



Porting NAHJ to CUDA

Evghenii Gaburov^a

^aSURFsara, Science Park 140, 1098XG Amsterdam, the Netherlands

Abstract

This white-paper reports on an enabling effort that involves porting a legacy 2D fluid dynamics Fortran code to NVIDIA GPUs. Given the complexity of both code and underlying (custom) numerical method, the natural choice was to use NVIDIA CUDA C to achieve the best possible performance. We achieved over 4.5x speed-up on a single K20 compared to the original code executed on a dual-socket E5-2687W.¹

Project ID: NAHJ

1. Introduction

Research into sonic and supersonic gas jets is important from the point of view of both fundamental fluid dynamics and engineering applications, such as engines and combustors. A comprehensive understanding of the global structure and the dynamics of these jets is important both for the developments of new strategies to improve the performance and efficiency of direct-injection engines. It can also be used to verify the feasibility of new propulsion systems and for safety issues. In this enabling work we used a code which is aimed to study the fluid dynamic behaviour of underexpanded hydrogen jets for ground applications. The code employs custom explicit fluid solver on two-dimensional axial symmetric grid [1]. The solver is able to take into account real gas effects by employing either Van der Waals or Redlich-Kwong equation of state.

Fluid dynamics codes usually have large degree of parallelism that can be well mapped to massively parallel architectures. In particular, the same set of operations, stencil-type or otherwise, are applied to the bulk of the grid, except domain boundaries, and these operations can be mapped to both SIMD- and thread-parallel processors, such as NVIDIA GPUs. In this report, we describe our enabling effort to port the original code to NVIDIA GPUs.

This white paper is structured as follows. In section Section2. we describe the code structure, and in Section3. we describe our porting efforts. Finally, we present the results in Section4..

2. Code description

The original programme is written in a mixture of Fortran 77 and Fortran 90, and is parallelized for execution on distributed memory systems via MPI with a single thread per process. According to the DEC18 proposal, the programme scales well up to 40 MPI processes for the problems of interest, however the application performance is sub-optimal which leaves plenty of room for programme optimizations via memory-access optimization and SIMD-vectorization. We have attempted this step but due to combination of programme complexity and immaturity of Intel compilers to generate optimal vectorized code we were unable to achieve substantial performance improvements in a portable manner, and this left NVIDIA GPUs as the only option that is capable of delivering superior performance. Unfortunately, the parallelization strategy employed in the programme is not suitable for GPUs, and therefore we decided to profile the programme to identify major hotspots in order to redesign parallel framework for multi-threaded execution.

In Fig. 1 we show a screenshot from Intel VTune amplifier which we used to profile the code. We compiled the code with Intel Fortran Compiler 13 using `-O3 -xavx -ipo` flags in order to obtain hotspot information for a production binary. The hotspot information revealed that $\sim 80\%$ of the runtime is spent in four subroutines, which we call the big 4: `filtvd`, `eprop`, `eulbl` and `pderiv`. The remaining runtime is spread among about a dozen of remaining functions. This profiling data revealed that, due to Amdahl's law, it requires substantial porting effort in order to achieve more than 5x speed-up. Namely, both the big 4 and the dozen of lightweight functions will have to be ported to CUDA.

¹The original code was compiled with Intel Fortran 13.0.1 compiler with `"-O3 -xavx -ipo"` flags, and Intel MPI 4.0.1. The runs were carried out on the EURORA system @Cineca.

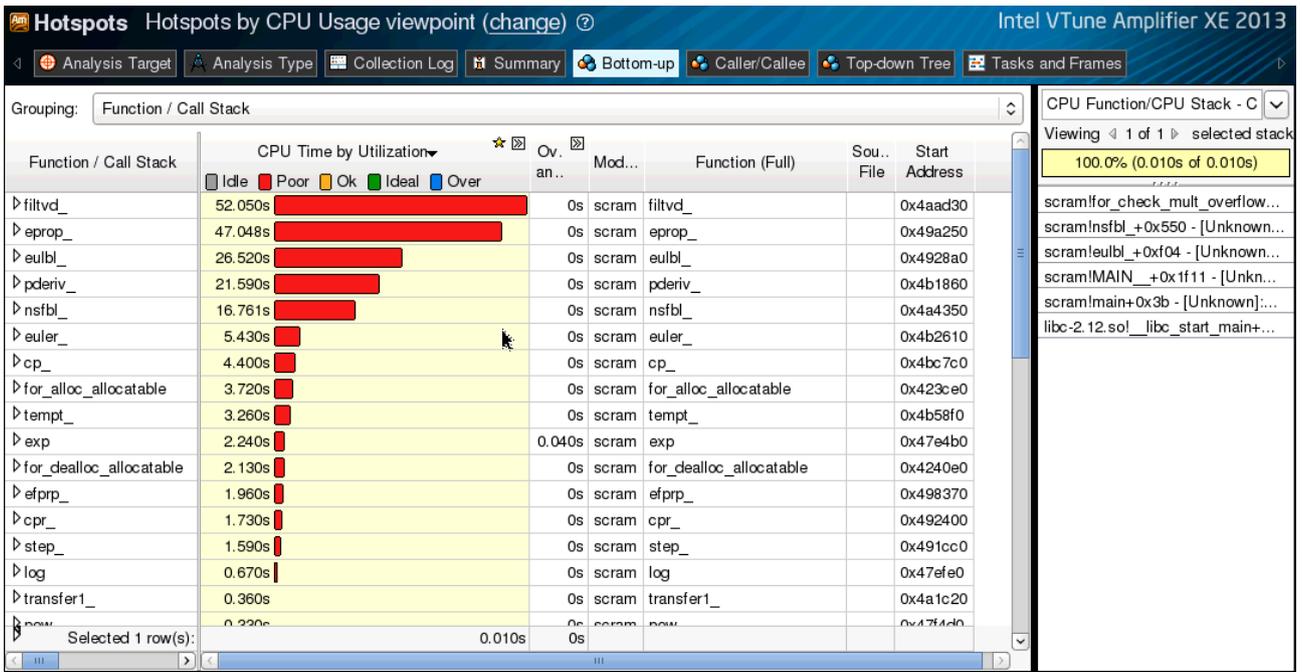


Fig. 1. Screenshot of Intel VTune Amplifier XE 2013 displaying profiling information.

3. Porting the code to CUDA

In Section 2. we identified candidate subroutines that require CUDA equivalents. Porting a code from Fortran to CUDA is a non-trivial task which prone to bugs. This is further complicated by separate memory spaces of CPU and GPU, which requires manual host↔device data movements.

Our attempt to simplify the porting process is predicated on minimal code changes while migrating hotspots from Fortran to CUDA C++. In a nutshell, the main iteration loop of the programme has the following form:

```

1 ... ! prologue
2 do iter=0,niter
3   ! this subroutine calls hotspots
4   call compute_kernels
5   ! every "nout" steps we conduct an intermediate analysis
6   if (mod(iter,nout).eq.0) then
7     call do_intermediate_analysis
8   end if
9 end do
10 ... ! epilogue

```

List 1. Main iteration loop

where `compute_kernels` subroutine calls all the hotspot identified during profiling step. Every `nout` iterations, the code carries out an analysis step to print some intermediate data, which helps the researchers to assess correctness of the simulation.

Our aim here is to port *all* functions inside `compute_kernels` to CUDA. For this purpose we must make sure that all necessary data is copied from the host to the device. However, if this is accomplished every-time step, the overall programme performance becomes limited by PCIe bandwidth². The only way to eliminate this bottleneck and achieve impressive speed-ups, is to communicate data between the host and the device only when needed. In particular, it is clear from the List.1 that we only need to send data to the device just before `do`-loop, and copy it back only inside the `if`-statement and at the end of the loop. Schematically, the modified code will have the following form:

```

1 call device_malloc
2 ... ! prologue
3 call copy_data_to_device
4 do iter=0,niter
5   ! this subroutine calls CUDA host-side code, which itself calls device kernels
6   call device_kernels
7   ! every "nout" steps we copy data from the device to the host

```

²Currently, Tesla K20 only support PCIe gen2.

```

8  if (mod(iter,nout).eq.0) then
9    call copy_data_from_device
10   call do_intermediate_analysis
11  end if
12 end do
13 ! we copy data at the end of simulation back to the host
14 call copy_data_from_device
15 ... ! epilogue
16 call device_free

```

List 2. PCIe transfer optimized loop

here `device_malloc` and `device_free` are CUDA subroutines that deal with device memory management, `copy_data_*` are CUDA subroutines that move data between the host and the device, and `device_kernels` are CUDA ports of `compute_kernels`. For details, we refer the reader to the source code that can be obtained upon request.

3.1. Multidimensional arrays

The big 4 are the subroutines that operate on two- and three-dimensional arrays via a basic nested loop:

```

1  do j = 2, jll
2    do i = 1, ill
3      ... ! some loop code on data1(i,j)
4      do k = 1, nk
5        ... ! some code on data2(i,j,k)
6      end do
7      ... ! write results to results(i,j)
8    end do
9  end do

```

List 3. Typical hotspot loop in the Fortran code

There are multitude of ways in which multi-dimensional arrays can be created and/or addressed in C-type languages; none of these are compatible with Fortran syntax out-of-the box. However, given the maturity of NVIDIA CUDA C++ compiler, which we refer to as `nvcc`, we decided to provide a class that emulates Fortran array notation. The merit of this approach is debatable, but it *does* noticeably streamline the porting process by allowing to copy-paste bulk of the code from Fortran into C++ verbatim, which in turn minimizes introduction of new bugs. To accomplish this we create a wrapper class for Fortran-type array notation,. Here is an example of a class for a three-dimensional array:

```

1  template<typename real>
2  class FArray3D {
3  private:
4    real *data;
5    int nx, ny, nz;
6  public:
7    ...
8    /* accessor methods */
9    __device__ __host__ real& operator()(const int i, const int j, const int k)
10     { return data[k*(ny + j)*nx + i]; }
11    __device__ __host__ const real& operator()(const int i, const int j, const int k)
12     const
13     { return data[k*(ny + j)*nx + i]; }
13 };

```

List 4. Templated C++ class to access multi-dimensional array in CUDA via Fortran notation

The class for two-dimensional arrays can be declared in a similar manner. The `nvcc` compiler does an excellent job at inlining these accessor-methods and optimizing out repeated integer arithmetics in address computations, therefore producing near-optimal code.

3.2. Work distribution in CUDA kernel

The CUDA programming model requires a programmer to write a programme that is executed by each of the many concurrent threads. This effectively shifts responsibility of correct code execution on multiple threads to the programmer, and lets the compiler to focus on producing quality code from a high-level language. This implies that nested loops in List. 3 must be adapted for massively multi-threaded execution.

The parallelization of the programme³ can be viewed as a multi-dimensional optimization problem. Therefore, the best results are obtained with heuristic approach which accounts for the problem-specific factors. In particular, since it is known that $j_{11} \sim i_{11} \gg nk$ in List. 3, therefore it is best to assign the code inside the double nested loop (lines 3-7) to a single thread. In the case of the both thread- and SIMD-parallel processors, such as XeonPhi, the *i*-loop must also be SIMD-vectorized, as a result the lines 3-7 will be executed not only by each thread but also by each SIMD lane.

The next step is to design optimal work distribution for this double nested loop under the assumption that *j*₁₁ is smaller than then maximal number of threads the processor can concurrently execute. Clearly, the optimal approach is a two-dimensional tiled parallelization over both *j*- and *i*-loops, which reduces surface-to-volume ratio. As a result, the CUDA code for the content of the nested loop will have the following form:

```

1  __global__ void foo() {
2  const int i = blockIdx.x*blockDim.x + threadIdx.x;
3  const int j = blockIdx.y*blockDim.y + threadIdx.y;
4  ... /* some code on data1(i,j), as well as variable & local array defintions */
5  #pragma unroll
6  for (int k = 0; k < NK; k++)
7  { ... /* some code on data2(i,j,k) */ }
8  ... /* write results to result(i,j) */
9  }

```

List 5. CUDA kernel skeleton to process 2D data

The actual computations of *i*- and *j*- indexes is somewhat more complex due to the need to account for the halo around zones in each of the thread-blocks which are processed in parallel in an undefined order and are unable to communicate with each other. For details we refer the reader to the source code.

3.3. Final touches

Finally, the CUDA port can be compiled either in single- or double-precision mode. In hindsight, the profiling revealed that kernel performance is limited by the instruction latency, as a result there is only marginal performance benefit of using single precision but it comes with high risk of producing erroneous results. Furthermore, we found that *fma*-optimization must be disabled, otherwise the algorithm exhibits unstable behaviour.

It is our conviction that the algorithm can be redesigned to make it instruction throughput, rather than latency, bound, and be less sensitive to *fma*-operations. This can significantly boost programme performance, but it is a formidable task which is beyond the scope of this enabling work.

4. Results

The results of our efforts are presented in Fig. 2. Here, the horizontal axis shows the problem size, in terms of number of grid-cells in *x*- dimension times number of grid-cells in *y*-direction. The amount of work scales as a product of these two numbers. The vertical axis shows runtime of the test case in seconds. It can be seen from the figure that for large enough problems the K20 substantial outperforms dual-socket E5-2687W CPUs.

In absolute terms, the big 4 CUDA kernels crunch data at ~ 50 to ~ 150 Gflops in double precision, which is roughly 10% of the K20 peak double precision performance. As we mentioned in previous section, the main limitation of the code is instruction latency, and this is caused by complex big 4 kernels that allocate too much resources per CUDA thread. As a result, the device is unable to keep sufficient number of threads on a fly in order to completely hide the instruction latency. We believe it is possible to overcome this difficulty by redesigning algorithms, but this work is beyond the scope of this project.

Acknowledgements

This work was financially supported by the PRACE-2IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493. The work was achieved using the PRACE Research Infrastructure resources at SURFsara and Cineca.

References

1. Grasso, F. & Maggi, V., 1995, AIAA Journal, 33 (1), 56-62.

³Here we also include distributed, multi-threaded and SIMD parallelism of the programme.

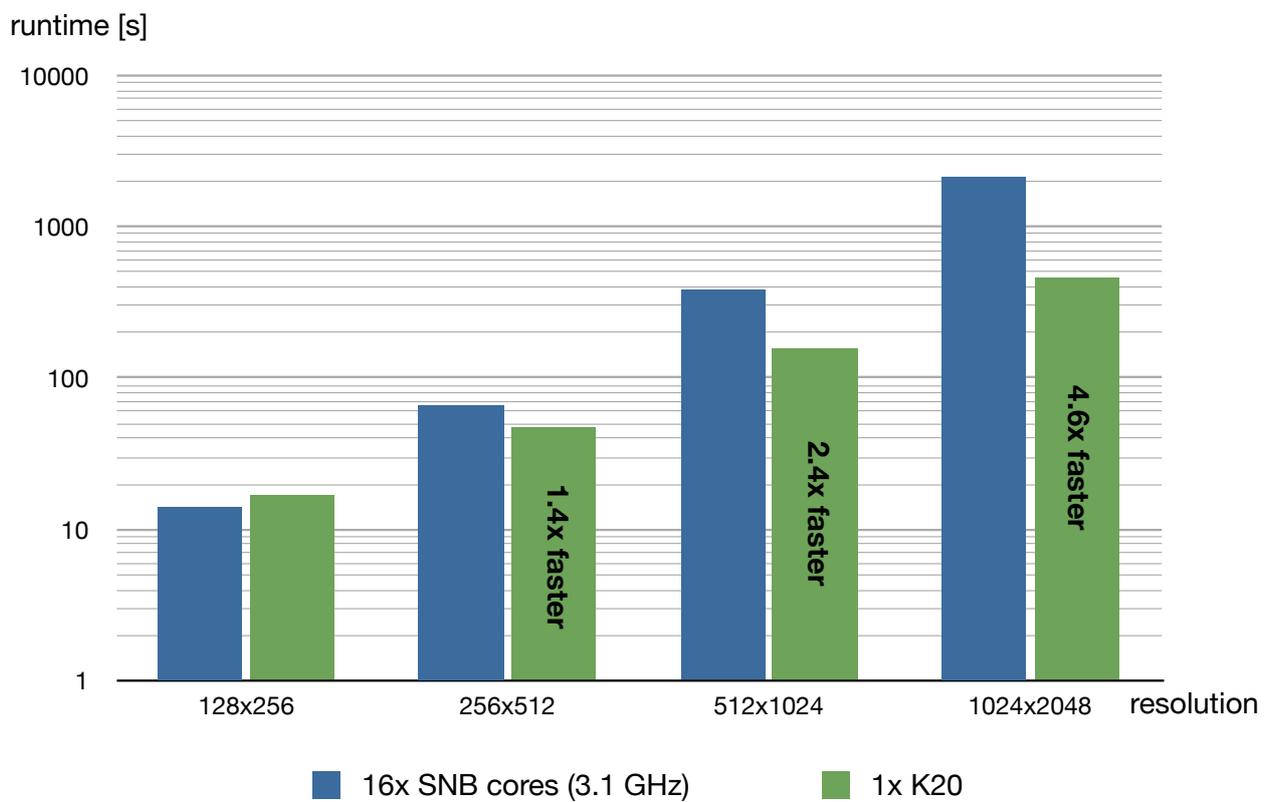


Fig. 2. Runtime in seconds of the original code on dual-socket (16 cores) Sandy Bridge E5-2687W CPU, and the CUDA port on a single K20s as a function of the problem size (horizontal axis).