Noname manuscript No.
(will be inserted by the editor)

Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications

Łukasz Jarząbek · Paweł Czarnul

Received: date / Accepted: date

Abstract The aim of this paper is to evaluate performance of new CUDA mechanisms – Unified Memory and Dynamic Parallelism for real parallel applications compared to standard CUDA API versions. In order to gain insight into performance of these mechanisms, we decided to implement three applications with control and data flow typical of SPMD, geometric SPMD and divide-and-conquer schemes, which were then used for tests and experiments. Specifically, applications tested include verification of Goldbach's conjecture, 2D heat transfer simulation and adaptive numerical integration. We experimented with various ways of how Dynamic Parallelism can be deployed into an existing implementation and further be optimized. Subsequently we compared best Dynamic Parallelism and Unified Memory versions to respective standard API counterparts. It was shown that usage of Dynamic Parallelism resulted in improvement of performance for heat simulation, better than static but worse than an iterative version for numerical integration and finally worse results for Golbach's conjecture verification. Unified Memory in most cases results in decrease in performance. On the other hand, both mechanisms can contribute to simpler and more readable codes. For Dynamic Parallelism, it applies to algorithms in which it can be naturally applied. Unified Memory generally makes it easier for a programmer to enter the CUDA programming paradigm as it resembles the traditional memory allocation/usage pattern.

Keywords CUDA · Dynamic Parallelism · Unified Memory · Parallel Programming

Ł. Jarząbek, P. Czarnul
Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology,
Poland
E-mail: lukjar92@gmail.com, pczarnul@eti.pg.gda.pl

1 Introduction

Recently, it can be noticed that heterogeneous computer systems have gained more and more popularity. Almost every user of a personal computer, beyond the standard CPU device, has an additional compute device with significant computing power which is the GPU. Because of such popularity and tremendous potential of such devices, it is crucial to be able to make the most of them and be able to assess how beneficial new features of the technology are. Especially important is the fact, that the GPU architecture makes it ideal for parallel processing of huge data sets. An example can be, very popular nowadays, deep learning. Usage of GPU can even save days, during neural network training [26]. For these reasons new tools and platforms are released, to provide better and simpler ways to create applications and programs. It is particularly important for people whose main profession is not programming i.e. domain specialists.

NVIDIA is one of the main players in the HPC market. Not so long ago NVIDIA introduced new mechanisms into the CUDA API – Unified Memory (UM) and Dynamic Parallelism (DP).

Typically, first CUDA based applications would allocate memory and initialize data in RAM, allocate memory on a GPU device, copy input data to a GPU global memory via PCI Express, run a kernel function on the GPU and copy results from the GPU back to the RAM on the host. This may be repeated several times as multiple kernels are invoked. Typical optimizations include overlapping communication between a host and a device and computations on the GPU and possibly the host, using streams. Unified memory, available in CUDA 6 and later versions, introduced the concept of managed memory, visible from both a host and a GPU, without the need for manual copying between memories of the two sides [20]. Migration of pages is performed by an underlying runtime system, transparently to the programmer. As a result, the programming model has been greatly simplified and requires just allocation in managed memory using `cudaMallocManaged(...)`, invocation of a kernel and synchronization upon kernel termination.

Dynamic parallelism, on the other hand, is a new mechanism introduced in CUDA 5 (for devices with compute capability 3.5+) that allows launching kernels from within kernels [20]. Recursive calls may continue up to 24 levels. This solution is well suited for divide and conquer applications [4] as no explicit synchronization through the host is needed before next kernel calls. Specifically, this allows recursive deepening in certain algorithms to increase resolution in computations in geometric SPMD applications [19], numerical applications such as adaptive integration [6] and others.

Currently there are some works which are focused on performance evaluation of these new CUDA APIs, either UM or DP. Compared to these works, described in Section 2, this paper analyzes three parallel applications, falling into either geometric SPMD [7], general SPMD [28] and divide-and-conquer [28,4] processing paradigms and compares effects of UM and DP for each of these applications. It is possible that one of the mechanisms does not bring

measurable benefits while the other does. This paper contains results of several experiments that allow to draw conclusions. We decided to implement the following three parallel programs, in various versions with and without the proposed mechanisms:

1. Heat transfer simulation in 2D space – an example of a geometric SPMD application.
2. Adaptive integration which uses a trapezoidal rule – an example of a divide and conquer approach.
3. Verification of Goldbach's Conjecture – requires checking a hypothesis for a set of even numbers i.e. an SPMD problem in its nature.

The outline of the paper is as follows. Section 2 includes related work regarding both Unified Memory and Dynamic Parallelism and how these affected performance and code readability of specific applications. Section 3.1 outlines methodology used during comparisons for UM and DP for testbed applications which are described next with basic and optimized versions along with comparison of performance, discussion of best settings and comments on how these affect readability of the code. Parallel applications include: heat distribution in Section 3.2, adaptive numerical integration in Section 3.3 and Goldbach Conjecture verification in Section 3.4. Finally, Section 4 includes a summary based on previously obtained results with conclusions on how applications might benefit from UM and/or DP.

2 Related Work

2.1 Unified memory

The Unified Memory mechanism might impact performance, compared to a standard implementation without it. Typically, solutions that increase flexibility and ease of programming impose a certain performance overhead.

The authors of [13] thouroughly tested the UM mechanism. They incorporated several benchmarks, both those written by the authors but also the Rodinia benchmark set. The latter is a set targeted for testing heterogeneous environments. The authors modified selected tests so that the latter use UM. Unfortunately, results of experiments revealed that typically UM would yield worse results compared to the standard approach, with manual management of memory. However, for individual tests it was demonstrated that application of UM may bring performance benefits. Specifically, if a subset of data is queried by multiple kernels multiple times before some other data is accessed. The authors state that in such a case the UM mechanism can place data favourably which brings benefits compared to the standard API. As data size is increased this benefit decreases. Furthermore, it was stated that for most applications complexity of code does not change considerably. For complex data structures, however, UM may make programming easier.

In paper [18] authors focused on solving systems of sparse linear equations and the algorithm implemented in the SPIKE::GPU library. The first step of the algorithm changes columns and rows of a matrix which makes further processing easier. It is composed of a few steps a part of which are executed on a GPU and a part on a CPU. Specifically there are 4 steps two of which are executed on a host. UM was deployed at this step. According to the authors, application of UM made the code clearer. Out of more than 120 large matrices, for more than a half the UM based version exhibited better performance.

Article [23] describes how to use UM along with parallelization using the OpenACC API for codes such as Jacobi iteration. Using the PGI compiler it just requires a compilation option to enable UM. The article shows the speed-up of around 30 on an NVIDIA K40 GPU over a single threaded CPU run on Intel Xeon E5-2698 v3 and around 7 over a CPU run using 8 threads.

Paper [15] evaluated UM by comparing a selected set of applications without and with UM run on NVIDIA K40 and Jetson TK1. The applications tested were: Diffusion3D Benchmark, Parboil Benchmark Suite and Matrix Multiplication. The UM enabled codes exhibited around 10% worse performance coming from additional memory transfers. This turned out to be the cost for an easier programming model.

In work [12] authors evaluated performance of UM on NVIDIA Tegra K1. They wrote code for Gauss-Seidel relaxation and a benchmark that increments data in buffers. Apart from that they ported pathfinder, needle, srad_v2, gaussian and lud Rodinia benchmarks to use UM. The authors concluded that if time spent in a kernel was smaller than 60% (which means that communication was a significant part) gains from UM were visible given the architecture of Tegra K1.

2.2 Dynamic Parallelism

In the case of DP it might be expected that, at least for a selected class of problems such as divide-and-conquer ones, lack of need for synchronization with the host between kernel invocations would result in performance benefits.

Two clustering algorithms: K-means and hierarchical clustering were investigated in work [8]. The K-means algorithm, due to dependencies between data, requires synchronization between iterations. On the other hand, hierarchical clustering can be naturally mapped to the divide and conquer scheme. In the case of the K-means implementation, a slight drop in performance was observed in the DP version. However, for hierarchical clustering a DP enabled version demonstrated considerable speed-up compared to a standard version without DP. Additionally, DP resulted in much clearer code. As a conclusion, DP is well suited for algorithms processing tree-like arranged data sets.

In publication [1] DP was used for generation of Mandelbrot fractals. In a first approach threads are assigned to individual points in space. This may result in loss of computational power as some points may require fewer or more computations. In a second approach, based on a Mariani-Silver algorithm,

space is divided adaptively within threads using DP. As a result speed-up with this solution was between 1.3 and 6 times depending on space size with higher speed-ups for larger space sizes.

Furthermore, demonstration of DP benefits is included in publication [11] for the quicksort algorithm. In this case, it allowed doubling performance with decreasing the code size at the same time. Again it resulted in clearer code, similarly to previous works.

Papers [21,22] investigated performance of DP based implementations of a tree search algorithm with irregular workloads, based on the N queens problem. As the authors concluded, after tests for this application using NVIDIA K20x, overheads for kernel invocations and dynamic memory allocation resulted in overall higher execution times for a few DP enabled codes compared to times for a basic, reference GPU implementation.

In paper [2] authors demonstrated benefits of a DP enabled version for a Conjugate Gradient (CG) method for iterative solving of sparse linear systems. It has been shown that a DP enabled version resulted in savings of around 3.6% in execution time and 14.2% in energy consumption compared to previous implementations. Tests were performed on an NVIDIA K20c GPU.

In paper [29] authors analyzed performance of a DP enabled algorithms for Breadth First Search (BFS) and Single-Source Shortest Paths (SSSP) algorithms compared to other existing implementations showing performance better than some but not the best (compared to algorithms with advanced queueing for SSSP) results.

Authors of [14] state that they obtained over 2.6 speed-ups for SSSP and over 1.4 for Sparse Matrix-Vector Multiplication (SpMV) codes compared to basic implementations without DP. Tests were run on an NVIDIA K20 GPU.

In paper [17] authors presented comparison of a CPU, naive GPU, tiled GPU and DP enabled GPU implementation of an Inverse Distance Weighting (IDW) interpolation algorithm. Results show that the DP enabled version performed consistently worse than the best tiled GPU implementation for this problem.

In paper [27] authors analyzed performance, overheads and memory footprint of DP enabled codes with a variety of benchmarks such as Adaptive Mesh Refinement, Barnes Hut Tree, Breadth-First Search, Graph Coloring, Regular Expression Match, Product Recommendation, Relational Join, Single-Source Shortest Path. The authors performed benchmarking that allowed to compute potential speed-ups of DP enabled codes by first calculating ideal execution times by subtracting kernel launch overheads. A conclusion is that there is a speed-up potential between 1.13 and 2.73 but current kernel launch overheads would result in average execution times slower than around 1.2 of implementations without DP.

Work [16] focuses on CUDA DP for low power ARM based prototypes demonstrating 20% savings in energy consumption on a Pedraforca prototype.

Work [25] focuses on normalization of Gene Expressions using a GPU with DP and comparison of such an implementation run on NVIDIA K20c compared to Intel Xeon E5650. The final version including CPU-GPU communication

and all steps of the procedure amounted to the speed-up of around 4.8. It is noted that the speed-up for quicksort, which is a part of the whole procedure, is over 22.5 for the DP enabled implementation on a GPU compared to a CPU.

3 Evaluation of Unified Memory and Dynamic Parallelism

3.1 Methodology

We implemented each application with and without the DP and UM mechanisms. Then we compared standard versions using the standard API with the other two applications. The next chapter contains short descriptions of each program and performed experiments. Each section, beyond test results, contains respective conclusions and discussion.

3.1.1 Test platforms

A workstation with an Intel Core i5-4690K @ 3.50GHz, 8 GB RAM memory and an NVIDIA GTX 970 was taken as the first test platform. We used the latest CUDA 7.5 platform. A second test device was a server with two Intel Xeon E5-2640 CPUs at 2.50GHz, 64 GB RAM and two NVIDIA Tesla K20m (compute capability 3.5).

3.2 Parallel Heat Distribution Application – Experiments

Jason Sanders and Edward Kandrot in their book 'CUDA by example' [24] used simulation of heat transfer to demonstrate how a texture memory can be used. In fact, the physical model was significantly simplified. In that work the most important aspect was to show how texture memory works. In this work we ran simulation based on a similar, simplified model and focused our work on testing UM and DP mechanisms.

Our benchmark is a simulation based on thermal conduction [10]. It takes place in a two-dimensional space which is divided into a number of small, square cells. At selected areas sources of heat with constant temperatures were placed. During simulation, cells become warm due to heat conduction from neighbours. To model heat loss through a wall, we can write a formula for the rate of conduction heat transfer as follows:

$$\frac{Q}{t} = \frac{\lambda A}{\delta} \Delta T \tag{1}$$

where: Q - heat transferred in time t, λ - thermal conductivity, A - area, δ - thickness of the wall, ΔT - temperature difference on both sides of the wall.

Hence, between simulation time frames we update the temperature in each cell according to the formula:

$$T_N = T_O + \sum_{neighbours} k \cdot (T_{neightbour} - T_O) \tag{2}$$

where k is a constant which controls the speed of heat transfer. It can be identified with $\frac{\lambda A}{\delta}$ from above.

A simulation loop is split into two parts. At the beginning, the temperature in every cell is updated. The next step involves renewing temperatures in heat sources. Finally, we compute proper colors for display and draw these on the screen. We perform drawing once per a predefined number of iterations/frames.

3.2.1 Dynamic Parallelism

As mentioned above, before drawing the simulation grid, the application performs a fixed number of loops, e.g. 90. In the non-DP version kernels responsible for computing iterations were called from the CPU side, which potentially involved considerable time spent on communication and synchronization. After each iteration temperatures for heat sources need to be updated. In a DP version, the whole loop was moved to the GPU and all kernel launches were executed from the GPU. In the DP version, CPU's only task was to draw the map after GPU computations. It means, that we needed to perform only one memory transfer, from GPU to CPU with results after a number of iterations.

The described implementation was tested with various sizes of the simulation space. During testing for one drawing 90 simulation loops were performed. We measured the time taken to generate a single heat map or the time of the 90 loops. Compared to the standard version we observed a slight performance gain – 2-3%.

In order to confirm a hypothesis, that moving the simulation loop to the GPU results in performance gains, we performed an additional test. With the same implementation we measured time taken to generate a single heat map for various numbers of simulation iterations. The greater the number of simulation iterations, the more time the application spends on the GPU side. We observed that with an increasing number of simulation iterations the ratio of the time for the DP version to the time for the non-DP version was dropping. Results are presented in Figure 1. Again the performance gain fluctuated around 2-3% for up to 100 simulation iterations.

Heat sources temperature update

To improve performance gains we took a look at simulation steps separately. As described above, the first part included an update of heat source temperatures.

The initial version of that kernel worked for the whole simulation grid. For every cell a thread would check whether it was a heater cell. Such cell temperatures were renewed. It can be easily seen that it causes work redundancy,

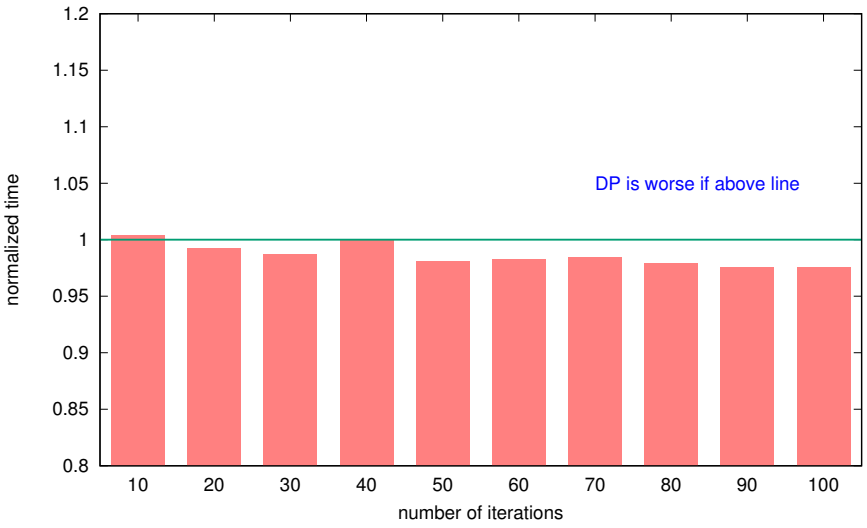

Fig. 1: Time needed to generate simulation frame depending on the number of iterations. Time measured for Dynamic Parallelism version is normalized to standard API version.

especially if the heater's area is relatively small compared to the whole grid. A possible solution could be to call the kernel only for the areas occupied by heaters, e.g. in a `for` loop. Another approach can be a single kernel that could use DP to call child kernels each for every heat source.

In order to benchmark various possibilities, we implemented the following 5 versions:

- `std-cpConst` – a kernel works on the whole simulation grid. For every cell that is a part of a heater renew its temperature.
- `std-for`– an application uses the standard CUDA API and invokes a kernel in a `for` loop from a CPU. Each kernel invocation renews temperature only for one heater.
- `dp-cpConst` – a kernel uses a similar approach as for `std-cpConst`. The difference is that we shifted the kernel invocation to the device side.
- `dp-for` – on the device side in a `for` loop we call the kernel for each heater. It is similar to `std-for` but the loop is on device side.
- `dp-cpHeaters` – we call a kernel that calls nested kernels for each heater on the device side.

These kernels were tested for various numbers of heat sources. The simulation grid size was 1024x1024px and we performed 90 loops for every drawing. Each heater takes a 50x50px area. The percentage of area taken by heaters is shown in Table 1.

Measured times for each version are presented in Figure 2. It turned out that for fewer than 20 heaters better approaches were those that copy indi-

Table 1: The percentage of area taken by heaters

number	2	3	4	5	6	7	8	9	10	20	30
area [%]	0.48	0.72	0.95	1.19	1.43	1.67	1.91	2.15	2.38	4.77	7.15

vidual heaters in loops. Depending on the number `dp-for` version (up to 10) or `dp-cpHeaters` was best (30 heaters). However, if the number of heaters exceeded 20, better versions were those working on the whole simulation area. Both solutions (`std-cpConst` and `dp-cpConst`) exhibited similar results. The most universal approach was the `std-for` version. It worked well both with small and bigger number of heat sources. In the following experiments we use the `dp-for` approach.

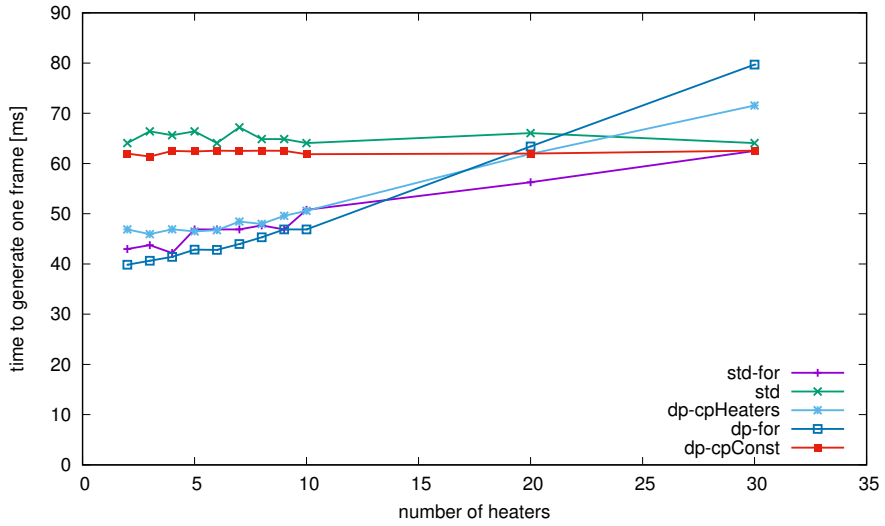

Fig. 2: Time to generate one frame based on the number of heaters

We also compared the chosen kernel version `dp-for` to the `dp-cpConst` in a separate chart to better show performance gains. Detailed results are presented in Figure 3. It can be seen that for 2 heaters the `dp-for` approach is almost 40% faster.

Calculation of temperature

The second step of simulation is computing a new temperature for each cell. In the initial version we computed it for all cells. With DP we can naturally change the code to update it only in cells where it is really necessary. In the simulation grid we can find places where, especially at the beginning of

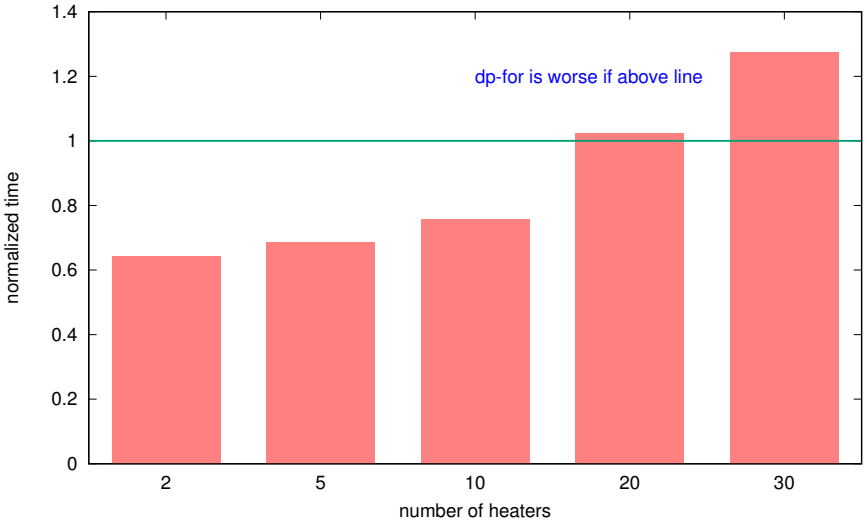

Fig. 3: Comparison of `dp-cpConst` and `dp-for` kernels. Time is normalized to measurements of `dp-cpConst` kernel

simulation, there is no activity. Heat will reach those parts only after several iterations.

We divided the grid into smaller, square parts – tiles. In each tile we detect whether temperature changes are big enough to update the tile. Detection is performed by adding temperatures of cells in the tile. If the sum exceeds a threshold we mark that tile and in the future iterations such detection will not be performed again. For marked tiles we compute temperatures as in standard version. It should be noted that this approach allows to speed up computations at the cost of slightly decreased simulation accuracy which is not visible in visualization though.

Additionally we distinguish two versions of this approach. In the first implementation we perform detection in every loop (marked as `dp`). The second version checks each tile one per visualization, e.g. once per 90 iterations (marked as `dp-heur`).

Obtained results, presented in Figure 4, show that the `dp-heur` version has improved performance 2 times. With time, when the simulation overtakes a greater area of the grid performance becomes similar to the standard version.

3.2.2 Unified Memory

In the tested application memory management is fairly standard. After initialization of the simulation only transfers between host and device occur before drawing. We copy only a bitmap ready for visualisation. Using UM required only replacing a memory allocation function and removing explicit data transfers.

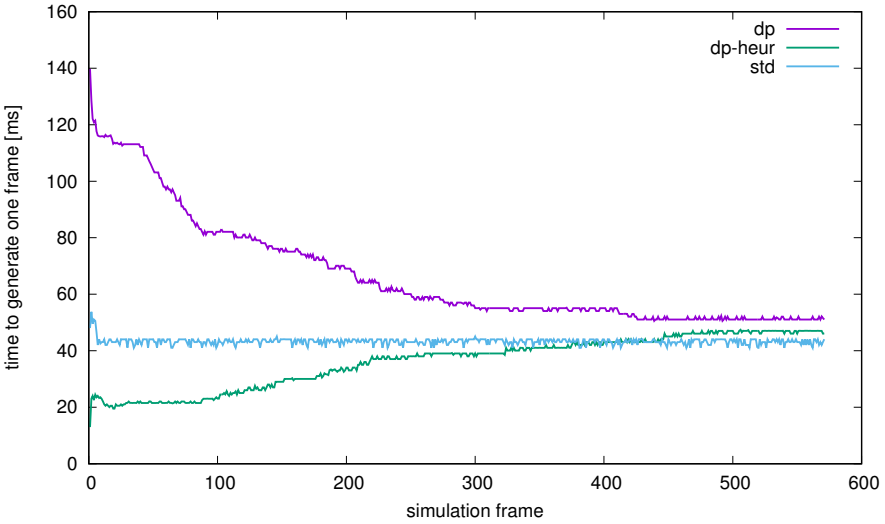

Fig. 4: Time needed to generate one frame depending on the duration of the simulation

We measured times for different sizes of the simulation grid. We observed minor differences in execution times (1-2%) which is negligible (Figure 5). More interesting was a second test. In that case we measured times for different numbers of iterations performed between drawings. The smaller that value, the more frequently a memory transfer is performed. Results for that test are presented in Figure 6. It turned out that frequent transfers resulted in larger (percentage wise) execution times for the UM version (almost 10% for 10 iterations).

3.2.3 Summary

Heat transfer simulation created an opportunity to gain an advantage from utilizing the Dynamic Parallelism (DP) mechanism. We obtained an interesting performance gain even for a simplified physical model.

On the other hand Unified Memory caused a visible performance drop. For this application, in which case memory management is fairly simple, UM does not bring benefits.

3.3 Parallel Adaptive Numerical Integration – Experiments

For the purpose of testing DP and UM we decided to implement an adaptive numerical integration algorithm. An original range is divided into a number of small subranges. Within each subrange the area is calculated as an area a_1 of a trapezoid which is compared to the sum a_2 of areas of two smaller trapezoids.

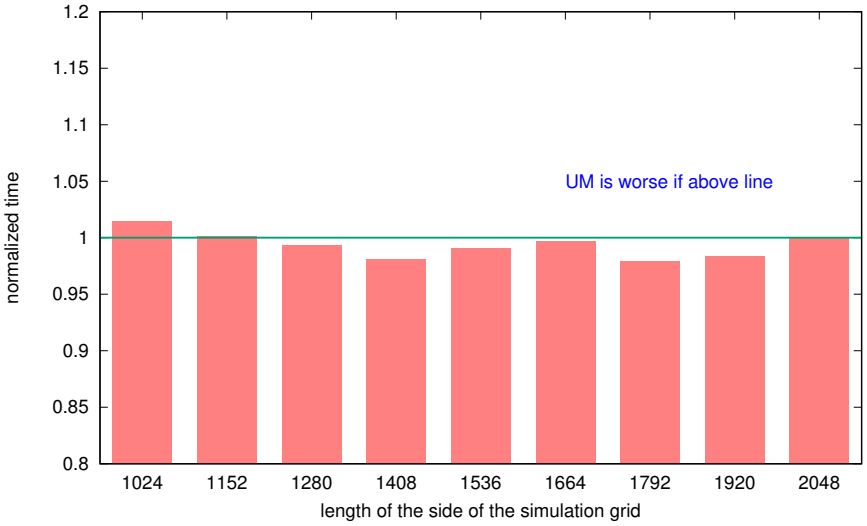

Fig. 5: Time needed to generate one frame for Unified Memory version compared to standard API version depending on grid size

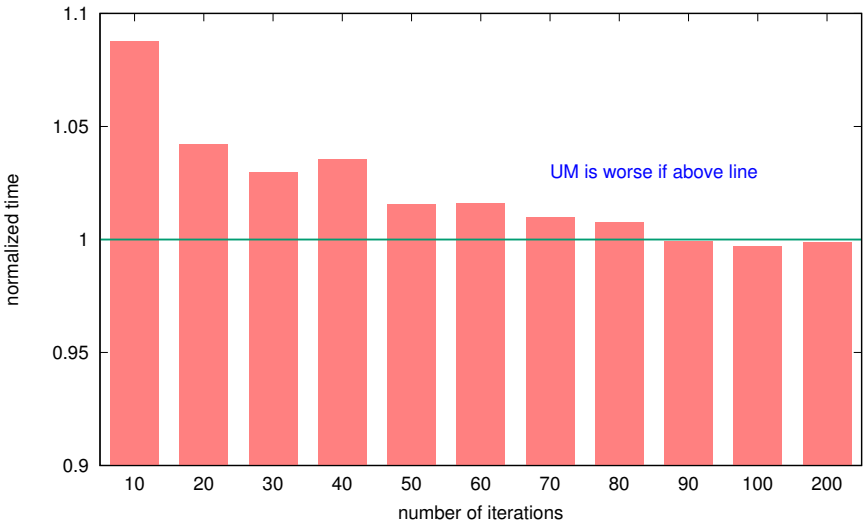

Fig. 6: Time needed to generate one frame for Unified Memory version compared to standard API version depending on number of iterations

Those two are created by splitting the subrange into two subranges of equal width. If $|a_1 - a_2|$ is greater than a given ϵ, then the procedure is performed again, for every half of the subrange.

Parallelization of the described algorithm is intuitive. Every CUDA thread is responsible for one subrange. After computations have been completed it writes a result into a vector prepared earlier. The last step of the algorithm is vector reduce in order to obtain the final integration result. It is performed by a well known and optimized reduction algorithm.

The aforementioned approach was implemented in three ways:

1. The first version, which will further be called `static`, splits the initial range into subranges each of width δ. Then, in every subrange a trapezoid area is calculated. There are no nested procedure calls nor testing accuracy of the calculations for a subrange. It is intuitively clear that in some cases (e.g. for a linear function) this method may result in significant redundancy of calculations compared to what could be done with a knowledge of the function.
2. The second implementation uses an adaptive approach. However, it is an approach still without recurrent calling for ranges that need more accuracy of calculations. For a subrange there is indeed checking of accuracy but calculations for smaller subranges are performed in an iterative way. This approach does not need Dynamic Parallelism (DP) to work.
3. The last algorithm finally utilizes the DP mechanism. Similarly to the previous approach there is accuracy checking but now we use recursive calls using DP to achieve the desired accuracy.

The main goal was the most fair comparison of the presented approaches. For this reason the algorithm in every version should be as similar as possible to other versions. In the `static` version the most important is δ parameter. It should be as small as the smallest subrange in an adaptive version.

Another important consideration is the size of a vector prepared for results of subranges. For big ranges it can be necessary to split the input range into smaller parts, because a vector would be too big for device memory. Because accuracy of a result is important the `double` type is used which requires 8 bytes for every subrange. Therefore, for large ranges additional transfers between the device and the host may occur.

3.3.1 Test functions

During experiments the following functions were used:

$$f(x) = sin(x)cos(x) \tag{3}$$

$$f(x) = xsin(x)cos(x) \tag{4}$$

$$f(x) = \begin{cases} xsin(x)cos(x) & x \leq \dfrac{b-a}{2} \\ 2x & x > \dfrac{b-a}{2} \end{cases} \tag{5}$$

To test application behaviour in different cases we selected various functions. Adaptive algorithms should perform fewer computations than a static version which will perform redundant calculations in this case. The last test function is an artificial benchmark which in one half of the range is a linear function and in the second half fluctuates greatly.

Next sections contain results of experiments. For adaptive versions an additional comment is required. It is well known that on a device running threads are grouped into warps (each warp contains 32 threads). With this in mind, when we split a subrange and call children kernels, it might be more efficient to split not into 2 but into 32 parts. We marked implementations with proper numbers to also check that hypothesis. To be fair, in the iterative version we also distinguish two versions of splitting. Each approach are marked as follows:

- `static` – a static version of the application,
- `adaptive32` – an adaptive version, without Dynamic Parallelism with splitting into 32 parts,
- `adaptive2` – an adaptive version, without Dynamic Parallelism with splitting into 2 parts,
- `dp32` – an adaptive version, using Dynamic Parallelism with spliting into 32 parts,
- `dp2` – an adaptive version, using Dynamic Parallelism with splitting into 2 parts.

3.3.2 Dynamic Parallelism

We ran tests with different lengths of the initial range for tested functions. Parameter δ was set to the width of the smallest subrange in an adaptive version. Obviously, the value of that parameter was set separately for every function. Additionally, we assumed that the result vector will contain 10^8 numbers. It means that every kernel call can integrate that number of subranges. If the range is split into more parts, there is a need for additional kernel call(s).

For sinusoidal functions the most interesting results were obtained for function 4 and are presented in Figure 7. That function fluctuates more than function 3. It means that recursive calls are more nested and a subrange is shorter. Integration results were more accurate for the static version (difference at the second digit after the dot). It is probably caused by rounding errors during adding areas for nested parts. Version `adaptive32` was significantly worse compared to other versions. The `static` version was slower than Dynamic Parallelism versions but at the same time it had a slightly better accuracy as mentioned above. However, `adaptive2` was the fastest version.

The biggest advantage of the adaptive algorithm is the ability to control subrange width depending on the accuracy. In practice, the static approach would not be very useful, because it is difficult to determine a correct subrange width to obtain a good result and reasonable execution time.

In order to compare both static and DP solutions fairly we created an artificial function 5, mentioned earlier. Measured execution times can be seen

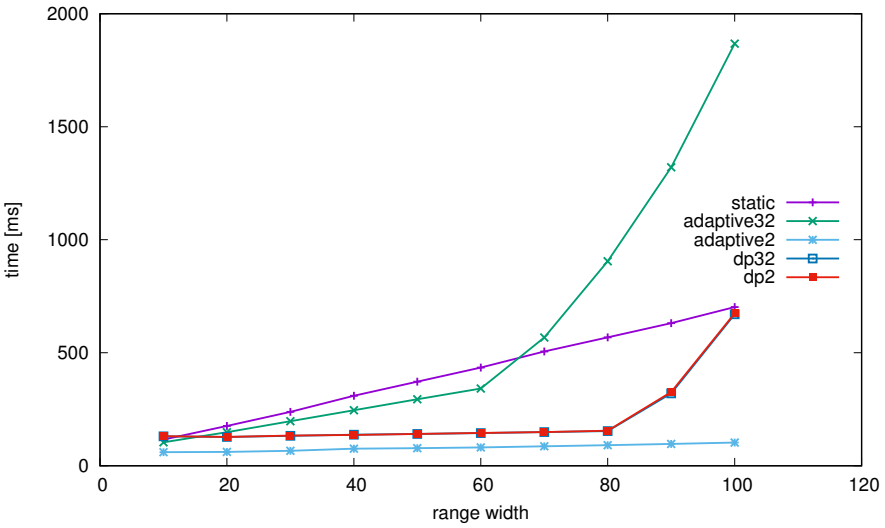

Fig. 7: Integration time depending on range width - function 4

in Figure 8. As before, we set the subrange width to the smallest width in adaptive version. The difference arises due to integrating the linear fragment with those small subranges. Other implementations were significantly faster. Again, the best results were achieved by `adaptive2` version.

It turned out that in this case the iterative implementation is slightly faster and at the same time more flexible. In almost every case that version behaved best. What is more, the application code was much more readable compared to a DP version. However, the implementation with recursion is still efficient and more flexible than the static approach.

3.3.3 Unified Memory

Memory management is different for particular versions. As mentioned before, we integrate a fixed number of subranges and write results into a previously prepared vector. Then the vector elements are reduced into a final result. For static and adaptive (iterative) approaches we transfer only that sum to the host. When the range is split into several parts we will sum those partial sums. The Dynamic Parallelism algorithm will only transfer one number with a final result. Partial sums are obtained on the device side.

Utilization of Unified Memory will affect efficiency of this part of the application. We measured execution times for different numbers of transfers. As a test function we used function 3 which was integrated from 0 to 100. We tested each implementation, i.e. we measured time with and without Unified Memory utilization. Unified Memory resulted in worse execution times for each tested version. An example is shown in Figure 9.

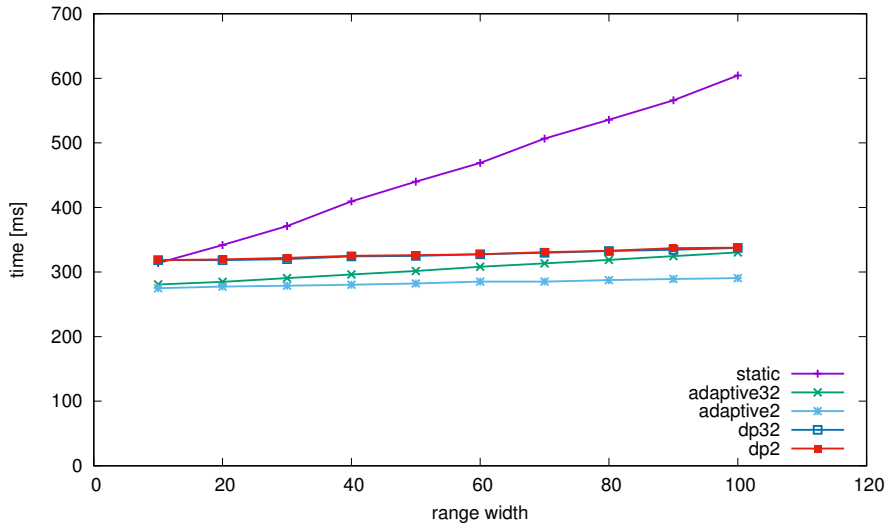

Fig. 8: Integration time depending on range width - function 5

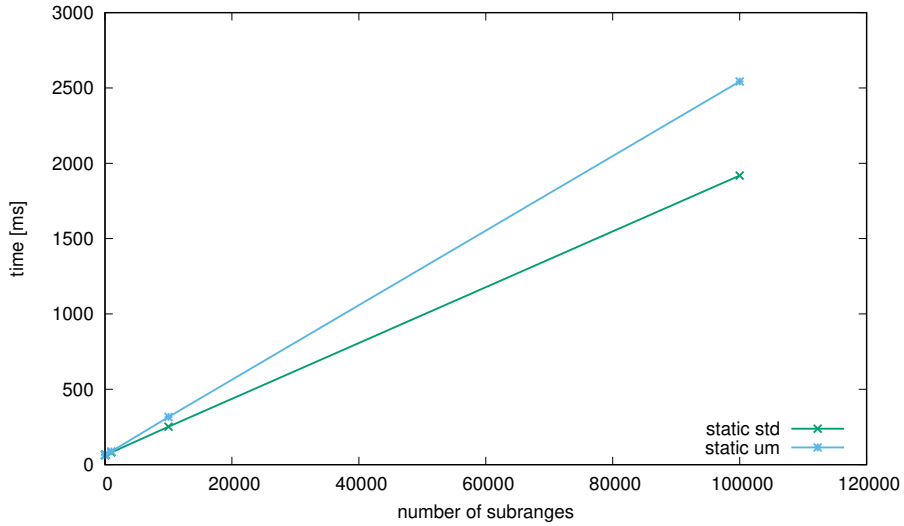

Fig. 9: Integration time depending on number of memory transfers - function 3

The last experiment was focused on comparison of two adaptive approaches. In this case, however, algorithms are compared in the context of time spent on memory transfers. As we wrote before Dynamic Parallelism will only transfer the final result of integration. In contrast, the iterative version must transfer partial sums which are added on the host side. It generates additional traffic between the host and the device. We compared algorithms for various numbers

of memory transfers. Results for each tested device can be found in Figures 10 and 11.

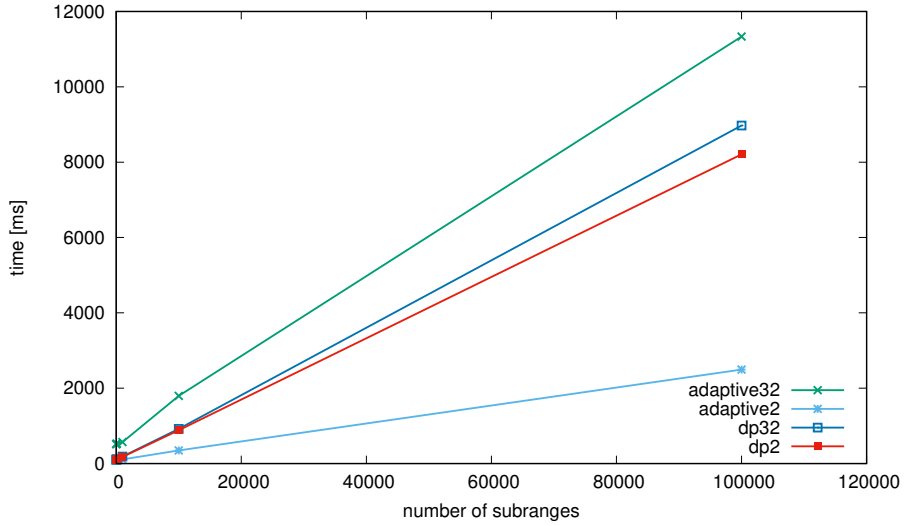

Fig. 10: Comparison of execution time for adaptive algorithms depending on number of memory transfers on GTX 970 - function 3

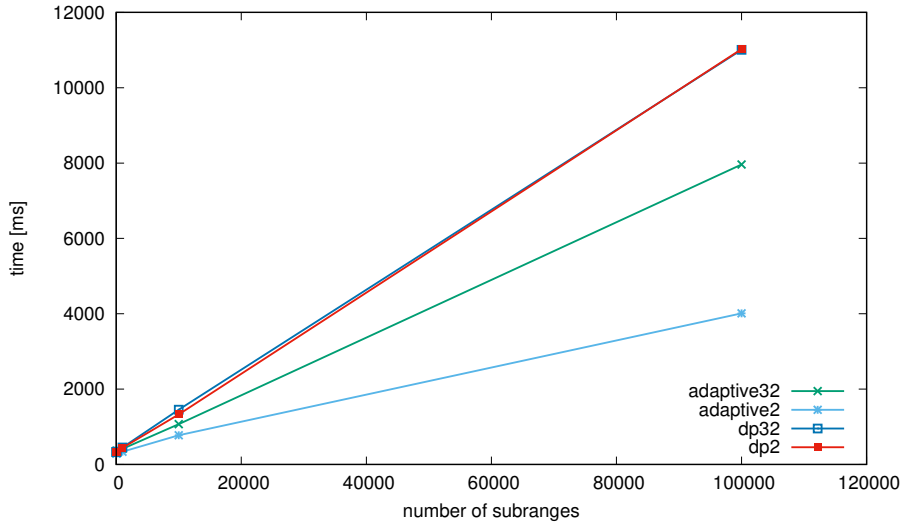

Fig. 11: Comparison of execution time for adaptive algorithms depending on number of memory transfers on Tesla K20m - function 3

For versions with a split into 2 parts the iterative approach was faster. The reason is that the split into 2 parts causes deeper nesting of recursion. In that case the overhead for an additional kernel call is more expensive than iterative execution. What is more, execution of only 2 threads is not optimal in terms of warp utilization.

Results for 32 versions are somewhat more interesting. On GTX 970 the approach utilizing Dynamic Parallelism was better. In turn on Tesla K20m the iterative version was faster. Additionally, we measured time of a kernel call. It turned out, that it is almost 4 times longer on the Tesla device. As a result the time which is saved on avoiding memory transfer is covered by kernels call times.

Table 2: Kernel call time on tested devices [ms]

	Time [ms]
Tesla K20m	30.95
GTX 970	8.39

3.3.4 Summary

The first obvious conclusion is that Unified Memory does not improve application efficiency. In the case of the testbed implementations it did not improve code quality nor readability.

More interesting were results obtained for versions with Dynamic Parallelism. For the integration problem the mechanism worked well, especially for functions with fluctuating trends. However, we managed to implement a slightly more efficient iterative version of the adaptive algorithm.

3.4 Parallel Goldbach Conjecture Application – Experiments

The last application is connected with one of the most famous and oldest mathematical problems. More than 270 years ago Leonhard Euler and Christian Goldbach formulated a hypothesis which is now known as Goldbach's Conjecture.

Conjecture 1 *Every even number greater than 2 can be expressed as a sum of two primes.*

Today there is still no formal proof that the hypothesis is true. Modern mathematicians believe that it is true, the more that by using supercomputers proved the hypothesis for numbers smaller than $4 \cdot 10^{17}$ [3,9].

A simple approach could be to check if for given interval the hypothesis is true. For each number a single thread looks for a sum of two primes. We

modified this idea. Instead of working on the whole range, we prepare numbers to check as an input for the program. During primality test we do not repeat tests on same numbers (which would be the case if we worked on numbers from a sequence). We prepared inputs for each test with randomly generated numbers with a given order of magnitude.

The key element of the program is the primality test. In the application a straightforward approach is used that checks divisibility of a number by another dividers. In order to improve this process we prepared an array of 1000 boolean values. For numbers smaller than 1000 the program easily checks primality by testing a value from a proper index.

3.4.1 Dynamic Parallelism

The approach using Dynamic Parallelism works slightly differently from the base reference version. Each thread utilizes child threads to find a sum of proper prime numbers. In order to stop the algorithm when that pair is found, the parent calls a nested kernel in a loop, for packets of numbers. If for a given packet prime numbers are found the thread ends its work.

We tested the described approach to find an optimal number of child threads. We ran tests for numbers in the order of 10^{10}. Among pools of 32, 64, 128 and 256 threads the best execution time was obtained for a version with the 128 thread pool. Results were compared to the standard version of the algorithm and are presented in Figure 12. It turned out that use of the Dynamic Parallelism caused a significant performance loss.

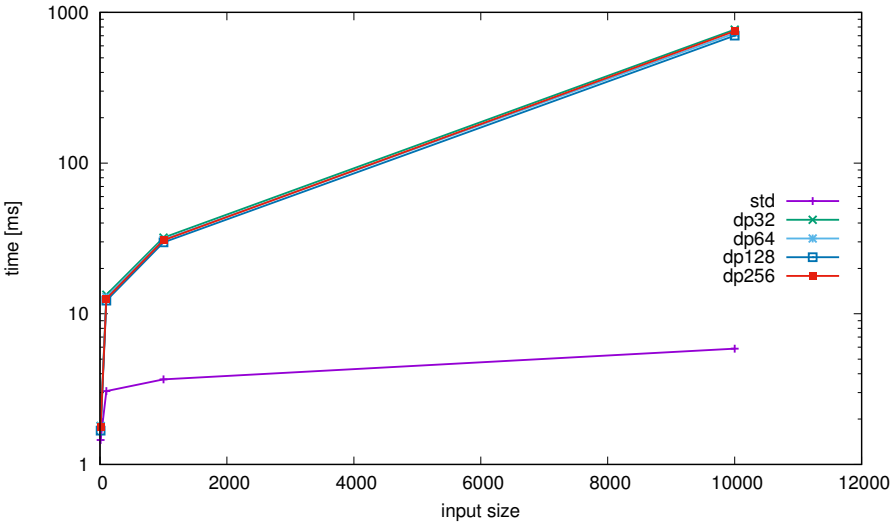

Fig. 12: Comparison of execution time Dynamic Parallelism version to standard API version

Such a result raises the question what is the cause for that efficiency loss. The first thing we should consider is the way CUDA parallelism works. CUDA threads are 'light' and work best for simple tasks, when they execute same instructions on different data. Such parallelism was discussed in the case of both previous applications. The described problem forces more work on each single threads. Often threads go through different paths of program execution. It means that some threads end computations before others (eg. during the primality test) which can be particularly expensive in terms of warp execution. As a result it can result in performance loss.

Goldbach's Conjecture problem could possibly be parallelized better using a different framework and hardware. In this case, it could be better to use threads working on cores of a standard CPU or possibly Intel Xeon Phi [5]. For example on a computing cluster each node could work on its numbers package. Such an environment works well in case of different execution paths of threads. As can be seen, a problem does not always fit the CUDA platform well. Similarly, Dynamic Parallelism does not always result in a performance gain.

3.4.2 Unified Memory

The described algorithm has two sets of data which are transferred between the host and the device. The first is the array with boolean values. Another data set is taken from the input and is sent to the device to verify the conjecture. We implemented Unified Memory in both previous versions, the standard and the one with Dynamic Parallelism.

Measured execution times are very similar for both versions (the standard and with Unified Memory) as shown in Figure 13.

As mentioned before, an important part of the algorithm is the boolean vector. It is calculated and copied to the device memory. We decided to compare 4 approaches in terms of access to that vector. We based on the standard version of the application (without Dynamic Parallelism):

- `std-constant` - version with standard memory management, the vector is stored in constant memory,
- `std-global` - version with standard memory management, the vector is stored in global memory,
- `um-constant` - version with Unified Memory, the vector is stored in constant memory,
- `um-global` - version with Unified Memory, the vector is stored in global memory (marked as `__managed__`).

We ran tests with 10000 numbers as an input but we changed their range from 10^8 to 10^{12}. The collected results are shown in Figure 14. Again, differences between versions were negligible. It can be concluded that in this case the memory management method is not important. The programmer can choose the most handy method.

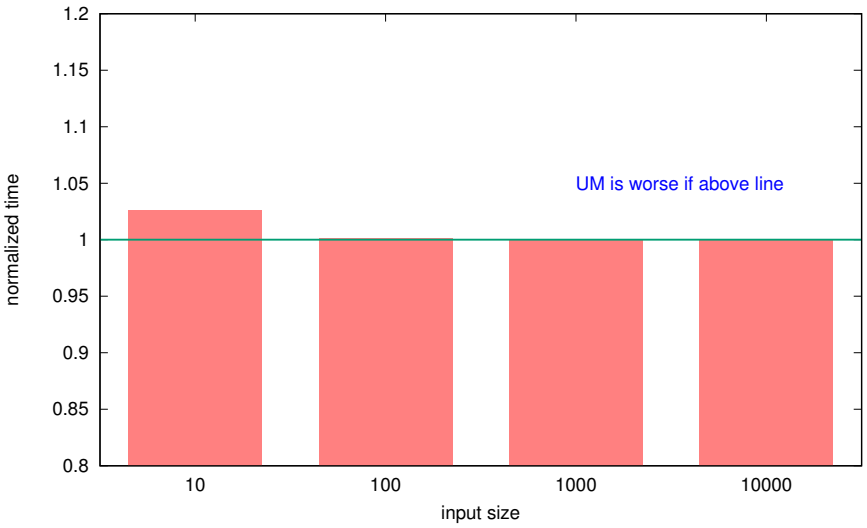

Fig. 13: Unified Memory version execution time normalized to standard API version time

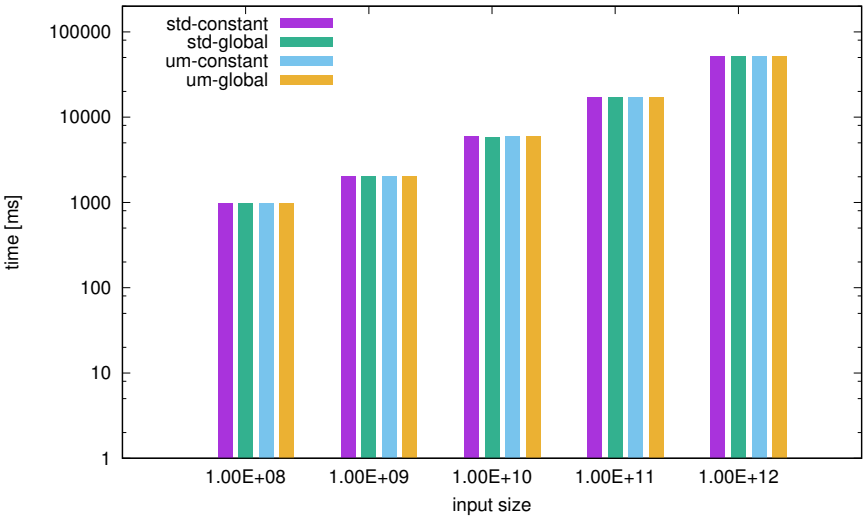

Fig. 14: Execution time for different memory management versions depending on input numbers range

3.4.3 Summary

The tested application is an example of a problem which cannot be easily and efficiently parallelized using the CUDA platform. Especially Dynamic Paral-

lelism caused performance loss compared to the standard API version. On the other hand, Unified Memory tests showed that sometimes it does not affect application performance in a negative way. Despite the fact that the application does not work faster, a programmer can choose it as a more convenient solution, which would slightly improve code readability.

4 Conclusions and Future Work

Within this paper, Dynamic Parallelism and Unified Memory mechanisms available in the CUDA API and platform were compared for three different applications, representative of: geometric SPMD processing – applications such as heat transfer simulation in 2D space and verfication of Goldbach's conjecture, as well as divide and conquer processing – adaptive numerical integration of a function over a given range. Detailed analysis was presented for each application with several optimization applied and compared with and without DP and UM. Results can serve as guidelines on whether to use DP and/or UM for other applications of similar type.

For DP, depending on an application, benefits varied. For heat distribution and an approach in which no redundant computations are performed in some areas of the domain, a performance gain was obtained. For adaptive integration a DP enabled algorithm was similar but slightly worse in performance than a version using the standard API. Finally, for verfication of Goldbach's conjecture, a DP enabled version could not match the version with the standard API because the type of the problem is not well suited for codes with thread divergence. In all of these cases, incorporation of DP into the code was not trivial and increased code complexity. In some cases, an approach had to be changed in order to apply DP to the problem.

Application of UM resulted, in case of all applications, in either performance drop or times very close to the times of versions using the standard API. In case of selected tests for heat distribution, we obtained slightly better results with UM, but in the order of 1-3%.

The aforementioned results allow us to draw the following conclusions. DP can bring considerable benefits for recursive algorithms or algorithms that use hierarchically arranged data. In such cases, code also becomes much more readable. It seems that it should be used when it applies naturally and not in cases where it is not directly applicable. It should also be taken into account that generally recursion involves an overhead compared to an iterative solution. UM is a mechanism that allows to enter the CUDA programming world fast as a program with UM resembles very much standard programs written for a CPU.

References

1. Adinetz, A.: Adaptive parallel computation with cuda dynamic parallelism. https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/

(2014). [Accessed 17.02.2016]
2. Aliaga, J.I., Davidovic, D., Pérez, J., Quintana-Ortí, E.S.: Harnessing CUDA dynamic parallelism for the solution of sparse linear systems. In: G.R. Joubert, H. Leather, M. Parsons, F.J. Peters, M. Sawyer (eds.) Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK, *Advances in Parallel Computing*, vol. 27, pp. 217–226. IOS Press (2015). DOI 10.3233/978-1-61499-621-7-217. URL http://dx.doi.org/10.3233/978-1-61499-621-7-217
3. Caldwell, C.: Goldbach's conjecture. http://primes.utm.edu/glossary/page.php?sort=GoldbachConjecture. [Accessed 10.06.2016]
4. Czarnul, P.: Programming, tuning and automatic parallelization of irregular divide-and-conquer applications in DAMPVM/DAC. IJHPCA **17**(1), 77–93 (2003). DOI 10.1177/1094342003017001007. URL http://dx.doi.org/10.1177/1094342003017001007
5. Czarnul, P.: Benchmarking performance of a hybrid intel xeon/xeon phi system for parallel computation of similarity measures between large vectors. International Journal of Parallel Programming pp. 1–17 (2016). DOI 10.1007/s10766-016-0455-0. URL http://dx.doi.org/10.1007/s10766-016-0455-0
6. Czarnul, P.: Parallelization of Divide-and-Conquer Applications on Intel Xeon Phi with an OpenMP Based Framework, pp. 99–111. Springer International Publishing, Cham (2016). DOI 10.1007/978-3-319-28564-1_9. URL http://dx.doi.org/10.1007/978-3-319-28564-1_9
7. Czarnul, P., Grzeda, K.: Parallel simulations of electrophysiological phenomena in myocardium on large 32 and 64-bit linux clusters. In: D. Kranzlmüller, P. Kacsuk, J. Dongarra (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings, *Lecture Notes in Computer Science*, vol. 3241, pp. 234–241. Springer (2004). DOI 10.1007/978-3-540-30218-6_35. URL http://dx.doi.org/10.1007/978-3-540-30218-6_35
8. DiMarco, J., Taufer, M.: Performance impact of dynamic parallelism on different clustering algorithms. In: SPIE Defense, Security, and Sensing, pp. 87,520E–87,520E. International Society for Optics and Photonics (2013)
9. Guy, R.: Unsolved problems in number theory. Springer Science & Business Media (2013)
10. Halliday, D., Resnick, R., Walker, J.: Fundamentals of Physics Extended, 10th Edition. Wiley (2013)
11. Jones, S.: How tesla k20 speeds quicksort, a familiar comp-sci code. https://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/ (2012). [Accessed 11.06.2016]
12. Joseph, J., Keville, K.: An evaluation of cuda unified memory access on nvidia tegra k1. Waltham, MA USA (2015). IEEE High Performance Extreme Computing Conference (HPEC '15) Nineteenth Annual HPEC Conference
13. Landaverde, R., Zhang, T., Coskun, A.K., Herbordt, M.: An investigation of unified memory access performance in cuda. In: High Performance Extreme Computing Conference (HPEC), 2014 IEEE, pp. 1–6 (2014)
14. Li, D., Wu, H., Becchi, M.: Exploiting dynamic parallelism to efficiently support irregular nested loops on gpus. In: Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores, COSMIC '15, pp. 5:1–5:1. ACM, New York, NY, USA (2015). DOI 10.1145/2723772.2723780. URL http://doi.acm.org/10.1145/2723772.2723780
15. Li, W., Jin, G., Cui, X., See, S.: An evaluation of unified memory technology on nvidia gpus. In: Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, pp. 1092–1098 (2015). DOI 10.1109/CCGrid.2015.105
16. Mehta, V.: Exploiting cuda dynamic parallelism for low power arm based prototypes (2015). GPU Technology Conference, San Jose, U.S.A., http://on-demand.gputechconf.com/gtc/2015/presentation/S5384-Vishal-Mehta.pdf
17. Mei, G.: Evaluating the power of gpu acceleration for idw interpolation algorithm. The Scientific World Journal **2014** (2014). Article ID 171574, doi:10.1155/2014/171574

18. 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)
19. NVIDIA Corporation: Dynamic parallelism in cuda (2012). Http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf
20. NVIDIA Corporation: NVIDIA CUDA C programming guide (2015). Version 7.5
21. Plauth, M., Feinbube, F., Schlegel, F., Polze, A.: Using dynamic parallelism for fine-grained, irregular workloads: A case study of the n-queens problem. In: 2015 Third International Symposium on Computing and Networking (CANDAR), pp. 404–407 (2015). DOI 10.1109/CANDAR.2015.26
22. Plauth, M., Feinbube, F., Schlegel, F., Polze, A.: A performance evaluation of dynamic parallelism for fine-grained, irregular workloads. International Journal of Networking and Computing **6**(2), 212–229 (2016). URL http://www.ijnc.org/index.php/ijnc/article/view/126
23. Sakharnykh, N.: Combine openacc and unified memory for productivity and performance (2015). Https://devblogs.nvidia.com/parallelforall/combine-openacc-unified-memory-productivity-performance/
24. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming, 1st edn. Addison-Wesley Professional (2010)
25. Souto, R.P., Osthoff, C., de Vasconcelos, A.T., Augusto, D.A., da Silva Dias, P.L., Rodriguez, A., Trelles, O., Ujaldon, M.: Applying gpu dynamic parallelism to high-performance normalization of gene expressions (2014). GPU Technology Conference, San Jose, U.S.A., http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4209_biofinformatics_sort_dynamic_parallelism.pdf
26. Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints **abs/1605.02688** (2016). URL http://arxiv.org/abs/1605.02688
27. Wang, J., Yalamanchili, S.: Characterization and analysis of dynamic parallelism in unstructured gpu applications. In: Workload Characterization (IISWC), 2014 IEEE International Symposium on, pp. 51–60 (2014). DOI 10.1109/IISWC.2014.6983039
28. Wilkinson, B., Allen, M.: Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, edition edn. Pearson (2004). ISBN 978-0131405639
29. Zhang, P., Holk, E., Matty, J., Misurda, S., Zalewski, M., Chu, J., McMillan, S., Lumsdaine, A.: Dynamic parallelism for simple and efficient gpu graph algorithms. In: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, IA3 '15, pp. 11:1–11:4. ACM, New York, NY, USA (2015). DOI 10.1145/2833179.2833189. URL http://doi.acm.org/10.1145/2833179.2833189