J Supercomput (2017) 73:5378–5401
DOI 10.1007/s11227-017-2091-x

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

Łukasz Jarząbek[1] · **Paweł Czarnul**[1]

Published online: 15 June 2017

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, tested applications 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 be optimized further. Subsequently, we compared the best dynamic parallelism and unified memory versions to respective standard API counterparts. It was shown that usage of dynamic parallelism resulted in improvement in 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. In most cases, unified memory 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

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

Łukasz Jarząbek
lukjar92@gmail.com

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

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 [27]. 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) [21]. Typically, initial 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. UM, 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 [21]. 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. DP, 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 [21]. 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 [20], numerical applications such as adaptive integration [6] and others. It should also be noted that OpenCL, as another popular API for programming GPUs (and also CPUs + GPUs), offers device side enqueuing (a parent kernel enqueues a child kernel) and shared virtual memory where a virtual address space is accessible both from the host and a device with three types: coarse-grained buffer SVM, fine-grained buffer SVM and fine-grained system SVM depending on granularity of synchronization [13].

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 Sect. 2, this paper analyzes three parallel applications, falling into either geometric SPMD [7], general SPMD [29] and divide-and-conquer [4,29] 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.

It is also worth to mention how tested mechanisms improve or diminish code readability and what effort should be taken to use DP or UM in particular applications. However, the flexibility of these tools is quite subjective. In contrast to performance, it is difficult to judge ease of programming in such technologies. We carried out a survey among a group of programmers who have basic knowledge regarding the CUDA platform. We asked them about readability of code in both version of each application (with and without DP/UM). The conclusions of the survey are in line with our perceptions, outlined in the summary of this work. However, we mainly focus our efforts on evaluation of performance impact of these mechanisms and we treated flexibility as a side aspect of this work.

Based on obtained results, we have formulated the following conclusions. DP can bring considerable benefits in performance terms for algorithms that use hierarchically arranged data and for which DP can be naturally applied. UM is a mechanism that makes it easier to enter the CUDA programming world. It does not provide significant performance improvements and often slows down an application.

The outline of the paper is as follows. Section 2 includes related work regarding both UM and DP 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 the best settings and comments on how these affect readability of the code. Parallel applications include: heat distribution in Sect. 3.2, adaptive numerical integration in Sect. 3.3 and Goldbach conjecture verification in Sect. 3.4. Finally, Sect. 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 UM 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 [14] thoroughly 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 are accessed. The authors state that in such a case the UM mechanism can place data favorably 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 [19], 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 four 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 [24] 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 speedup 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 eight threads. Paper [16] 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 speedup 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 the 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 the second approach, based on a Mariani–Silver algorithm, space is divided adaptively within threads using DP. As a result speedup with this solution was between 1.3 and 6 times depending on space size with higher speedups 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 [22,23] 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 performing tests for this application using NVIDIA K20×, 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 [30] 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 paper [15] state that they obtained over 2.6 speedups 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 [18], 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 [28], 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 speedups of DP-enabled codes by first calculating ideal execution times by subtracting kernel launch overheads. The conclusion is that there is a speedup 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 [17] focuses on CUDA DP for low-power ARM-based prototypes demonstrating 20% savings in energy consumption on a Pedraforca prototype. Work [26] 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 speedup of around 4.8. It is noted that the speedup 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 out of three applications with and without the DP and UM mechanisms. Then, we compared standard versions using the standard API with the other two applications. Table 1 summarizes the applications and how DP and UM were deployed in the code as well as what additional improvements have been considered. The next chapter contains short descriptions of each program and performed experiments. We ran each test 20 times for application 1 (because of short execution times) and 10 times each for applications 2 and 3. All presented results are averaged. Each section, beyond test results, contains respective conclusions and discussion.

3.1.1 Test platforms

A workstation with an Intel Core i5-4690K @3.50 GHz, 8 GB RAM memory and an NVIDIA GTX 970 was taken as the first test platform. We used the CUDA 7.5 platform. A second test device was a server with two Intel Xeon E5-2640 CPUs at 2.50 GHz, 64 GB RAM and two NVIDIA Tesla K20m (compute capability 3.5). We tested each application on both platforms except the first one, where additionally we performed visualization that was only possible on the first machine. Each experiment utilized only one device, we did not test dual GPUs configuration. Because in most cases tendencies in results were similar in both environments, in order to avoid redundancy we present only selected graphs (it is specified which device was used).

Table 1 Applications tested

	Application 1	Application 2	Application 3
Purpose	Heat distribution SPMD type application	Numerical integration	Goldbach conjecture
DP	Moving the time simulation loop to the GPU	Used in adaptive integration for subranges that need more accuracy	Finding pairs of prime numbers
UM	Transferring results for visualization on the CPU side	Transferring partial results for integrated subranges	Transferring tested numbers to the device
Tested optimizations	Various ways of updating heaters' temperatures, considering only areas with changes over threshold	Static, iterative and recursive version of integration algorithm for different functions	Different sizes of thread pools; helper boolean vector for primality test

3.2 Parallel heat distribution application-experiments

Jason Sanders and Edward Kandrot in their book 'CUDA by example' [25] 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 neighbors. 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:

$$T_N = T_O + \sum_{neighbors} k \times \left(T_{neighbor} - T_O\right) \tag{2}$$

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. Because we perform visualization, all described tests for this application were executed on the platform with GTX 970.

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%.

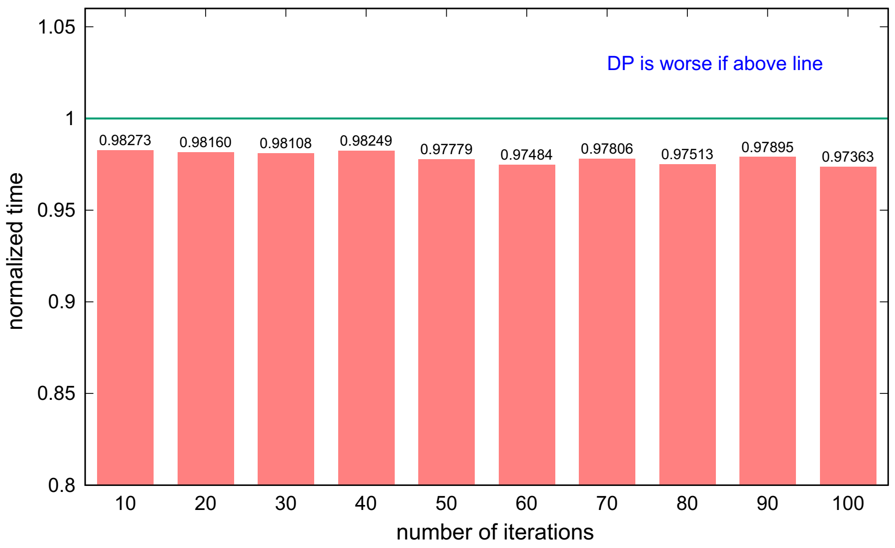

Fig. 1 Time needed to generate simulation frame depending on the number of iterations. Time measured for DP version is normalized to standard API version

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 Fig. 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, 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 five 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.

Table 2 Percentage of areas 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

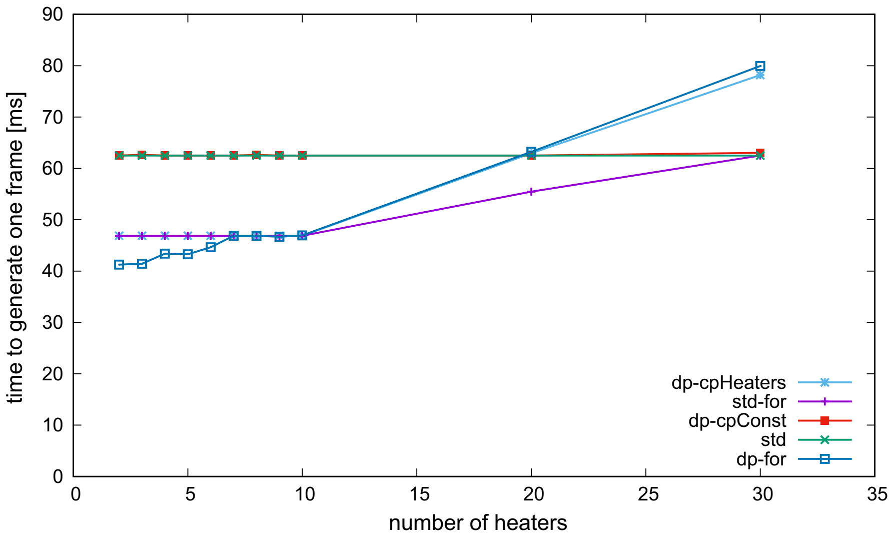

Fig. 2 Time to generate one frame based on the number of heaters

- `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 1024×1024px, and we performed 90 loops for every drawing. Each heater takes a 50×50 px area. The percentage of areas taken by heaters is shown in Table 2.

Measured times for each version are presented in Fig. 2. It turned out that for fewer than 20 heaters better approaches were those that copy individual heaters in loops. Depending on the number `dp-for` version (up to 10) or `dp-cpHeaters` was the 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 used the `dp-for` approach.

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 Fig. 3. It can be seen that for two heaters the `dp-for` approach is almost 40% faster.

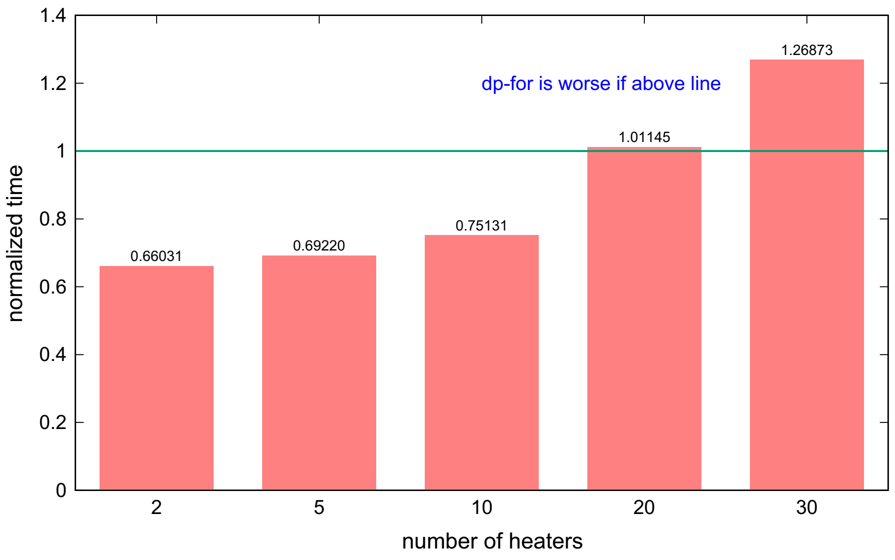

Fig. 3 Comparison of `dp-cpConst` and `dp-for` kernels. Time is normalized to measurements of `dp-cpConst` kernel

All in all, it can be seen that for a small number of heaters, non-const versions, i.e., those that update heaters only result in performance benefits because only a part of the grid needs to be updated. For a larger number of heaters, overhead for kernel invocations results in worse performance.

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 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 following 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 iteration (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 Fig. 4, show that the `dp-heur` version has improved performance two times because only a part of the grid needs to be updated. With time, when the simulation overtakes a greater area of the grid, performance becomes similar

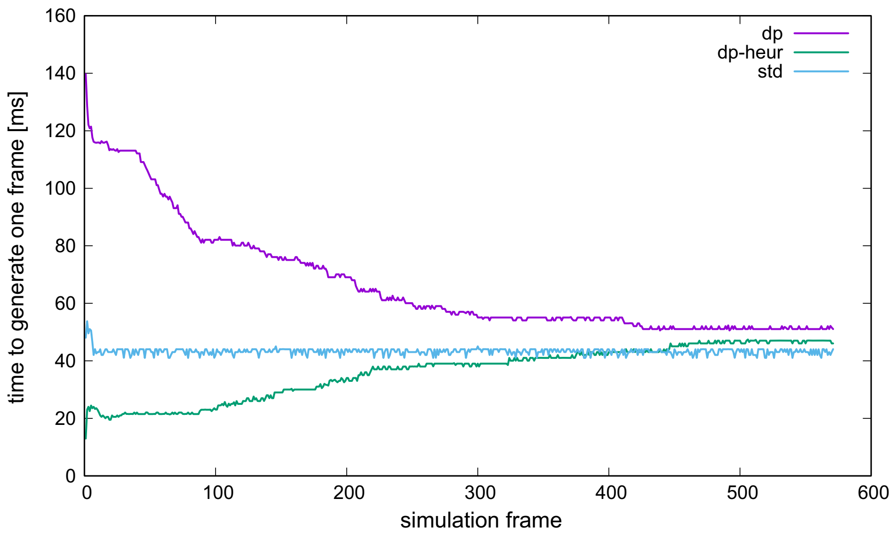

Fig. 4 Time needed to generate one frame depending on the duration of the simulation

to the standard version. On the other hand, *dp* approach caused a significant decrease in performance. It is caused by additional operations performed in every loop iteration.

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 visualization. Using UM required only replacing a memory allocation function and removing explicit data transfers. Consequently, there are no visible differences between the two versions in terms of performance. We measured times for different sizes of the simulation grid. We observed minor differences in execution times (1–2%) which is negligible (Fig. 5). More interesting was the 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. Consequently, it turned out that frequent transfers resulted in larger (percentage wise) execution times for the UM version (almost 10% for 10 iterations). Results for that test are presented in Fig. 6.

3.2.3 Summary

Heat transfer simulation created an opportunity to gain an advantage from utilizing the DP mechanism. We obtained an interesting performance gain even for a simplified physical model. On the other hand, UM caused a visible performance drop. For this application, in which case memory management is fairly simple, UM does not bring benefits.

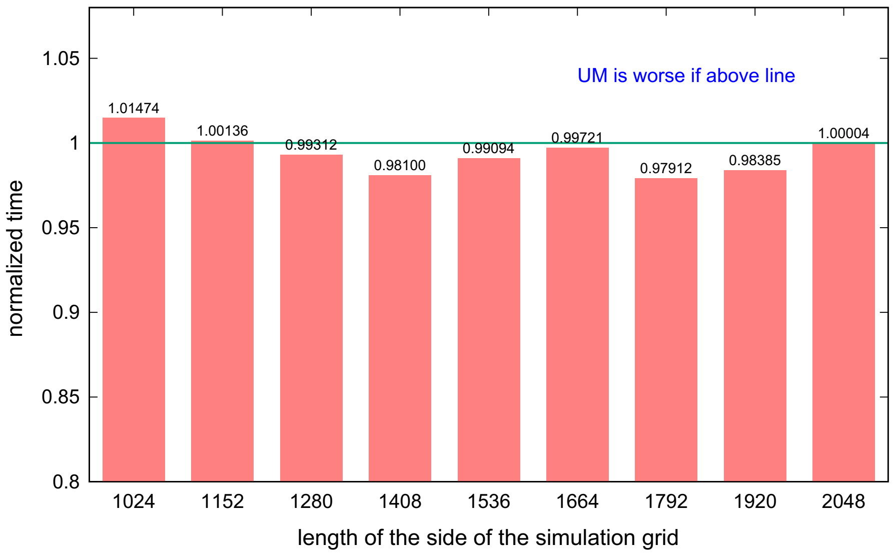

Fig. 5 Time needed to generate one frame for UM version compared to standard API version depending on simulation grid size—heat distribution application

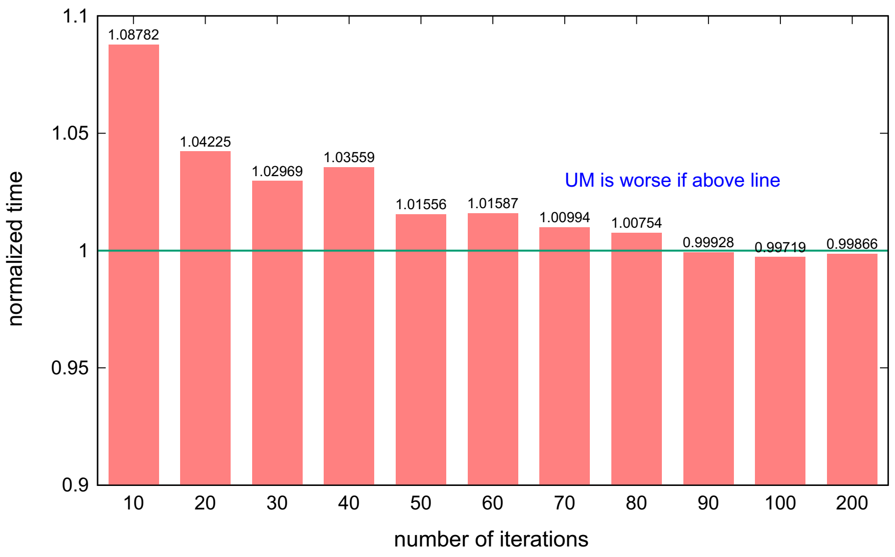

Fig. 6 Time needed to generate one frame for UM version compared to standard API version depending on number of iterations—heat distribution application

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. 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. Each 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 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 the other versions. In the `static` version, the most important is the 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:

$$\text{Integrated function 1:} \quad f(x) = \sin(x)\cos(x) \tag{3}$$

$$\text{Integrated function 2:} \quad f(x) = \mathrm{x}\sin(x)\cos(x) \tag{4}$$

$$\text{Integrated function 3:} \quad f(x) = \begin{cases} \mathrm{x}\sin(x)\cos(x) & x \le \frac{b-a}{2} \\ 2x & x > \frac{b-a}{2} \end{cases} \tag{5}$$

To test application behavior 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 DP with splitting into 32 parts,
– `adaptive2`—an adaptive version, without DP with splitting into 2 parts,
– `dp32`—an adaptive version, using DP with splitting into 32 parts,
– `dp2`—an adaptive version, using DP 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 integrated function 2 and are presented in Fig. 7. That function fluctuates more than integrated function 1. 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 eighth digit after the decimal point). 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 DP 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 integrated function 3, mentioned earlier. Measured execution times can be seen in Fig. 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.

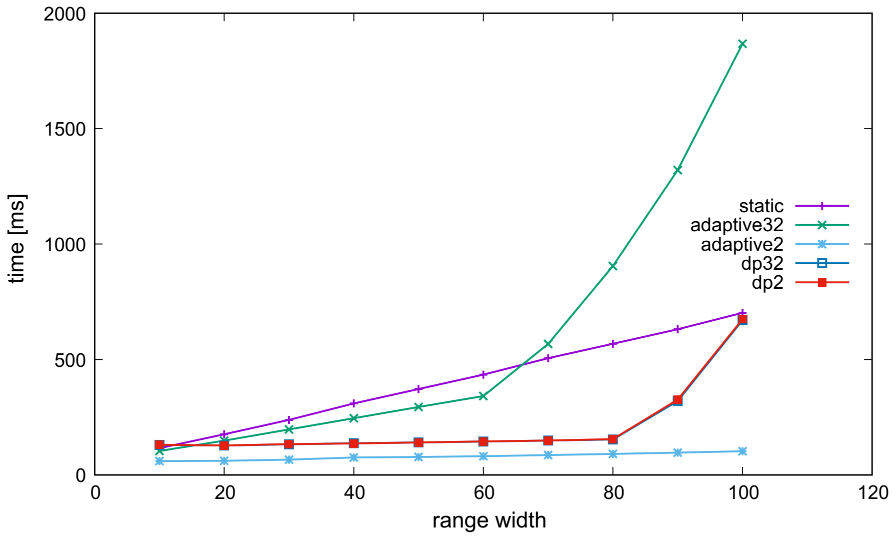

Fig. 7 Integration time depending on range width on GTX 970—integrated function 2

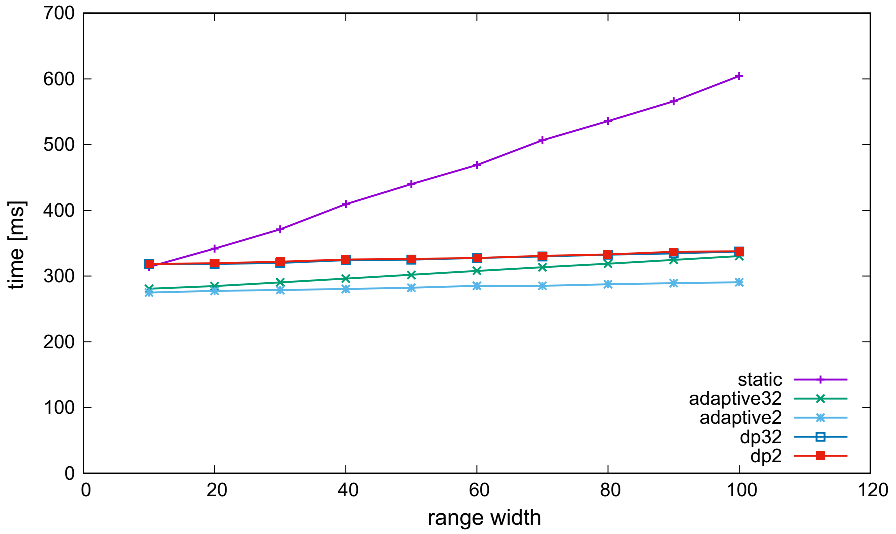

Fig. 8 Integration time depending on range width on Tesla K20 m—integrated function 3

It turned out that in this case the iterative implementation is slightly faster and at the same time more flexible. The former is due to the lack of overhead for kernel invocations compared to DP versions. Splitting into a larger number of parts also resulted in too much overhead at the given level. In almost every case, the adaptive 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 due to fewer number of computations.

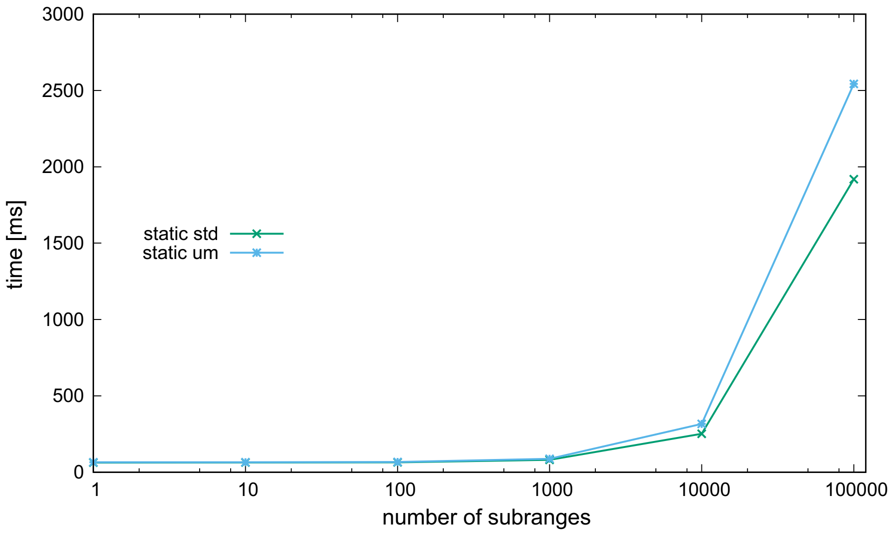

Fig. 9 Integration time depending on number of memory transfers on GTX 970—integrated function 1

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 sum relevant partial sums. The DP algorithm will only transfer one number with a final result. Partial sums are obtained on the device side.

Utilization of UM will affect efficiency of this part of the application. We measured execution times for different numbers of transfers. As a test function, we used integrated function 1 which was integrated from 0 to 100. We tested each implementation, i.e., we measured time with and without UM utilization. UM resulted in worse execution times for each tested version. In this application, a single result is transferred frequently as the result of the reduction kernel which uses shared memory and atomic add operations. Results are shown in Fig. 9.

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, DP 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 Figs. 10 and 11.

For versions with a split into two parts, the iterative approach was faster. The reason is that the split into two 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 two threads is not optimal in terms of warp utilization.

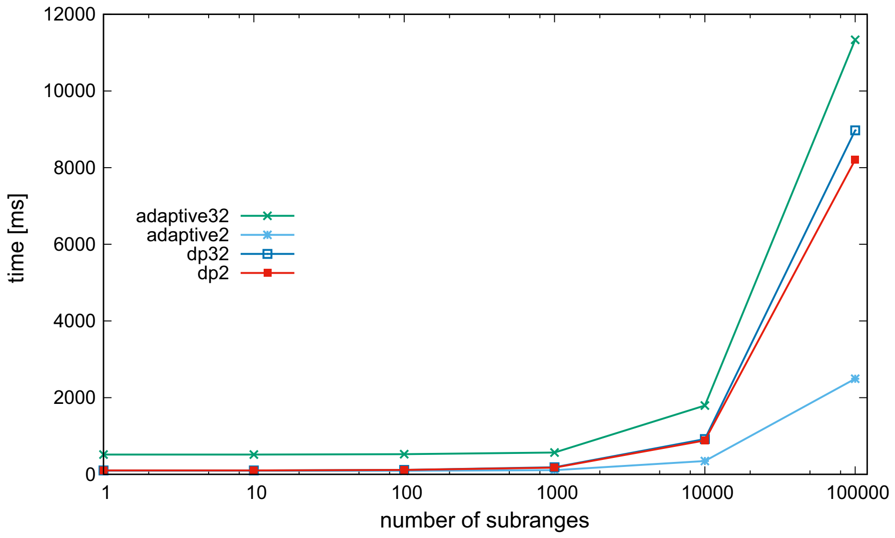

Fig. 10 Comparison of execution time for adaptive algorithms depending on number of memory transfers on GTX 970—integrated function 1

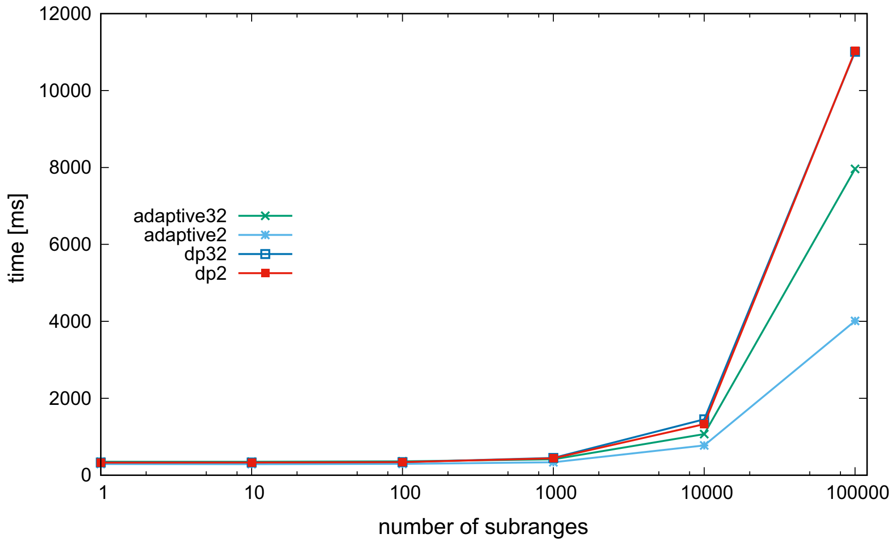

Fig. 11 Comparison of execution time for adaptive algorithms depending on number of memory transfers on Tesla K20 m—integrated function 1

Results for `32` versions are somewhat more interesting. On GTX 970, the approach utilizing DP 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 four times longer on the Tesla device (Table 3). As a result, the time which is saved on avoiding memory transfer is covered by kernels call times.

Table 3 Kernel call time on tested devices (ms)

	Time (ms)
Tesla K20 m	30.95
GTX 970	8.39

3.3.4 Summary

The first obvious conclusion is that UM 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 DP. 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 the 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 4×10^{17} [3,9]. A simple approach could be to check whether 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 DP 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.

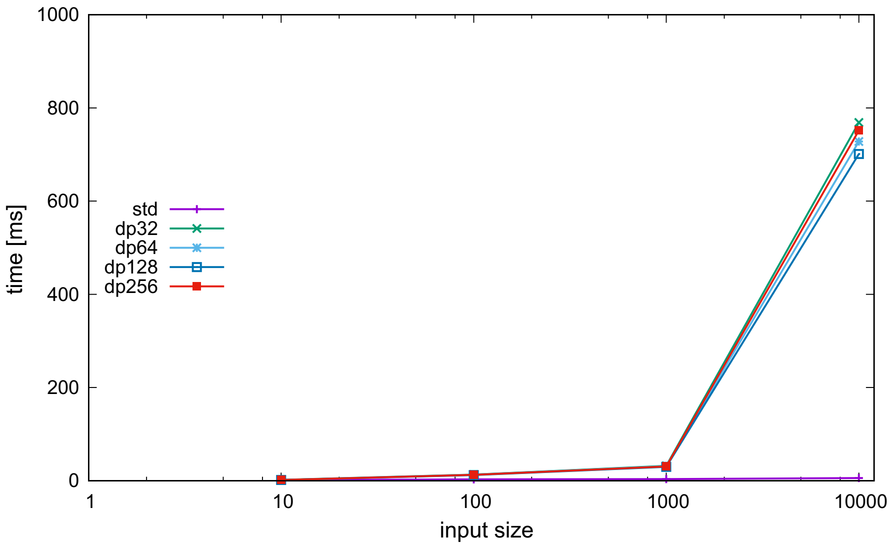

Fig. 12 Comparison of execution time DP version to standard API version on GTX 970

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 Fig. 12. It turned out that use of the DP caused a significant performance loss.

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 (e.g., 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, DP 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 UM

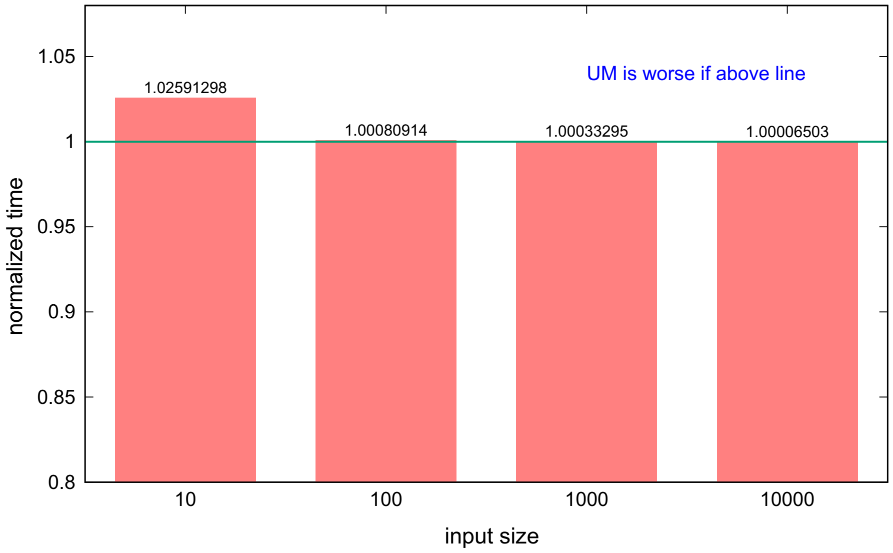

Fig. 13 UM version execution time normalized to standard API version time on GTX 970—Goldbach conjecture application

in both previous versions, the standard and the one with DP. Measured execution times are very similar for both versions (the standard and with UM) as shown in Fig. 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 four approaches in terms of access to that vector. We based on the standard version of the application (without DP):

- `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 UM, the vector is stored in constant memory,
- `um-global`—version with UM, the vector is stored in global memory (marked as `__managed__`).

We ran tests with 10,000 numbers as an input but we changed their range from 10^8 to 10^{12}. The collected results are shown in Fig. 14. Again, differences between versions were negligible. This is due to few and analogous copying operations before and after computations on a GPU in both versions. It can be concluded that in this case the memory management method is not important. The programmer can choose the most handy method.

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 DP caused performance

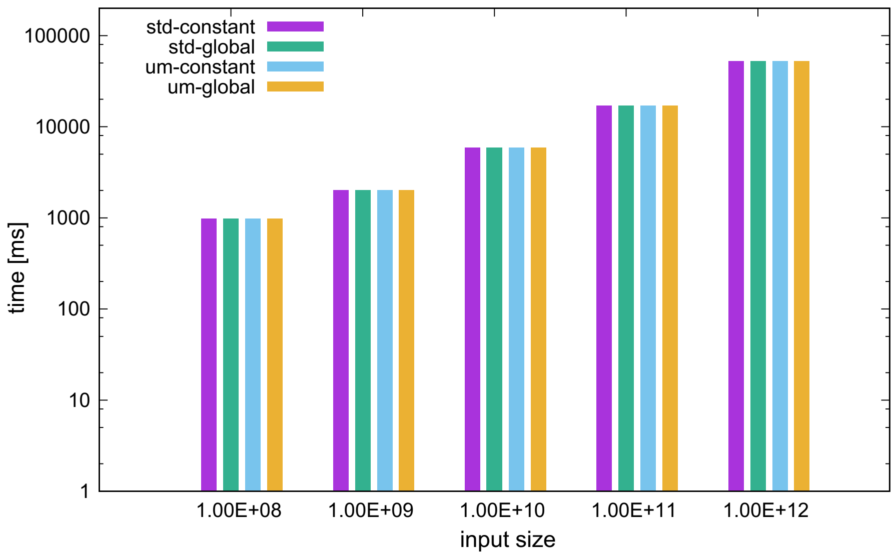

Fig. 14 Execution time for different memory management versions depending on input numbers range on GTX 970—Goldbach conjecture application

loss compared to the standard API version. On the other hand, UM 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, DP and UM 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 verification 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 verification 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%. These results are in line with the main goal of UM, i.e., improving ease of GPU programming.

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. In general, from the experiments we can conclude that DP offers benefits if it can be applied naturally in the algorithm and results in visibly fewer computations than a version without it. On the other hand, DP appeared to be worse than an approach without it if: DP introduced thread divergence or resulted in too much overhead from additional kernel calls.

In general, based on the experiments UM appears to be worse when transfer times relative to computations on a GPU are substantial and there is a possibility that more data compared to the standard version is transferred in the UM version. 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 (2014) Adaptive parallel computation with CUDA dynamic parallelism. https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/. Accessed 17 Feb 2016
2. Aliaga JI, Davidovic D, Pérez J, Quintana-Ortí ES (2015) Harnessing CUDA dynamic parallelism for the solution of sparse linear systems. In: Joubert GR, Leather H, Parsons M, Peters FJ, Sawyer M (eds.) Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1–4 September 2015, Advances in parallel computing, vol 27. IOS Press, Edinburgh, pp 217–226. doi: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 June 2016
4. Czarnul P (2003) Programming, tuning and automatic parallelization of irregular divide-and-conquer applications in DAMPVM/DAC. IJHPCA 17(1):77–93. doi:10.1177/1094342003017001007
5. Czarnul P (2016) Benchmarking performance of a hybrid intel xeon/xeon phi system for parallel computation of similarity measures between large vectors. Int J Parallel Program. doi:10.1007/s10766-016-0455-0
6. Czarnul P (2016) Parallelization of divide-and-conquer applications on intel xeon phi with an OpenMP based framework. Springer International Publishing, Cham, pp 99–111. doi:10.1007/978-3-319-28564-1_9
7. Czarnul P, Grzeda K (2004) Parallel simulations of electrophysiological phenomena in myocardium on large 32 and 64-bit linux clusters. In: Kranzlmüller D, Kacsuk P, Dongarra J (eds.) Recent Advances

in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Proceedings, Lecture Notes in Computer Science, vol 3241. Springer, Budapest, Sept 19–22, 2004, pp 234–241. doi:10.1007/978-3-540-30218-6_35

8. DiMarco J, Taufer M (2013) Performance impact of dynamic parallelism on different clustering algorithms. In: SPIE Defense, Security, and Sensing. International Society for Optics and Photonics, pp 87520E–87520E
9. Guy R (2013) Unsolved problems in number theory. Springer Science & Business Media, Berlin
10. Halliday D, Resnick R, Walker J (2013) Fundamentals of physics extended, 10th edn. Wiley, London
11. Jones S (2012) 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/. Accessed 11 June 2016
12. Joseph J, Keville K (2015) An evaluation of CUDA unified memory access on NVIDIA tegra k1. Waltham, MA USA. In: IEEE High Performance Extreme Computing Conference (HPEC'15) 19th Annual HPEC Conference
13. Khronos OpenCL Working Group, Editor: Lee Howes: The opencl specification version: 2.1, document revision: 23 (2015). https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf
14. Landaverde R, Zhang T, Coskun AK, Herbordt M (2014) An investigation of unified memory access performance in CUDA. In: High Performance Extreme Computing Conference (HPEC), 2014 IEEE, pp 1–6
15. Li D, Wu H, Becchi M (2015) 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. ACM, New York, pp 5:1–5:1. doi:10.1145/2723772.2723780
16. 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 (CCGrid), pp 1092–1098. doi:10.1109/CCGrid.2015.105
17. Mehta V (2015) Exploiting CUDA dynamic parallelism for low power arm based prototypes. In: GPU Technology Conference, San Jose. http://on-demand.gputechconf.com/gtc/2015/presentation/S5384-Vishal-Mehta.pdf
18. Mei G (2014) Evaluating the power of GPU acceleration for IDW interpolation algorithm. Sci World J 2014. Article ID 171574. doi:10.1155/2014/171574
19. Negrut D, Serban R, Li A, Seidl A (2014) Unified memory in CUDA 6.0. a brief overview of related data access and transfer issues. Tech. Rep. TR-2014-09, University of Wisconsin–Madison
20. NVIDIA Corporation: Dynamic Parallelism in CUDA (2012). http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf
21. NVIDIA Corporation: NVIDIA CUDA C Programming Guide (2017). http://docs.nvidia.com/cuda/cuda-c-programming-guide
22. Plauth M, Feinbube F, Schlegel F, Polze A (2015) Using dynamic parallelism for fine-grained, irregular workloads: a case study of the n-queens problem. In: 2015 3rd International Symposium on Computing and Networking (CANDAR), pp 404–407. doi:10.1109/CANDAR.2015.26
23. Plauth M, Feinbube F, Schlegel F, Polze A (2016) A performance evaluation of dynamic parallelism for fine-grained, irregular workloads. Int J Netw Comput 6(2):212–229. http://www.ijnc.org/index.php/ijnc/article/view/126
24. Sakharnykh N (2015) Combine openacc and unified memory for productivity and performance. https://devblogs.nvidia.com/parallelforall/combine-openacc-unified-memory-productivity-performance/
25. Sanders J, Kandrot E (2010) CUDA by example: an introduction to general-purpose GPU programming, 1st edn. Addison-Wesley Professional, Reading
26. Souto RP, Osthoff C, de Vasconcelos AT, Augusto DA, da Silva Dias PL, Rodriguez A, Trelles O, Ujaldon M (2014) Applying GPU dynamic parallelism to high-performance normalization of gene expressions. GPU Technology Conference, San Jose. http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4209_biofinformatics_sort_dynamic_parallelism.pdf
27. Theano Development Team (2016) Theano: a python framework for fast computation of mathematical expressions. http://arxiv.org/abs/1605.02688
28. Wang J, Yalamanchili S (2014) Characterization and analysis of dynamic parallelism in unstructured GPU applications. In: 2014 IEEE International Symposium on Workload Characterization (IISWC), pp 51–60. doi:10.1109/IISWC.2014.6983039
29. Wilkinson B, Allen M (2004) Parallel programming: techniques and applications using networked workstations and parallel computers, edition edn. Pearson. ISBN 978-0131405639

30. Zhang P, Holk E, Matty J, Misurda S, Zalewski M, Chu J, McMillan S, Lumsdaine A (2015) Dynamic parallelism for simple and efficient GPU graph algorithms. In: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, IA3'15. ACM, New York, pp 11:1–11:4. doi:10.1145/2833179.2833189