PROJECT PROPOSAL PHPC-2019

A High Performance Implementation of a Shallow Water Wave Equation With a Finite Volume Solver

Principal investigator (PI)	Jiahua Wu
Institution	EPFL-CSE
Address	Station 1, CH-1015 LAUSANNE
Involved researchers	Only PI
Date of submission	May 26, 2019
Expected end of project	July 1, 2019
Target machine	Mount Everest
Proposed acronym	WAVE

Abstract

Numerical simulation is a powerful tool in research of oceanography. In this case, we present a parallel implementation of a finite volume solver that solves a shallow water wave equation. Our application is coded in C and uses the Compute Unified Device Architecture (CUDA), a parallel computing platform and application programming interface (API) model created by Nvidia.

1 Scientific Background[1]

A parallel iterative finite volume scheme for solving a shallow water wave equation calculated on GPU using Compute Unified Device Architecture (CUDA) is presented. This method is based on a domain decomposition where the global 2D domain is divided into multiple structured rectangular uniform mesh. Denoting $I_{i,j} = \left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right] \times \left[y_{j-\frac{1}{2}}, y_{j+\frac{1}{2}}\right]$ as one of the subdomains, we approximate $q\left(x_i, y_j, t_n\right)$ by the cell average $\overline{q}_{ij}\left(t_n\right)$ at each cell $I_{i,j}$ for a certain time-step t_n:

$$I_{i,j} = \left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right] \times \left[y_{j-\frac{1}{2}}, y_{j+\frac{1}{2}}\right] \approx \overline{q}_{ij}\left(t_n\right) = \frac{1}{\Delta x \Delta y} \int_{I_{i,j}} q\left(x, y, t_n\right) dydx \tag{1}$$

As for the discretization, the Lax-Friedrichs method is employed. This method is explicit and the update rule is given as follows:

$$\begin{aligned} \overline{h}_{i,j}^{n+1} &= \frac{1}{4}\left[\overline{h}_{i,j-1}^{n} + \overline{h}_{i,j+1}^{n} + \overline{h}_{i-1,j}^{n} + \overline{h}_{i+1,j}^{n}\right] \\ &+ \frac{\Delta t^n}{2\Delta x}\left[\overline{hu}_{i,j-1}^{n} - \overline{hu}_{i,j+1}^{n} + \overline{hv}_{i-1,j}^{n} - \overline{hv}_{i+1,j}^{n}\right] \end{aligned} \tag{2}$$

$$
\begin{aligned}
\overline{hu}_{i,j}^{n+1} &= \frac{1}{4}[\overline{hu}_{i,j-1}^{n} + \overline{hu}_{i,j+1}^{n} + \overline{hu}_{i-1,j}^{n} + \overline{hu}_{i+1,j}^{n}] - \Delta t^n g h^{-}_{i,j} n + 1 \\
&\quad + \frac{\Delta t^n}{2\Delta x}\left[\frac{\left(\overline{hu}_{i,j-1}^{n}\right)^2}{\overline{h}_{i,j-1}^{n}} + \frac{1}{2} g \left(\overline{h}_{i,j-1}^{n}\right)^2 - \frac{\left(\overline{hu}_{i,j+1}^{n}\right)^2}{\overline{h}_{i,j+1}^{n}} - \frac{1}{2} g \left(\overline{h}_{i,j+1}^{n}\right)^2\right] \\
&\quad + \frac{\Delta t^n}{2\Delta x}\left[\frac{\overline{hu}_{i-1,j}^{n}\overline{hu}_{i-1,j}^{n}}{\overline{h}_{i-1,j}^{n}} - \frac{\overline{hu}_{i+1,j}^{n}\overline{hu}_{i+1,j}^{n}}{\overline{h}_{i+1,j}^{n}}\right]
\end{aligned}
\tag{3}
$$

$$
\begin{aligned}
\overline{hv}_{i,j}^{n+1} &= \frac{1}{4}\left[\overline{hv}_{i,j-1}^{n} + \overline{hv}_{i,j+1}^{n} + \overline{hv}_{i-1,j}^{n} + \overline{hv}_{i+1,j}^{n}\right] - \Delta t^n g \overline{h}_{i,j}^{n+1} z_y \\
&\quad + \frac{\Delta t^n}{2\Delta x}\left[\frac{\overline{hu}_{i,j-1}^{n}\overline{hv}_{i,j-1}^{n}}{\overline{h}_{i,j-1}^{n}} - \frac{\overline{hu}_{i,j+1}^{n}\overline{hv}_{i,j+1}^{n}}{\overline{h}_{i,j+1}^{n}}\right] \\
&\quad + \frac{\Delta t^n}{2\Delta x}\left[\frac{\left(\overline{hv}_{i-1,j}^{n}\right)^2}{\overline{h}_{i-1,j}^{n}} + \frac{1}{2} g \left(\overline{h}_{i-1,j}^{n}\right)^2 - \frac{\left(\overline{hv}_{i+1,j}^{n}\right)^2}{\overline{h}_{i+1,j}^{n}} - \frac{1}{2} g \left(\overline{h}_{i+1,j}^{n}\right)^2\right]
\end{aligned}
\tag{4}
$$

The index n on Δt implies an update of the time-step length at each iteration. This update is to ensure that the CFL condition is satisfied. The time-step is given by:

$$
\Delta t^n = \min_{i,j} \frac{\Delta x}{\sqrt{2}\nu_{i,j}^{n}}
\tag{5}
$$

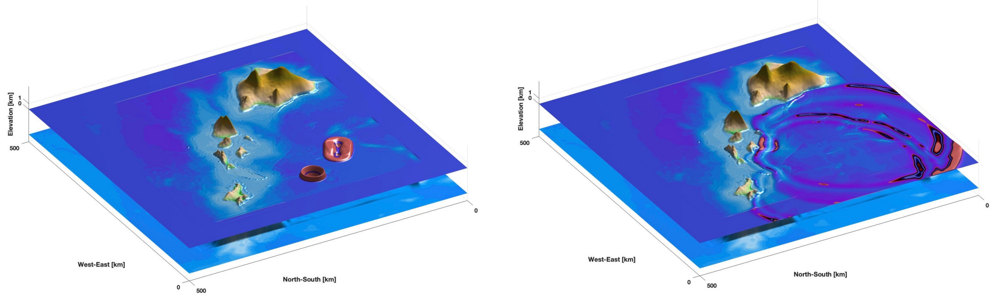
where ν^n is calculated via:

$$
\nu_{i,j}^{n} = \sqrt{\max\left(\left|\frac{\overline{hu}_{i,j}^{n}}{\overline{h}_{i,j}^{n}} + g\overline{h}_{i,j}^{n}\right|, \left|\frac{\overline{hu}_{i,j}^{n}}{\overline{h}_{i,j}^{n}} - g\overline{h}_{i,j}^{n}\right|\right)^2 + \max\left(\left|\frac{\overline{hu}_{i,j}^{n}}{\overline{h}_{i,j}^{n}} + g\overline{h}_{i,j}^{n}\right|, \left|\frac{\overline{hu}_{i,j}^{n}}{\overline{h}_{i,j}^{n}} - g\overline{h}_{i,j}^{n}\right|\right)^2}
\tag{6}
$$

After each iteration, cells lower than a given water threshold are set inactive:

$$
\overline{h}_{i,j}^{n+1} \le 0 \rightarrow \overline{h}_{i,j}^{n+1} = 10^{-5} \quad \overline{h}_{i,j}^{n+1} \le 10^{-4} \rightarrow \overline{hv}_{i,j}^{n+1} = 0, \overline{hv}_{i,j}^{n+1} = 0
$$

The equation is discretized on a 2D rectangular domain of size $nx \times nx$.



(a) Wave at T=0s (b) Wave at T=0.2s

Figure 1: Initial Condition and Solution of the Shallow Wave Equation

2 Implementations

The application was initially implemented in Matlab. We have investigated two other implementations of the same solver:

- A serial version in C++
- A parallelized version with CUDA in C++

The serial code has been written in C++11 standard and complied with VC++ compiler. The parallel code has been developed using CUDA Toolkit on MS-Windows 10 and the CDUA version is 10.1. The serial code has been debugged with the help of `gdb` and the CUDA code has been debugged through `Nsight` and `CUDA Memory Checker`.

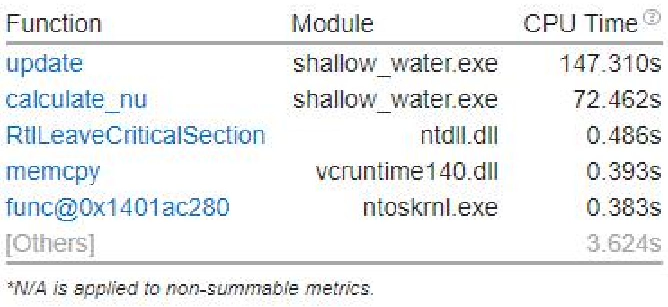
3 Serial Code Profiling

The serial code is mainly comprised of two functions `calculate_nu` and `update`. `calculate_nu` corresponds to Eq.5 and Eq.5 (Calculation of time-step) and `update` corresponds to Eq.2, Eq.3 and Eq.4 (Update of H,HV and HU). We profile the serial code compiled with/without optimization through Intel Vtune Amplifier and the results are summarized in Table1.

Compile Options	Elapsed Time/s	Average DRAM Bandwidth/GB/s	CPI Rate
/Od	903.782	2.457	0.630
/O2 + /fp:fast	235.637	6.147	2.260

Table 1: Comparison of metrics of performance with/without optimization

The compilation flag /Od means no optimization is applied. The average DRAM bandwidth is much less than the Max DRAM Single-Package Bandwidth measured by Vtune before the collection starts(21GB/s). The CPI(Cycles per instruction Retired) Rate of the code is smaller than one and this means that the code is instruction bound.
After the use of fast-math(complied with /fp:fast) and vectorization(complied with /O2), the numerical results remain reliable (maximum relative error is of order of magnitude 10^{-14}). As for performance, although the memory bandwidth improves, it is still far from the max bandwidth and the value of CPI(greater than one) indicates that the code is clearly memory bound. Besides, this augmentation of CPI is partly due to vectorization which has the effect of replacing many single-data math instructions with fewer multiple data math instructions. Such multiple-data instructions are more complex and take longer to execute, which result in higher CPI. To find out the part of code that accounts for the most use of memory, we have turned to the "Memory usage" viewpoint and have found that the most intense memory use is in the update part of the function `update` because the update of a point requires the value of the surrounding points. Regardless of our efforts, the improvement is still negligible.
As revealed by the results shown in figure2, the two top hotspots are `update` and `calculate_nu` and we endeavour to parallelize these two parts using CUDA with a view to a better performance.



Function	Module	CPU Time
update	shallow_water.exe	147.310s
calculate_nu	shallow_water.exe	72.462s
RtlLeaveCriticalSection	ntdll.dll	0.486s
memcpy	vcruntime140.dll	0.393s
func@0x1401ac280	ntoskrnl.exe	0.383s
[Others]		3.624s

*N/A is applied to non-summable metrics.

Figure 2: Top Hotspots given by Vtune profiler

4 Parallelization of the serial code

4.1 Overview

We shall start parallelizing the serial code with the help of CUDA. As for structure of the map, the grid is defined such that one thread corresponds to one element in the resulting matrix (H) and since the width and the length of the matrices H, HV and HU are not divisible by the size of blocks, we add "padding" to the matrix (Figure3) to avoid invalid access to memory by the threads whose indices exceed the size of the matricew. (We can also use thread divergence but that would do harm to performance because each thread needs to execute the conditional statement).

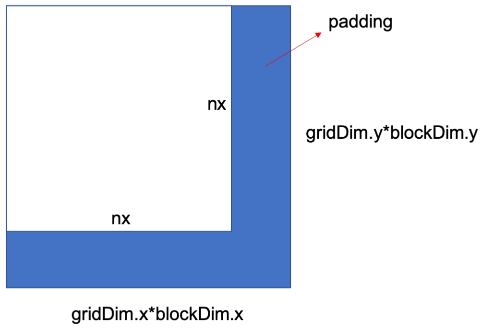


Figure 3: Padding the matrices to fit in the gird

For CUDA the function names are adopted from the serial version, i.e., `calculate_nu` take charge of the calculation of ν(Eq.5) and `update` is responsible for the update of matrices(Eq.2, Eq.3, Eq.4) Additionally, we introduce `max_reduction` to perform the max reduction technique for the computation of time-step(Eq.6) and its principle is illustrated in Figure 4. In implementation, inspired by [2], we make use of Kepler's shuffle instruction which allows threads to directly read a register from another thread in the same warp . The threads belonging to the same warp can therefore exchange data and do a warp reduction. Afterwards, a block reduction is performed to get the max inside a block. Finally, we need to compare the max of different blocks to obtain a global max. For the last step, we exploit the atomic operations to simplify the code (otherwise we will need to launch a new kernel to do another max reduction). Note that the `atomicMax` function in CUDA libary does not support double precision float, we overload this function by using `atomicCAS` to define a new atomic max operation for double.

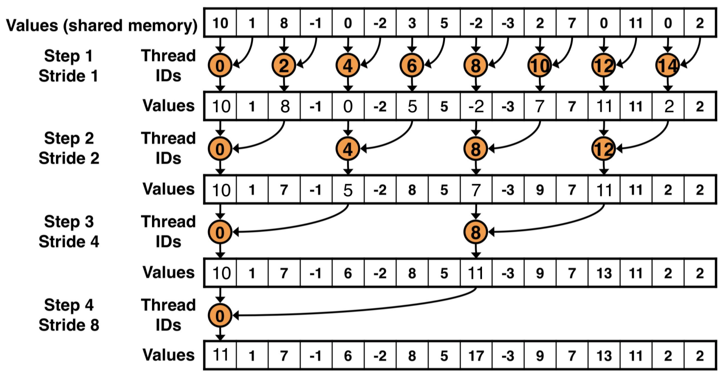


Figure 4: Illustration of max reduction(modified from [3])

4.2 Memory Optimization

Although directly manipulating the global memory seems intuitive and convenient, one should avoid repeatedly loading/writing data from/to global memory during computation because global memory not only has low bandwidth but also we may encounter the problem of uncoalesced memory accesses which will drag down the performance. On the contrary, we should make use of the on-chip shared memory which has the advantage of higher bandwidth and lower latency. Furthermore, shared memory provides more flexible memory access pattern: Except for memory bank conflicts, non-sequential or unaligned accesses by a warp in shared memory will not do harm to the performance. Therefore, it is preferable to load and store data from global memory and then reorder it in shared memory.

In order to take advantage of the shared memory, we employ a tiling approach in which each thread block loads a tile of data from H,HU,HV,Zdx and Zdy into shared memory, so that each thread in the block can access all elements of the shared memory tile as needed. However, in order to perform the update rule inside a block, the data at the boundary of the neighbouring blocks are needed. To ensure that the blocks can execute independently, each block not only loads the data corresponding to the threads inside but also loads one additional $1 \times sidelength$ section data at each side(Figure5). The update results will be written back to the global memory for next update.

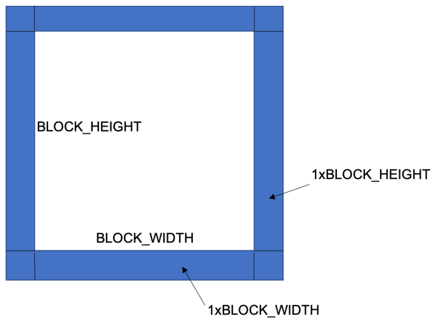


Figure 5: Illustration of data tiling

Besides, we load the frequently used constants(g, dx and nx) to the constant memory which is cached on chip and provides faster access.

5 Results and Analysis

To access the performance of the implemented algorithm, we have run the code at 2001×2001 resolution and have profiled the code with NVIDIA `Visual Profiler` which provides detailed information about the involved kernels. All the experiments have been conducted on a NVIDIA GeForce GTX 850M graphic card with specification presented in Table2.

Compute Capability	5.0
Number of Multiprocessors	5
Multiprocessor Clock Rate	901.5MHz
Max. Shared Memory per Block	48 kB
Max. Shared Memory per Multiprocessor	64 kB
Single Precision FLOP/s	1.154 TFLOP/s
Double Precision FLOP/s	36.06 GFLOP/s
Threads per Warp	32
Global Memory BandWidth	28.8 GB/s
Global Memory Size	4GB
L2. Cache Size	2MB
Constant Memory Size	64 kB

Table 2: Specification of the GPU used

5.1 Performance-Critical Kernels

The running of the kernel takes 58.293s and the average relative error is 0.0071799 for a grid of size 126×126 with block size 16×16 and the rank of the running time occupation of the kernels is given in Table3.

Time Occupation	Kernel
75.5%	`update`
20.9%	`calculate_nu`
3.6%	`max_reduction`

Table 3: Running Occupation of the kernels

Still, the calculation of the update rule is the most time-consuming part. For the first two kernels in the rank, we examine their compute/memory unit usage and the result is shown in Figure 6 and Figure 7.

As we can notice, the two kernels' memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on SMs. Having observed in the GPU specification(Table.2) that its single precision FLOP is much higher than its double precision FLOP, we have supposed that the use of single precision number instead of double precision may enhance the performance of the kernel. However, the loss of precision renders the results unacceptable, we obtained `NaN` during the calculation and the error is extremely high.

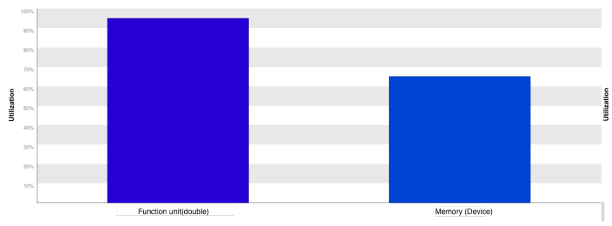


Figure 6: `calculate_mu`

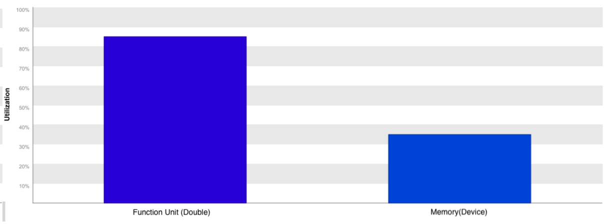


Figure 7: `update`

Furthermore, as reported by the profiler, the GPU suffers from low occupancy (24.75%) in kernel `update`. This is due to the usage of shared memory: The kernel uses 21.969 KB of shared memory for each block and as the maximum shared memory per block is 48kB, each SM is limited to simultaneously executing 2 blocks (16 warps) while it can execute 32 blocks at most at the same time. To reduce the use of shared memory, we choose not to load Zdx and Zdy to shared memory and to read it directly from global memory since they are only used once during the update. In this way the shared memory use is reduced by 4kB and the occupancy is increased to 37.2%. As a result, the execution time is reduced to 52.806s.
In addition, the "Shared Memory Access Pattern" Check shows that the kernel `update` encounters some bank conflicts during the assignment of boundary conditions of Ht, HVt and HUt. However, to fully address this problem and to prevent two threads access the same bank at the same time, the width of the shared memory should be a multiple of 33 since shared memory is organized into 32 banks. This will result in a suboptimal block size and since the bottleneck is the computation part as shown above, we choose not to adjust the shared memory size.

5.2 Optimized Block Size

According to [4], since the GPU schedules the same instruction to a warp in SIMD(single instuction multiple data) fashion, to achieve an optimal performance, the number of threads in a block should be a multiple of the warp size (in our case 32). In order to verify this principle, we test several block size and the according execution time is summarized in Table4

Block Size	Execution Time
16×16	52.806s
32×32	52.901s
10×10	67.410s
13×13	60.753s

Table 4: Running of different block sizes

We observe that there is no much difference in performance between the block sizes which are multiples of 16 while for block sizes that are not divisible by 16, a decrease in performance can be expected.

5.3 An important Factor in GPU Performance

One of the most important characteristics of CUDA is the stream which allows a concurrent execution of the kernels and allows the transfer of the memory to overlap with the execution of the kernels. Hence, one important factor of GPU performance is the level of overlapping. The more Memcpy and kernels overlap, the higher acceleration can be achieved. However, in our context, the kernels are inherently dependent: the calculation of the update depends on the time-step and the update of HU and HV depends on H. Therefore the nature of the problem prevents us from establishing a concurrency of the kernels.

6 Conclusion

In this project, we have developed a serial version as well as a CUDA-parallelized version of a solver on shallow water equations. We have accessed and analyzed their performance with the help of different profilers. As we can see, the parallelized version using CUDA provides a speedup of around 10 times compared to the serial version. However, the work would be more complete if a performance model assessing the scalability of a CUDA code and explaining the difference of CPU and GPU can be found.

References

[1] phpc project description

[2] https://devblogs.nvidia.com/faster-parallel-reductions-kepler

[3] https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[4] Andr R. Brodtkorb, Martin L. Stra, Mustafa Altinakar, Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation, Computers & Fluids, Volume 55, 2012, Pages 1-12, ISSN 0045-7930, https://doi.org/10.1016/j.compfluid.2011.10.012.