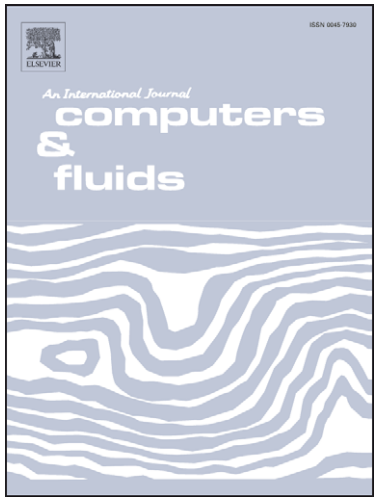
**Computers & Fluids, Volume 46, Issue 1, July 2011, Pages 101–105**

# Accepted Manuscript




Higher-order CFD and Interface Tracking Methods on Highly-Parallel MPI and GPU systems

J. Appleyard, D. Drikakis

PII: S0045-7930(10)00287-2
DOI: 10.1016/j.compfluid.2010.10.019
Reference: CAF 1430

To appear in: *Computers & Fluids*

Received Date: 29 April 2010
Revised Date: 15 October 2010
Accepted Date: 24 October 2010

Please cite this article as: Appleyard, J., Drikakis, D., Higher-order CFD and Interface Tracking Methods on Highly-Parallel MPI and GPU systems, *Computers & Fluids* (2010), doi: 10.1016/j.compfluid.2010.10.019

# Higher-order CFD and Interface Tracking Methods on Highly-Parallel MPI and GPU systems

J. Appleyard, D. Drikakis

*Fluid Mechanics and Computational Science Department*

*Cranfield University*

*United Kingdom*

**Abstract**

A computational investigation of the effects on parallel performance of higher-order accurate schemes was carried out on two different computational systems: a traditional CPU based MPI cluster and a system of four Graphics Processing Units (GPUs) controlled by a single quad-core CPU. The investigation was based on the solution of the level set equations for interface tracking using a High-Order Upstream Central (HOUC) scheme. Different variants of the HOUC scheme were employed together with a $3^{rd}$ order TVD Runge-Kutta time integration. An increase in performance of two orders of magnitude was seen when comparing a single CPU core to a single GPU with a greater increase at higher orders of accuracy and at lower precision.

*Key words:* Interface Tracking, High-Order Schemes, GPU

## 1. Introduction

Developments in Graphics Processing Units (GPUs) have recently allowed for them to be easily used as general purpose massively parallel (thousands of concurrently running threads) computing devices. While these advancements were originally intended to compute complex visual effects on large groups of pixels for computer games, it was found that the same technology could be applied to scientific computing. Table 1 illustrates the superior peak theoretical computational properties of the NVIDIA Tesla C1060 GPU compared to those of a top of the range quad-core CPU (Intel i7-975). While the pricing of the components can vary, at the time of writing the C1060 is approximately 25% more expensive than the i7.

| | Arithmetic Throughput | | Memory |
|---|---|---|---|
| Device | Single Precision | Double Precision | Bandwidth |
| C1060 | 933 GFLOPS | 78 GFLOPS | 102 GB/s |
| i7 | 55 GFLOPS | 55 GFLOPS | 26 GB/s |

Table 1: Properties of top of the range hardware [1–3].

One of the first scientific applications for GPUs was presented by Lengyel et al. [4] and concerned robot motion planning. Many other applications have since followed [5–8] and the field is continually growing.

In this work we are interested in the application of GPUs to CFD problems; specifically focusing on the impact of both higher order methods and varying precision on the relative performance changes between CPUs and GPUs. In the past few years many authors [9–14] have studied performance changes due to implementing CFD codes on GPUs and have found performance improvements of one to two orders of magnitude. Indicatively, we report that Antoniou et al. [9] recently showed a performance increase by a factor of 53 when comparing four GPUs to a quad-core CPU using a finite-difference WENO scheme in single precision. Cohen and Molemaker [10] showed a sim-

*Email addresses:* j.appleyard@cranfield.ac.uk (J. Appleyard), d.drikakis@cranfield.ac.uk (D. Drikakis)

ilar per-core performance increase solving the incompressible 3D Navier-Stokes equation in double precision.

Despite the above studies, there is still little information as to the effect of varying precision and accuracy on the performance of these computational methods and codes, and this has motivated the present study. In this work the performance of high-order level set methods will be demonstrated on two different massively parallel architectures. The first architecture is a set of four NVIDIA Tesla C1060 GPUs. The second architecture is a HPC facility based on a 856 processor HP XC Cluster built in 2007.

This paper is organised as follows. In the next section the level set method is described briefly. The GPU architecture is outlined in Section 3 and the algorithms used to solve the level set equation are described in Section 4. Section 5 presents the results from GPU and CPU implementations at different grid sizes and with different orders of accuracy. The conclusions drawn from the present study are summarised in Section 6.

## 2. The Level Set Method

To simulate motion of an interface the scalar function $\phi(x,t)$ is introduced. This function is initialised as a signed distance function representing the distance from the interface. The position of the interface at time $t$ is therefore defined by the isosurface given by $\phi(x,t) = 0$. This function is advected by solving the level set equation [15]:

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \bullet \nabla \phi = 0$$

In this work the level set equation is solved using High-Order Upstream Central (HOUC) finite element schemes [16] (simple upwinding expanded to higher orders) to update $\phi$ at each time step.

These schemes have been found to be more efficient than traditional essentially non-oscillatory WENO or ENO schemes and, in the case of the smooth level set function, cause no detriment to numerical stability [16]. A $3^{rd}$-order TVD Runge-Kutta method was used to advance time [17].

## 3. GPU Architecture

A Tesla C1060 GPU comprises 30 multiprocessors each containing eight scalar cores. Each multiprocessor is capable of running up to 1024 threads in parallel, although only a small proportion of these are executing instructions at any given time. This massive level of parallelisation allows for significant hardware optimisations and is the main reason that GPUs can perform so well compared to CPUs [18].

The GPU implementation presented in this work is based upon NVIDIA's CUDA (Compute Unified Device Architecture) programming model. The CUDA environment comprises extensions to both the C and Fortran programming languages allowing the CFD code to be written specifically for NVIDIA GPUs. These extensions allow large numbers of threads to be launched on the GPU from within a program running on the CPU. With a few exceptions, executing the CFD code can be very similar to executing a code written to run on a CPU.

One of the most important aspects of programming and optimising in CUDA is memory management. This is due to the high arithmetic intensity (ratio of floating operations per second to memory bandwidth) of GPUs. Given a GPU can attain 933 GFLOPS in single precision with only 102 GB/s of global memory bandwidth, we can calculate that for each four byte floating point number loaded from global memory 37 floating point operations must be completed to maximise floating point throughput. As very few applications have such a high arithmetic intensity most codes are strongly limited in performance by memory access speeds making efficient use of memory highly desirable. In double precision this becomes less important (seven operations per load), however it still remains significant.

The two main methods for maximising memory bandwidth on a GPU are memory re-use and memory coalescing:

### *3.1. Memory re-use*

To understand how memory can be re-used one must first understand the different type of memory available on the GPU. The three main memory types on the Tesla C1060 are:

1. Global memory: accessible to every thread on every multiprocessor. This is the main memory type and a Tesla C1060 has 4GB.
2. Shared memory: accessible to every thread in a single block. Each block resides on a single multi-processor and the total shared memory for all the blocks on a single multi-processor is limited to 16KB on the Tesla C1060.
3. Register memory: accessible to an individual thread only. Limited to 64KB per multi-processor on the Tesla C1060.

Both shared and register memory are very high bandwidth and very low latency whereas global memory can be orders of magnitude slower. For this reason if memory that can be re-used is stored in either register or shared memory it is possible to greatly accelerate the application by reducing usage of the slow global memory. This is the core idea behind the algorithm described in Section 4.

### *3.2. Memory coalescing*

Memory coalescing allows for the minimum instructions necessary to be issued to access data from global memory. Given the highly parallel nature of most GPU applications, the hardware is designed to be most efficient when a small group of consecutively assigned threads accesses consecutive memory locations. This access pattern is known as a coalesced access pattern. If the memory structures can be designed so that coalesced accesses are possible then the memory throughput of the application can be increased by up to an order of magnitude over an application with random memory access.

## 4. Algorithms

### *4.1. GPU*

The method presented here is designed to solve the level set equation on a uniform three dimensional grid, though it is also applicable to many non-uniform grid problems. The method is similar to the method used by both Brandvik and Pullan [12] and Micikevicius [19] and is as follows:

Firstly, the solution space is subdivided into $n$ equally sized cuboidal domains. Each domain spans the solution space in one dimension. The sizes of the other two dimensions are then calculated based on four factors:

1. The size of the solution space in these directions. It is most efficient to have its side length as a factor of the length of the solution space. If this is not the case additional logic is required to prevent threads outside of the domain from executing.
2. The limitations on the availability of the fast shared memory and register spaces on the GPU. Larger cuboids require more shared memory. It is also more efficient to have a cross-section as close to square as possible so as to minimise boundary data.
3. Memory coalescing requirements. Memory can be coalesced if the length of one side is a multiple of eight in single precision, four in double precision.
4. Number of shared memory bank conflicts. If one side length is a multiple of 16 then shared memory is guaranteed to be accessed in the fastest possible way (conflict free). Side lengths which are not multiples of 16 may still access shared memory conflict free, however it is not guaranteed in every case.

The optimum size of the cross section varies with problem size and the order of accuracy required, however, is typically 8x16 or 16x16. Larger cross sections require too much memory while smaller cross sections are inefficient.

Having subdivided the solution space a thread block is assigned to each of the cuboidal domains. Every time step the thread blocks iterate in parallel through the solutions space spanning dimension using shared memory and registers to explicitly cache data required for the next iterations. Each thread in the thread block calculates the result for a single cell per iteration. This is shown diagrammatically in Figure 1. Shared memory is used to swap data between the threads of the thread block so as to minimise loading from global memory. If the limitations on shared/register memory availability were lifted this method would allow for each global memory location to be read from only once. Instead, each thread block must load data from neighbouring domains, decreasing efficiency.

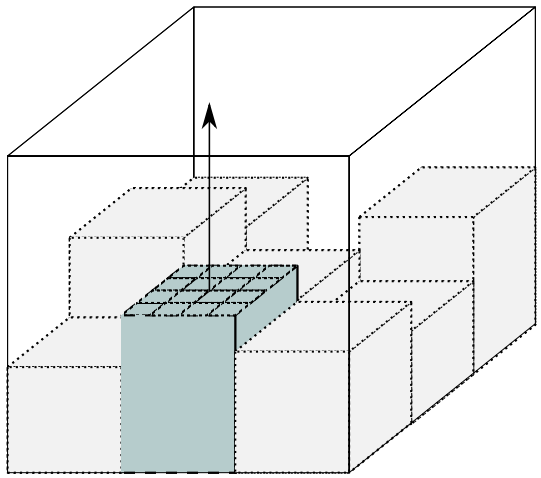


Figure 1: Thread blocks spanning the domain in two dimensions iterating in parallel along the third dimension.

A more naïve solution would be to simply assign each thread to perform the calculations for one grid cell. While this is a lot simpler (as it requires no shared memory programming) it would result in an order of magnitude increase in global memory requirements due to the lack of memory re-use. This would therefore be a lot slower.

After the time step is completed boundary data must be transferred between GPUs. Due to hardware limitations there is no way of directly transferring data between GPUs and so the data must be copied across the PCI-E bus to the host memory before being copied onto a different GPU. The maximum theoretical bandwidth of this transfer is 8 GB/s (an order of magnitude slower than GPU global memory). Fortunately, it is possible to copy data across the PCI-E bus while continuing calculations on the GPU by splitting the algorithm into two sections (one which requires the boundary information and one which doesn't). This effectively hides the memory transfer with only a small cost due to the splitting.

### *4.2. CPU*

The CPU implementation is much simpler than the GPU implementation. Each core iterates over a small part of the domain before transferring data between cores. The same asynchronous memory transfer masking technique as in the GPU method is used.

## 5. Results

The test case used to generate these results was the motion of a slotted sphere in a rotational velocity field. The grid was strongly scaled across many devices, i.e., total data processed remains constant. Extrapolation boundary conditions were used at each of the interfaces. It should be noted that while the velocity field was constant in time it was treated as a variable and no optimisation was based upon it being constant.

### *5.1. Architectural comparison*

Figure 2 shows arithmetic throughput of the two architectures using a $3^{rd}$-order HOUC scheme in single precision. As each architecture solves the same equations this can be used as a direct measure of performance.

Table 2 shows the equivalent processing power of multiple GPUs in terms of CPU cores. It is clear to see that a single GPU is capable of performing two orders of magnitude more work than a single CPU core when solving the level set equation. It is also noticeable that both architectures scale in performance in a close to linear manner.

| GPUs | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Equivalent Cores | 92 | 187 | 280 | 371 |

Table 2: Equivalent CPU cores for up to 4 GPUs.

### *5.2. Order of accuracy*

Figure 3 shows the arithmetic throughput of the two architectures using $3^{rd}$, $5^{th}$, $7^{th}$ and $9^{th}$-order HOUC schemes in single precision.

These results show that the higher-order schemes on the GPU show significantly better floating point performance than lower-order schemes. This is because the algorithm is quite efficient in terms of data re-use and hence can achieve a higher arithmetic intensity with higher-order schemes.

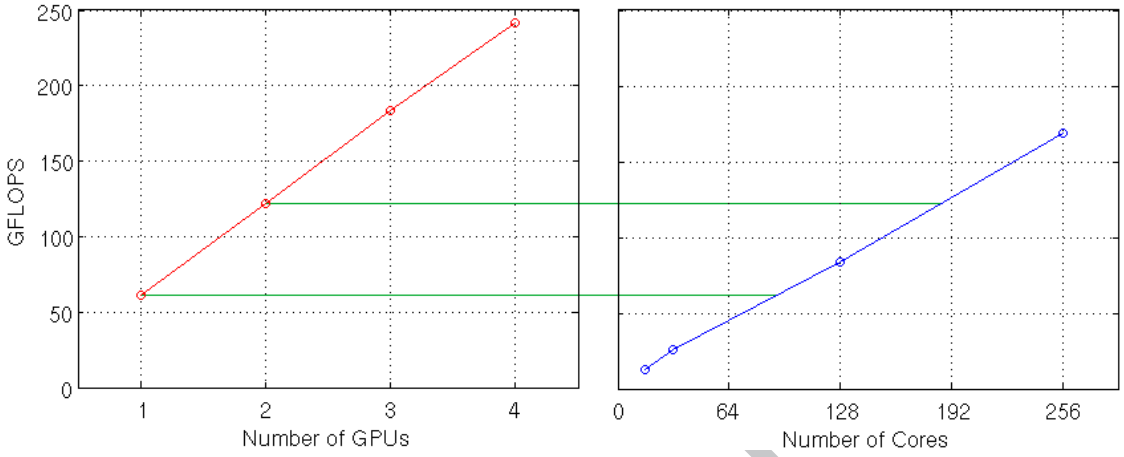



Figure 2: GFLOPS produced from a 3rd order HOUC scheme in single precision by up to 4 GPUs (left) and by up to 256 cores (right).

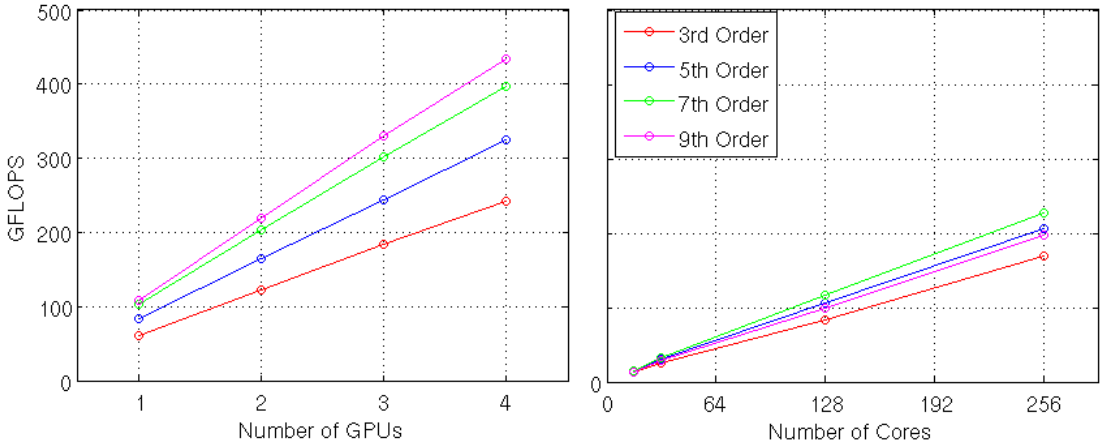



Figure 3: GFLOPS produced from 3rd, 5th, 7th and 9th-order HOUC schemes in single precision by up to 4 GPUs (left) and by up to 256 cores (right).

The results from the CPU are less conclusive. While a similar trend appears to be in place, the $9^{th}$-order scheme achieves lower throughput than the $5^{th}$ and $7^{th}$-order schemes. It is unclear as to the cause of this, however, it seems likely that it is due to the limited CPU cache size.

### *5.3. Precision*

The results presented thus far have been purely single precision and while this accuracy would be acceptable for some purposes many applications would require double precision. Figure 4 shows the arithmetic throughput of the two architectures using $3^{rd}$ and $9^{th}$ order HOUC schemes in single and double precision. Figure 5 shows the memory bandwidth required on the two architectures using the same configurations.

As is to be expected the double precision throughput on the GPU is significantly lower than the single precision throughput, however, it is a much greater proportion (30-50%) of the peak theoretical throughput. Comparatively, the CPU shows approximately half the throughput in double precision than in single precision. This suggests that the CPU method is bound by the memory bandwidth of the system. Comparison of the memory bandwidth achieved to that achieved on the same system by the STREAM [20] benchmark confirms this.

From these data we can also conclude that the GPU method is bound by neither solely by either of the two constraints (memory bandwidth or arithmetic throughput) of the GPU. Instead, it would appear that algorithm is bound by both at different stages of execution. This is possible despite the massively parallel nature of the GPU as each iteration requires two steps and after each a block-wide synchronization occurs. The first step mainly comprises memory transfer while the second mainly comprises arithmetic operations. As a multiprocessor typically executes only two blocks concurrently, the system is prone to bottlenecking in either step.

The proportion of time that the GPU spends bound by each limit varies with precision and order of accuracy. To illustrate this: although in both precisions the $9^{th}$-order scheme shows improvements over the $3^{rd}$-order scheme, in double precision this improvement is nowhere near as significant. This is due to the inferior double precision performance of the GPU leading to a greater proportion of the execution time spent doing arithmetic operations. This effect is not seen so prominently in single precision as increasing the order of accuracy does not bring the arithmetic throughput to such a large fraction of the peak.

## 6. Conclusions

GPUs have be used to greatly accelerate the computation of the level set equation on a block-structured domain. The performance increase seen was two orders of magnitude in both single and double precision and was found to be much greater at higher orders of accuracy due to the increased arithmetic intensity of the problem. This is significant as it allows for higher-order schemes to be implemented without as much concern regarding their computational expense.

While the GPU showed greater performance in single precision its double precision performance was not nearly as poor as might be expected given the high ratio of single to double precision floating point capacity of the GPU. This is because the performance was greatly restricted by available memory bandwidth. As the arithmetic intensity was increased the difference between single and double precision became more apparent and it is likely that this trend would continue if greater arithmetic intensities could be obtained.

The improvements seen here far outweigh the price difference between the two pieces of hardware in all cases, although some time had to be spent to design and optimise an algorithm for the GPU, the expense of which is less clear.

## References

[1] NVIDIA Tesla Technical Specifications, `http://www.nvidia.co.uk/page/tesla_supercomputer_tech_specs.html` (August 2010).

[2] `http://www.intel.com/support/processors/sb/cs-023143.htm` (August 2010).

[3] `http://ark.intel.com/Product.aspx?id=37153&processor=i7-975&spec-codes=SLBEQ` (August 2010).

[4] J. Lengyel, M. Reichert, B. Donald, D. Greenberg, *Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware*, Proceedings of SIGGRAPH 1990, 327-335.

[5] HH. Hsie, WK. Tai, *A Simple GPU-Based Approach for 3D Voronoi Diagram Construction and Visualization, Simulation Modelling Practice and Theory*, 13 (2005) 681-692.

[6] M. Schatz, C. Trapnell, A. Delcher, A. Varshney, *High-Throughput Sequence Alignment Using Graphics Processing Units*, BMC Bioinformatics, 8 (2007) 474.

[7] R. Nieuwpoort, J. Romein, *Using Many-Core Hardware to Correlate Radio Astronomy Signals*, Proceedings of the 23rd International Conference on Supercomputing.

[8] T. Preisa, P. Virnaua, W. Paula, J. Schneider, *GPU Accelerated Monte Carlo Simulation of the 2D and 3D Ising Model*, Journal of Computational Physics, 228 (2009) 4468-4477.

[9] A. Antoniou, K. Karantasis, E. Polychronopoulos, J. Ekaterinaris, *Acceleration of a Finite-Difference WENO Scheme for Large-Scale Simulations on Many-Core Architectures*, 48th AIAA Aerospace Sciences Meeting, 2010.

[10] J. Cohan, M. Molemaker, *A Fast Double Precision CFD Code using CUDA*, Proceedings of Parallel CFD 2009.

[11] J. Thuibault, I. Senocak, *CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows*, Proceedings of 47th AIAA Aerospace Sciences Meeting, 2009.

[12] T. Brandvik, G. Pullan, *An Accelerated 3D Navier-Stokes Solver for Flows in Turbomachines*, ASME Turbo Expo, Orlando, FL, June 2009.

[13] E. Elsen, P. LeGresleya, E. Darve, *Large Calculation of the Flow Over a Hypersonic Vehicle Using a GPU*, Journal of Computational Physics, 227 (2008) 10148-10161.

[14] A. Corrigan, F. Camelli, R. Löhner, J. Wallin, *Running Unstructured Grid Based CFD Solvers on Modern Graphics Hardware*, AIAA Paper 2009-4001, 19th AIAA Computational Fluid Dynamics, June 2009.

[15] S. Osher, J. Sethian, *Fronts Propagating with Curvature-Dependent speed: Algorithms Based on Hamilton-Jacobi Formulations*, Journal of Computational Physics, 79 (1988) 12-49.

[16] R.R. Nourgaliev, T.G. Theofanous, *High-Fidelity Interface Tracking in Compressible Flows: Unlimited Anchored Adaptive Level Set*, Journal of Computational Physics, 224 (2007) 836-866.

[17] D. Drikakis, W. Rider, *High-Resolution Methods for Incompressible and Low-Speed Flows*, Springer, 2005.

[18] NVIDIA CUDA Programming Guide Version 2.3, Page 2.

[19] P. Micikevicius, *3D finite difference computation on GPUs using CUDA*, Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009, 79-84.

[20] STREAM Benchmark, `http://www.cs.virginia.edu/stream/ref.html` (August 2010)

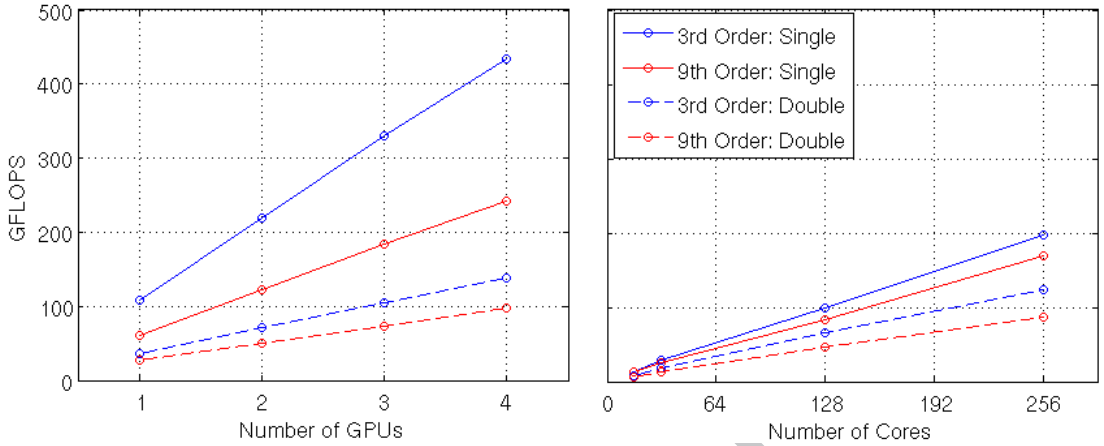



Figure 4: GFLOPS produced from 3rd and 9th order HOUC schemes in single and double precision by up to 4 GPUs (left) and by up to 256 cores (right).

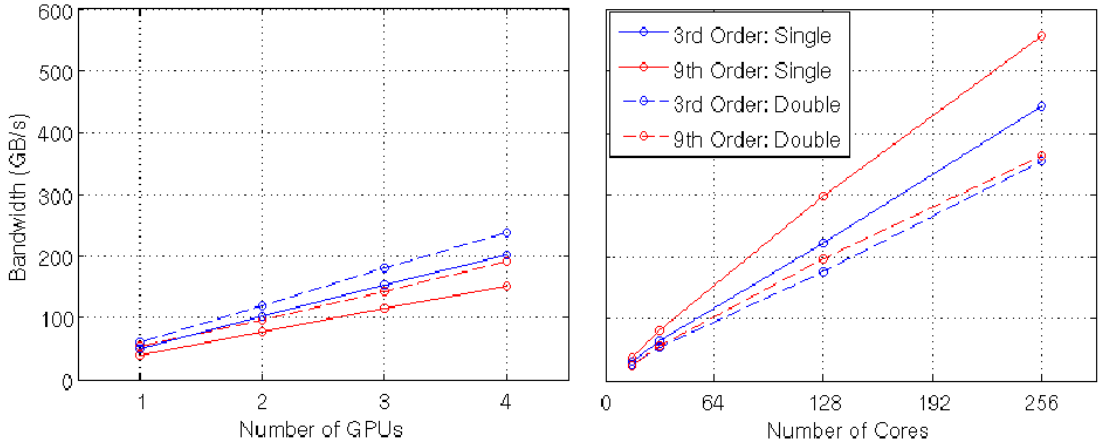



Figure 5: Memory bandwidth for 3rd and 9th order HOUC schemes in single and double precision by up to 4 GPUs (left) and by up to 256 cores (right).