with standard memory copying. The number of 1920 × 1080 × 3 images was adjusted to exceed a GPU memory size. The solution is similar to the single-image approach, but each image is assigned to one of the five streams.
- UM multiple multi-image processing with Unified Memory. The assumptions are the same as those of standard multiple multi-image processing. The difference is that there is no need for explicit copying of data for the each image. Such an approach has also been benchmarked with memory prefetching (in order to minimize page faults).

Results for the GTX 1080 are presented in Figs. 12 and 13 for single- and multi-image configurations, respectively. Results for the V100 are shown in Figs. 14 and 15 for single- and multi-image configurations, respectively. Overall, oversubscription with Unified Memory is slower compared to a proper implementation with explicit data migration. Optimization with prefetching brought 5–6% improvement for the multi-image processing (Figs. 13 and 15). In case of the

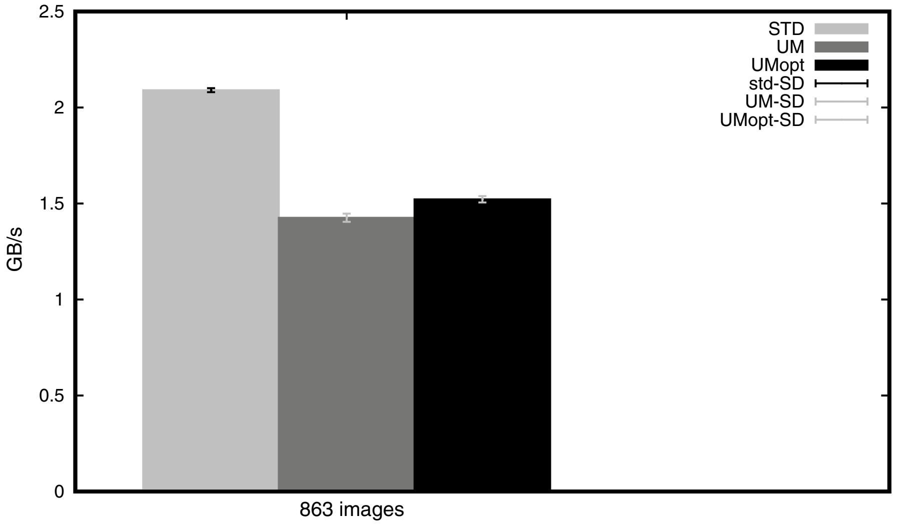


Fig. 13 Oversubscription mechanism for 863 images (1920 × 1080 × 3) with 10 GB allocation on GTX 1080

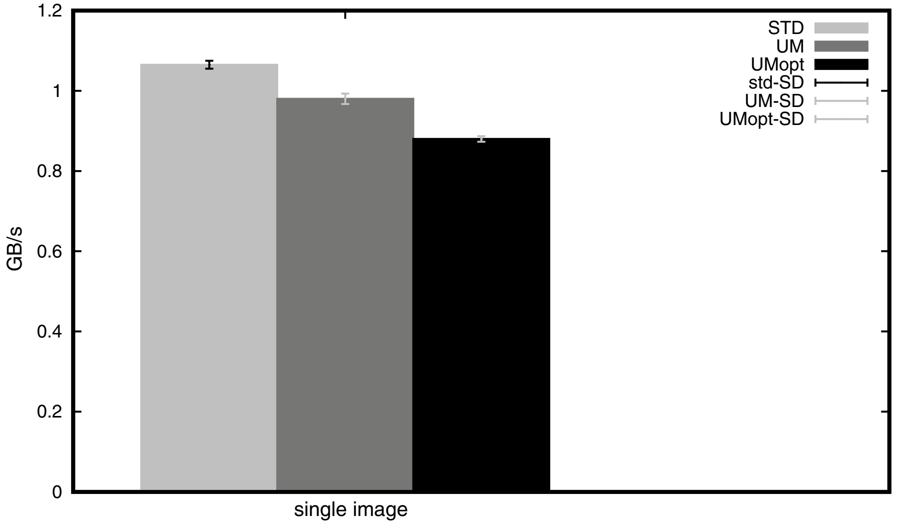


Fig. 14 Oversubscription mechanism for a single image (59, 826 × 59, 826 × 3) with 20 GB allocation on V100

Volta architecture, oversubscription with streams was even slightly better than the version with explicit data migration (Fig. 15).

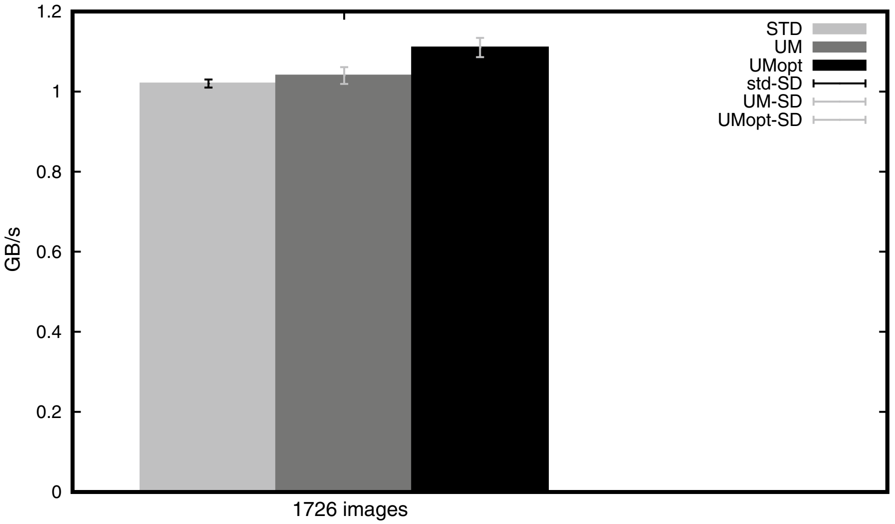


Fig. 15 Oversubscription mechanism for 1726 images (1920 × 1080 × 3) with 20 GB allocation on V100

4.3 Discussion

The results have shown practical differences between the Pascal GTX 1080 dedicated to desktop user platforms and the high-end Volta V100 released for professional purposes. Firstly, we can see much smaller differences between coalesced and non-coalesced memory accesses for V100 compared to the GTX 1080. We should note the 4096 bits memory bus compared to 256 bits for the two GPUs, respectively.

Secondly, an interesting correlation can be noticed in the fluid simulation figures (Figs. 10 and 11). The larger the number of copies was taken during a simulation, the worse the results Unified Memory had. With about 1–5 copies, Unified Memory has better performance than explicit memory copying, which suggests that the mechanism is more suited for applications that are more computationally intensive. It may result from Unified Memory specifics and its page faulting system. Moreover, prefetching for UM resulted in a visible performance boost for all cases.

Tests undoubtedly indicated the type of applications that can really benefit from memory prefetching when Unified Memory is used. Overall, it brings benefits when streams are used. It allows to overlap memory migration and computations. In the standard approach, the driver responsible for memory management is not aware of the order of the data usage in further steps. Therefore, it is recommended to provide a hint by explicit invocation of prefetching. Eventually, applications like simulations and video processing would benefit the most.

The oversubscription mechanism is a significant improvement when ease of parallel programming is considered. The programmer does not have to bother with complicated memory migration for huge amounts of data. Depending on a GPU, it can either result in a significant decrease in performance (a totally naive UM version versus streamed standard copying) by about even 30% or offer even marginally faster execution (optimized UM streamed images processing versus streamed standard copying images processing on V100). Practically, it depends on programmer skills how much optimized code they have created.

5 Summary and future work

Within this paper, performance of Unified Memory was assessed using the following applications: image processing including the Sobel filter and rotation, streamed image processing and fluid dynamic simulation. We compared performance of standard Unified Memory implementations as well as Unified Memory with data prefetching to standard implementation with implicit memory copying. Furthermore, we investigated performance of applications using Unified Memory for multistream codes as well as using Unified Memory with memory oversubscription. We have presented conclusions on relative performances of the implementations for particular application types and have shown differences for two modern GPU models—desktop Pascal series NVIDIA GTX 1080 and server Volta series NVIDIA V100 GPUs.

We have concluded that Unified Memory generally results in worse performance than the standard memory management approach offering the benefit of easier programming, and Unified Memory implementation also allows performance improvements through programmer-assisted data prefetching. However, in our experiments it resulted in better results than standard Unified Memory only for selected applications—namely multi-image processing with streams as well as for the fluid dynamic simulation. Unified Memory can bring better results than explicit memory copying for simulation codes such as fluid dynamic implementations with a relatively large compute-to-communication ratio.

In the future, we plan to extend this research toward systems with more GPUs as well as incorporation of UM into previous hybrid CPU+GPU implementations [4]. Another direction of research will include testing impact of various host architectures on performance of GPU processing. Recently, we have tested impact of IBM's AC922 system architecture compared to NVIDIA's DGX station, both with 4 NVIDIA V100 GPUs on the performance of training deep neural networks [24]. We plan to make a similar comparison using various host architectures for the applications presented in this work, using Unified Memory.

References

1. Ashcraft MB, Lemon A, Penry DA, Snell Q (2017) Compiler optimization of accelerator data transfers. Int J Parallel Prog. https://doi.org/10.1007/s10766-017-0549-3

2. Czarnul P (2018) Benchmarking overlapping communication and computations with multiple streams for modern gpus. In: Ganzha M, Maciaszek LA, Paprzycki M (eds) Communication Papers of the 2018 Federated Conference on Computer Science and Information Systems, FedCSIS 2018, Poznań, Poland, September 9–12, 2018, pp 105–110
3. Czarnul P (2018) Parallel programming for modern high performance computing systems, 1st edn. Chapman and Hall/CRC, Taylor&Francis, Boca Raton
4. Czarnul P (2018) Parallelization of large vector similarity computations in a hybrid cpu+gpu environment. J Supercomput 74(2):768–786. https://doi.org/10.1007/s11227-017-2159-7
5. Finkel H, Sharif H (2017) Openmp, unified memory, and prefetching. PADAL17: 2017-08-03, Exascale Computing Project. https://www.bnl.gov/compsci/docs/Hal-Finkel-padal_2017.pdf
6. Garg R, Mohan A, Sullivan M, Cooperman G (2018) CRUM: Checkpoint-Restart Support for CUDA's Unified Memory. ArXiv e-prints
7. Grinberg L, Bertolli C, Haque R (2017) Hands on with openmp4.5 and unified memory: developing applications for ibm's hybrid cpu + gpu systems (part ii). In: de Supinski BR, Olivier SL, Terboven C, Chapman BM, Müller MS (eds) Scaling openMP for exascale performance and portability. Springer International Publishing, Cham, pp 17–29
8. Harris MJ (2007) Fast fluid dynamics simulation on the gpu. http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch38.html
9. Hindriksen V (2013) Cuda 6 unified memory explained. http://streamcomputing.eu/blog/2013-11-14/cuda-6-unified-memory-explained/. Accessed 17 Feb 2016
10. Jarząbek Ł, Czarnul P (2017) Performance evaluation of unified memory and dynamic parallelism for selected parallel cuda applications. J Supercomput 73(12):5378–5401. https://doi.org/10.1007/s11227-017-2091-x
11. Ji F, Lin H, Ma X (2013) Rsvm: a region-based software virtual memory for gpu. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT '13. IEEE Press, Piscataway, NJ, USA, pp 269–278. http://dl.acm.org/citation.cfm?id=2523721.2523758
12. Joseph J, Keville K (2015) An evaluation of cuda unified memory access on nvidia tegra k1. Waltham, MA USA. IEEE High Performance Extreme Computing Conference (HPEC '15) Nineteenth Annual HPEC Conference
13. Kehne J, Metter J, Bellosa F (2015) Gpuswap: enabling oversubscription of gpu memory through transparent swapping. In: Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '15. ACM, New York, NY, USA, pp 65–77. https://doi.org/10.1145/2731186.2731192
14. Landaverde R, Zhang T, Coskun AK, Herbordt M (2014) An investigation of unified memory access performance in cuda. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–6. https://doi.org/10.1109/HPEC.2014.7040988
15. Li W, Jin G, Cui X, See S (2015) 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. https://doi.org/10.1109/CCGrid.2015.105
16. Malinowski A, Czarnul P (2018) A solution to image processing with parallel MPI I/O and distributed NVRAM cache. Scalable Comput Pract Exp 19(1):1–14. https://www.scpe.org/index.php/scpe/article/view/1389
17. Miles D (2017) Openacc and unified memory. Cray User Group Meeting, Redmond, Washington. https://cug.org/proceedings/cug2017_proceedings/includes/files/ven112s1.pdf
18. Mishra A, Li L, Kong M, Finkel H, Chapman B (2017) Benchmarking and evaluating unified memory for openmp gpu offloading. In: Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC'17. ACM, New York, NY, USA, pp 6:1–6:10. https://doi.org/10.1145/3148173.3148184
19. Negrut, D., Serban, R., Li, A., Seidl, A.: Unified memory in cuda 6.0. a brief overview of related data access and transfer issues. In: Tech. Rep. TR-2014-09, University of Wisconsin–Madison (2014)
20. NVIDIA: CUDA Toolkit Documentation. CUDA Runtime API (2018). V 10.0.130. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html
21. NVIDIA: Cuda c programming guide (2019). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

22. Pirja A, Petrosanu M (2014) Improving parallel programming in the compute unified device architecture using the unified memory feature. ftp://ftp.repec.org/opt/ReDIF/RePEc/rau/jisomg/WI14/JISOM-Wi14-A14.pdf
23. Rościszewski P, Czarnul P, Lewandowski R, Schally-Kacprzak M (2019) Kernelhive: a new workflow-based framework for multilevel high performance computing using clusters and workstations with cpus and gpus. Concur Comput Pract Exp(9):2586–2607. https://doi.org/10.1002/cpe.3719. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3719
24. Rościszewski P, Iwański M, Czarnul P (2019) The impact of the ac922 architecture on performance of deep neural network training. In: Proceedings of the 2019 International Conference on High Performance Computing & Simulation (HPCS 2019). Dublin, Ireland. In press
25. Sakharnykh N (2015) Combine openacc and unified memory for productivity and performance. https://devblogs.nvidia.com/parallelforall/combine-openacc-unified-memory-productivity-performance/
26. Sakharnykh N (2016) Beyond gpu memory limits with unified memory on pascal. https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/
27. Sakharnykh N (2017) Maximizing unified memory performance in cuda. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/
28. Sakharnykh N (2017) Unified memory on pascal and volta. http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf
29. Sobel I (2014) An isotropic 3 × 3 image gradient operator. Presentation at Stanford A.I. Project 1968
30. Soto AR (2017) Design and development of support for gpu unified memory in ompss. Master's thesis, Universitat Polit'ecnica de Catalunya. https://upcommons.upc.edu/bitstream/handle/2117/112437/126955.pdf
31. Unified memory on p9+v100 (2018) ORNL workshop. https://www.olcf.ornl.gov/wp-content/uploads/2018/03/ORNL_workshop_mar2018.pdf
32. Wang P, Sakharnykh N (2016) Hpgmg performance on pascal gpu architecture. https://crd.lbl.gov/assets/Uploads/SC16-HPGMG-BoF-NVIDIA.pdf
33. Zheng T, Nellans D, Zulfiqar A, Stephenson M, Keckler SW (2016) Towards high performance paged memory for gpus. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp 345–357. https://doi.org/10.1109/HPCA.2016.7446077

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.