Towards a Unified CPU–GPU code hybridization: A GPU Based Optimization Strategy Efficient on Other Modern Architectures

Ludomir Oteski, Guillaume Colin de Verdière, Sylvain Contassot-Vivier, Stephane Vialle, Juliet Ryan

Towards a Unified CPU–GPU code hybridization: A GPU Based Optimization Strategy Efficient on Other Modern Architectures

Ludomir OTESKI [a,1], Guillaume COLIN DE VERDIÈRE [b],
Sylvain CONTASSOT-VIVIER [c], Stéphane VIALLE [d,e], Juliet RYAN [a]
[a] *ONERA, 91123 Palaiseau, France*
[b] *CEA, DAM, DIF, F-91297 Arpajon, France*
[c] *LORIA - UMR 7503, Université de Lorraine, 54506 Vandoeuvre-lès-Nancy, France*
[d] *UMI 2958 - GT-CNRS, CentraleSupelec, Université Paris-Saclay, 57070 Metz, France*
[e] *LRI - UMR 8623, 91190 Gif-sur-Yvette, France*

Abstract. In this paper, we suggest a different methodology to shorten the code optimization development time while getting a unified code with good performance on different targeted devices. In the scope of this study, experiments are illustrated on a Discontinuous Galerkin code applied to Computational Fluid Dynamics. Tests are performed on CPUs, KNL Xeon-Phi and GPUs where performance comparison confirms that the GPU optimization guideline leads to efficient versions on CPU and Xeon-Phi for this kind of scientific applications. Based on these results, we finally suggest a methodology to end-up with an efficient hybridized CPU–GPU implementation.

Keywords. Parallel code optimization, vectorization, hybrid code, GPU, Xeon-Phi.

1. Introduction

The recent development of Graphic Processing Units (GPU) and Xeon-Phi (X-Phi) accelerators now offers a large panel of solutions for an extensive increase of computing performance without requiring heavy structures of supercomputing centres. However, the adaptation of existing codes on accelerators may remain tricky. For example, the *SeisSol* [4] software for seismic simulations based on a Discontinuous Galerkin (DG) numerical scheme, has been severely optimized for the Xeon-Phi Knights Landing (KNL) architecture. Since 2013, complex optimization works were made to optimize DG computations on GPU architecture: [5] introduced an auto-tuning mechanism to improve an already optimized generic GPU implementation. In 2016, a semi-Lagrangian DG method applied to plasma physics applications was developed and implemented on Xeon proces-

[1]Corresponding Author: Book Production Manager, IOS Press, Nieuwe Hemweg 6B, 1013 BG Amsterdam, The Netherlands; E-mail:bookproduction@iospress.nl.

sor, Xeon-Phi KNC and GPU co-processors [2]. However, all these developments have been designed and optimized for one specific type of device (CPU, Xeon-Phi or GPU).

The application used in this study consists in a 2^{nd} order DG scheme designed to solve the two-dimensional compressible Navier-Stokes equations. In High Performance Computing, DG schemes are very well suited to efficient parallelization since the resolution of partial differential equations are independent for each cell of the numerical mesh and thus lead to a high degree of parallelism. Anyone confronted to a computationally intensive sequential code, will try to derive a parallel version from the sequential one. In such context, the classical development approach consists in adding successive levels of parallelism. Usually, a multi-threaded and vectorized version is developed as a first step, then comes a GPU version. Both CPU and GPU codes are optimized apart from each other and eventually merged in a single program. Finally, a distant communication layer is added in order to make the program usable on clusters and distant computers.

In this paper, we suggest a different methodology to shorten the code optimization development time while getting a unified code with good performances on different targeted devices.

The mathematical statement of the problem is given in section 2. In section 3 the approach for a unified development in presented. Section 4 details the optimization guideline common to the different device architectures. Section 5 presents multiple experiments, validating the approach on various devices. Based on the results presented in sections 4 and 5, a hybridization technique for CPU–GPU code development is introduced in section 6.

2. Use case: Discontinuous Galerkin

The two dimensional time-dependent compressible Navier-Stokes equations are expressed as:

$$\partial_t \mathscr{W} + \nabla \cdot \mathbf{F}(\mathscr{W}) - \nabla \cdot \mathbf{D}(\mathscr{W}, \nabla \mathscr{W}) = \mathbf{0}, \tag{1}$$

where $\mathscr{W} = (\rho,\ \rho\mathbf{U},\ \rho\mathrm{E})$ is the conservation variable vector with the following notations: ρ is the density, $\mathbf{U}$ is the 2D velocity field, and E stands for the total energy. $\mathbf{F} = \mathbf{F}(\mathscr{W})$ and $\mathbf{D} = \mathbf{D}(\mathscr{W}, \nabla \mathscr{W})$ are the convective and diffusive fluxes, respectively. Boundary conditions are set as periodic in each directions for each equation and the test case is that of a viscous propagating vortex (see Fig 1).

Eqs. (1) with associated boundary conditions are solved in a domain Ω discretized by either a Cartesian or an unstructured triangular grid $\mathscr{T}_h = \bigcup \Omega_i$. The associated function space V_h is

$$V_h = \{\phi \in L^2(\Omega) \mid \phi/\Omega_i \in P_k\}, \tag{2}$$

where P_k is the space of polynomials of degree k ($k = 2$ in this study).

The Discontinuous Galerkin formulation is based on a weak formulation after a first integration by parts: find $\mathscr{W}_h$ in $(V_h)^4$ such that for any cell Ω_i in $\mathscr{T}_h$,

$$\forall \phi \in V_h, \quad \int_{\Omega_i} \partial_t \mathscr{W}_h\, \phi\, dx = \int_{\Gamma_i} (\mathbf{F}_h - \mathbf{D}_h)\, \phi\, d\gamma - \int_{\Omega_i} (\mathbf{F}_h - \mathbf{D}_h) \nabla \phi\, dx. \tag{3}$$

where $\Gamma_i = \partial \Omega_i$ is the cell boundary.

Here, the numerical fluxes $\mathbf{F}_h$, $\mathbf{D}_h$ and $\mathscr{W}_h$ are numerical approximations of $\mathbf{F}$, $\mathbf{D}$ and $\mathscr{W}$. The inviscid fluxes $\mathbf{F}$ are determined using the HLLC scheme [9], the viscous fluxes $\mathbf{D}$ are computed with the Elastoplast Discontinuous Galerkin method [1]. Time is discretized with Shu-Osher's explicit Total Variation Diminishing 3^{rd} order (3 steps) Runge Kutta time scheme [3].

The time step is computed as $\delta t = C \times \min_i \left(\frac{min(dx_i, dy_i)}{|\mathbf{U}_i| + c_i} \right)$ where $C = 0.15$ is the CFL number, i is the cell number, dx_i, dy_i the local cell dimensions, $\mathbf{U}_i$ the local velocity and c_i the local speed of sound. This scheme is fully described in [1] and references therein.

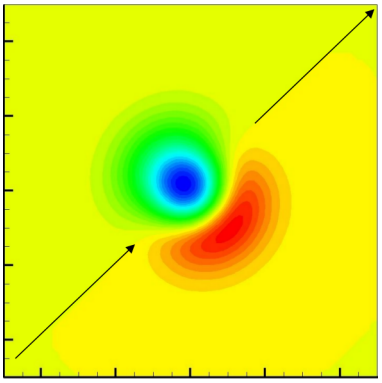

Figure 1. Instantaneous density field of the flow. The vortex moves across the periodic numerical domain according to the arrows.

All integrals are computed using Gaussian quadrature rules. We designed the study for 2^{nd} order polynomials on a Cartesian grid. Therefore, 3 Gauss points are chosen in each direction, leading to 3 Gauss points on each segment of Γ_i, and 9 Gauss points for integrals on Ω_i. Connectivity between cells is restricted to cells with a common edge.

From a computational point of view, each polynomial coefficient needs to be propagated in time in order to reconstruct the total solution. This leads to a total storage space of $N_{coefs} \times N_{eq} \times N_{cells}$ per arrays of variables, with N_{coefs} the number of polynomial coefficients (6 for 2-dimensional second order polynomials), N_{eq} the number of equations (4 in our case) and $N_{cells} = N_x \times N_y$ the number of cells in the x and y directions.

For this application, 3 double precision arrays of variables were used (one to store the system current state, one for the previous time-step and one to store time-derivatives). Thus, memory cost can be approximated by $8 \times 3 \times (6 \times 4 \times N_{cells})$ Bytes.

3. Unified development approach

Even though GPUs and CPUs are different in their conception, both share the similar idea of increasing performance through vectorization (vectors for CPUs, warps for GPUs). Therefore, any optimization which is not fully dependent on the GPU architecture (such as texture caches) should also be relevant for vectorial CPUs. Since GPU compilers offer accurate information in terms of resources used per kernel, the effect of every optimization can be carefully monitored. This is why, in order to take advantage of the similarities between GPUs and CPUs, we developed a CPU/X-Phi code by following the same optimization guideline applied to a GPU version of the application. More precisely, once an optimization is identified as efficient for the GPU code, it is reported in the CPU code. Basically, the idea is to copy/paste the content of GPU kernels into the vectorized part of the CPU code, as sketched in Table 1. In this example, the 2D grid of threads is translated into two nested loops in the CPU version. The outer one is multithreaded (lines 4-5) and the inner one is vectorized (lines 8-9).

The GPU developments could be done with any GPU API providing a sufficiently fine control of kernel programming and memory management. The present study is re-

2D CUDA-GPU code

```
void __global__ function(...)
{
  //Thread index in x-direction
  int tidx = ...;
  //Thread index in y-direction
  int tidy = ...;
  //Logical condition
  //to make computations
  if(tidx<Nx && tidy<Ny)
  {
    ... //Copy to CPU code
  }
}

```

2D CPU code (vectorized with icc)

```
void function(...)
{
  //Loop on y-direction
  #pragma omp for
  for(tidy=0; tidy<Ny; tidy++)
  {
  //Loop on x-direction
    #pragma simd
    for(tidx=0; tidx<Nx; tidx++)
    {
      [...] //Insert from GPU code
    }
  }
}

```

Table 1. Sketch of the experimental protocol. Vectorization of the CPU code is ensured by `simd` pragma.

stricted to the CUDA API [6] as it provides the required low level of code control, yet we believe that similar results could be achieved with other programming models such as OpenACC[2] and OpenCL[3].

4. Optimization guideline

4.1. Compilation setup

Each optimization suggested in this study is tested on a K20Xm for the GPU code and on an Intel E5–1650v3 (1 socket of 6 cores, no hyperthreading) for the CPU code. The compilation is done with the Intel 2017 icc compiler for the CPU code and the CUDA8.0 nvcc compiler for the GPU part. Both codes are compiled with `-O3` optimization option with the following additional options for icc:

- `-march=native` : add specific optimizations considering the machine used for the compilation,
- `-fma` : activates fused multiply-add operations for processors that support it,
- `-align` : activates the memory alignment of data according to the target architecture.

4.2. Optimization steps and performance measurements

Results of our experiments are reported in Table 2. Performances are expressed in cell updates per second (*cus*) as:

$$P = (N_{cells} \times n)/t \tag{4}$$

where N_{cells} is the number of cells and t the time (in seconds) required to make n time-steps of the system.

[2] `http://www.openacc.org/About_OpenACC`
[3] `https://www.khronos.org/opencl`

Code versions	CPU (6 th.) E5- 1650 v3	Speedup *vs* seq. CPU	Progres- sive speedup	GPU K20Xm	Speedup *vs* seq. CPU	Progres- sive speedup
Sequential CPU code (*1 thread*)	$8.35 \times 10^4 cus$	1.00	–	–	–	–
Initial CPU version (*OpenMP + simd pragmas*)	$5.91 \times 10^5 cus$	7.06	7.06	–	–	–
Initial GPU version (*CUDA*)	–	–	–	$1.44 \times 10^6 cus$	17.21	17.21
Reduced accesses to distant memories	$8.05 \times 10^5 cus$	9.63	1.36	$5.71 \times 10^6 cus$	68.36	3.97
Merged funcs with same mem. patterns	$1.09 \times 10^6 cus$	13.05	1.37	$10.0 \times 10^6 cus$	119.62	1.85
Simplification of computations	$2.01 \times 10^6 cus$	24.05	1.82	$12.1 \times 10^6 cus$	145.00	1.21
CPU Only: Tiling (*cache optim.*)	$2.48 \times 10^6 cus$	29.72	1.23	–	–	–
Data alignment	$2.86 \times 10^6 cus$	34.25	1.19	$12.5 \times 10^6 cus$	149.53	1.03
CPU Only: Tuning on vectorization	$2.96 \times 10^6 cus$	35.44	1.03	–	–	–
CPU Only: MPI + OpenMP (*3 proc.× 2 th.*)	$3.05 \times 10^6 cus$	36.52	1.03	–	–	–

Table 2. Performance results of the different versions of code. The test case is made on a 2001×2001 mesh and is integrated over 100 time-steps. Performances (in cell updates per seconds [*cus*]) are evaluated from eq. (4).

Except for a few optimizations (such as tiling algorithms) which are specific to CPU developments, four optimization steps have been identified as crucial. These are summed up in the list below, but it is worth noticing that some of them induce important code modifications.

1. *Reduce accesses to distant memories* which is achieved by transferring redundant accesses to global/DRAM memory into registers. While on the GPU this strategy avoids unnecessary accesses to global memory, for the CPU it consists in improving data cache locality. When completed, this step increases performance on GPU by a factor of 3.97, and on CPU by a factor of 1.36.

2. *Merge kernels which share the same memory patterns*. This optimization consists in limiting the number of accesses to distant memories by improving the re-use of distant variables. For the CPU this translates in an increase of cache re-use. According to Table 2, this step increases performance on the GPU by a factor of 1.75, and by a factor of 1.37 on the CPU.

3. *Simplify and factor computations* which consists in the suppression of non-essential variables as well as storing recurrent computations (such as multiplications by constants, constant square roots,...) in intermediate variables, and to con-

trol whether or not a static loop should be unrolled. Although this step is the most time-consuming in terms of development time, it increases GPU performance by a factor of 1.21, whereas it enhances CPU performance by a factor of 1.82.

4. *Align data*. From the GPU point of view, this step consists in aligning data on the size of a warp (32 elements). By doing so, we ensure coalescent memory accesses for each warp. For the CPU, this corresponds to a tiling algorithm (for cache fitting) with static tile sizes (for better vectorization). This induces two more nested loops in the CPU version of Table 1 (two for the tiles and two for the cells). On GPU, this optimization step slightly increases performance by a factor of 1.03 (for warp size data alignment). On CPU, the performance increase factor is 1.23 (for tiling) $\times 1.19$ (for static tile sizes) $= 1.46$.

4.3. Vectorization tuning

The approach described in sec. 3 and optimizations steps described above tend to create huge vectorized loops, which highly increases the register pressure and may not be suitable for CPU vectorization. Therefore, we adjusted the CPU vectorization by splitting SIMD loops into smaller loops and replaced every `#pragma simd` by `#pragma vector always` indications. According to [7], this instruction tells the compiler to perform auto-vectorization if the loop does not carry dependencies. By doing so, CPU performances have been increased by a factor 1.03.

4.4. Data locality improvement

Finally, we created a mixed OpenMP–MPI version of the CPU code in order to make sure that the memory locations of data used in one MPI process and its threads are close to their associated cores. The domain decomposition has been performed along the $y-$axis by using a ghost-cell technique to share boundaries of neighbouring domains between MPI processes. This optimization, well suited to NUMA architectures, increases the code performance by a factor of 1.03. In the end, the CPU (resp. GPU) application is around 36 (resp. 149) times faster than the original sequential code.

5. Experiments on various devices

5.1. Testbed

The final CPU code has been tested on a bi-socket of E5–2698v3 (with DDR4 memory) processors (2×16 cores), a bi-sockets of E5–2680v4 (DDR4) and a KNL 7250 in quadrant mode, with either DDR4 or MCDRAM, MCDRAM itself configured in flat mode (all data are stored in MCDRAM) or in cache mode (the MCDRAM acts as a L3 cache). The quadrant mode allows to manage the KNL in four distinct areas where internal data accesses are preferred to accesses between areas. The final GPU code has been tested on different Tesla devices: a K20Xm (DDR5), a K40 (DDR5), a P100 (HBM2) with a bandwidth of 540GB/s and another P100 (HBM2) with a bandwidth of 720GB/s. Each simulation has been made with active ECC.

In order to see the performance limit, the test case has been extended to a 2731×2731 mesh (this multiplies the problem size by 1.86, the total memory consumption of the application being around 5GB).

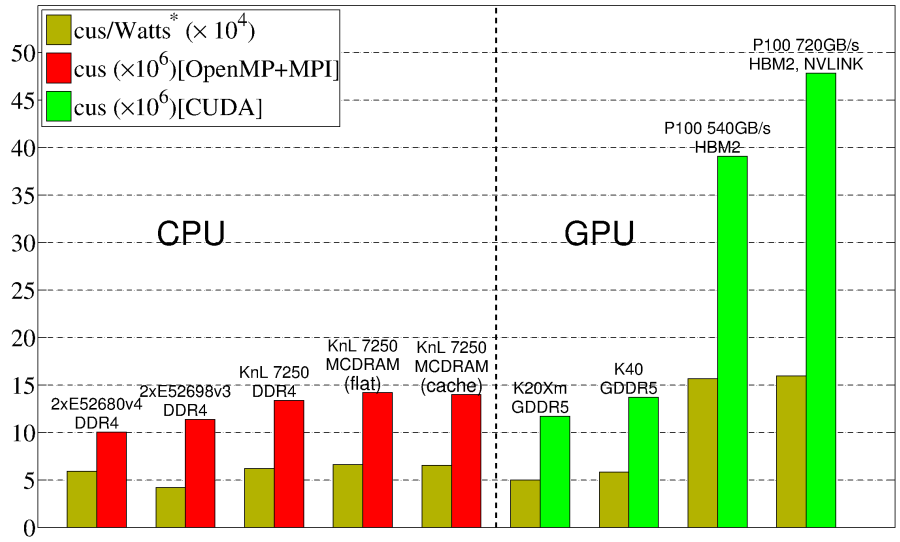

Figure 2. Performance for various devices. The energy efficiency is evaluated from the corresponding device documentation.

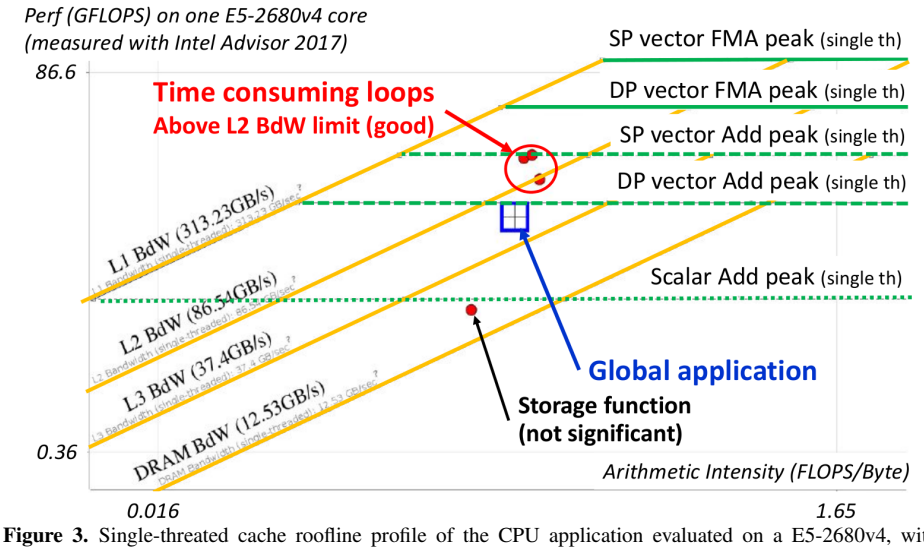

Figure 3. Single-threated cache roofline profile of the CPU application evaluated on a E5-2680v4, with Intel® Advisor (2017). BdW stands for Bandwidth. Only the loops (red) which are the most time-consuming ($>2\%$ of the application runtime) are shown for clarity. The dot between DRAM and L3 cache corresponds to the cost of saving variables at each Runge-Kutta step. The overall performances of the code is evaluated around 11 GFlops (blue square).

5.2. *Performance analysis*

Results presented in Fig. 2 show that the CPU version of the code running on the X-Phi KNL 7250 is competitive with the CUDA version on the GPU K40, and that the bi-socket E5-2698v3 is competitive with the K20Xm. The P100 (540GB/s) is around 2.64 times

faster than the KNL and this ratio becomes 3.29 for the P100 (720GB/s). And last comes the bi-socket E5-2680v4 which is almost as power efficient as the KNL.

Due to the cache optimization effort (*tiling* in Table 2), the use of the MCDRAM does not scale with the increase of available bandwidth. According to [8], MCDRAM has a bandwidth faster than 400GB/s while the DRAM used in the KNL 7250 stays around 90GB/s. Profiling the code with the Intel® Advisor (2017) cache roof-lines tool shows that most of data accesses are kept between L2 and L1 caches (see Fig. 3). Therefore, MCDRAM is not expected to be as useful as it could be as the L2 bandwidth is around five times faster (around 2TB/s as given by the Intel® Advisor (2017) profiler) than the bandwidth of MCDRAM. The size of the problem ($\sim$ 5GB) being less than the total amount of available MCDRAM ($\sim$ 16GB), all the data are kept in MCDRAM. Therefore, we do not see any impact on the mode used to configure MCDRAM (flat or cache).

The cache roof-lines provided by Advisor (Fig. 3) show that the CPU code could still be improved (by increasing L1 arithmetic intensity). However, such an optimization would require consequent code modifications which may not be suitable as it would create strong divergences between GPU and CPU codes.

6. Hybrid implementation

2D CUDA-GPU code (GPU.cu)

```
#define VSIZ 1
//Include the kernel
#include "kernel.h"

void __global__ gpu_function(
  double *__restrict__ val)
{
  //Thread indexes
  int tidx = ...;
  //Logical condition
  //to make computations
  if(tidx<Nx)
  {
    kernel<VSIZ>(val,tidx);
  }
}

```

Common vectorized kernel (kernel.h)

```
template<const int VSIZ>
//Conditional compilation
#ifdef DEFGPU
__device__
#endif
__inline void kernel(
  double *__restrict__ val,
  const int tidx)
{
  //Vectorized loop
  #pragma vector always
  #pragma unroll
  for(vec=0; vec<VSIZ; vec++)
  {
    ...
    val[VSIZ*tidx+vec] = ...;
  }
}
```

2D CPU code (CPU.cpp)

```
#define VSIZ 32
//Include the kernel
#include "kernel.h"

void cpu_function(
  double *__restrict__ val)
{
  //CPU loop
  for(int tidx=0;
      tidx<Nx;
      tidx+=VSIZ)
  {
    kernel<VSIZ>(val,tidx);
  }
}

```

Table 3. Programming model for CPU–GPU code hybridization.

Since CPUs and GPUs can benefit from the same optimizations (see sec. 3 and 4), it is possible to consider that both devices could use the same computational kernels. To do so, we used C++ templates in order to state the size VSIZ of the vector to be used by each device. Since GPU threads cannot be considered as vectorial units, VSIZ=1 for GPUs whereas VSIZ should be taken as a integer multiple of the vector size on CPUs. The statement indicating if the code of the kernel should be compiled for CPU or GPU is given by a compilation variable. This programming methodology is illustrated in Table 3 and the overall hybrid implementation in Fig. 4.

In this version of the code, we use only one thread per MPI process. So, one MPI process should be used for each physical core. Also, the first MPI process of each node is responsible for the management of the GPUs present on the node. If at least one GPU is present, this MPI process runs the GPU code while all the remaining MPI processes

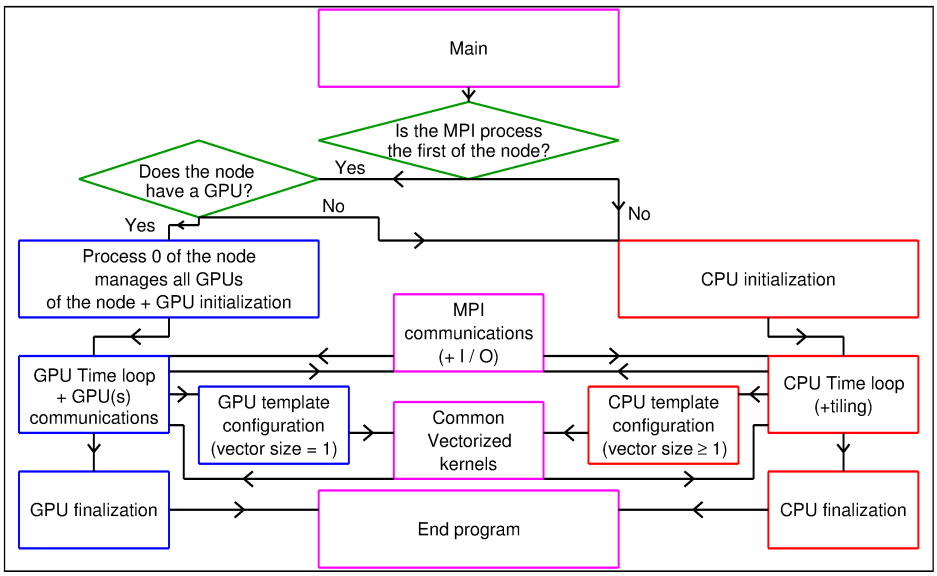

Figure 4. Sketch of the hybrid implementation. Colours indicates the GPU part (blue), the CPU part (red), branching conditions (green) and common CPU–GPU parts (pink).

of the node execute the CPU code. Otherwise, all MPI processes of the node execute the CPU code. By doing so, according to Fig. 2, if a GPU is present one can expect an ill-balanced workshare since GPUs have proved to be faster than a single CPU core. This is why, we implemented a pre-processing step which consists in the evaluation of the performances P_i for each MPI process running the application.

Let W_T be the total work to be done and W_i the work to be done by the MPI process i. After the pre-processing step, the work is statically distributed as:

$$W_i = \frac{P_i}{\sum_{j=0}^{n_p} P_j} W_T, \tag{5}$$

where n_p corresponds to the number of MPI processes.

This implementation has been tested on a two-sockets E5-2580v4 (14 cores, hyper-threading disabled) processor connected with a P100 (540GB/s) through a PCIe 3.0 connection. The final performances are shown in Table 4, where the hybrid implementation achieved around 97% of the combined performances. The remaining 3% are lost due to communications between the different devices and to the dedication of the first CPU core to the GPU management.

Devices	CPU run: 2xE5-2680v4	GPU run: P100 (540GB/s)	Hybrid run: 2xE5-2680v4+P100
P ($\times 10^6 cus$)	10.4	37	46

Table 4. Performances of the hybrid implementation on the 2731×2731 and $n = 50$ time-steps test case.

7. Conclusion

In this study, a general guideline for high performance achievement on different parallel computing devices has been presented. Contrary to classical usage, the proposed approach consisted in deriving CPU and Xeon-Phi optimized code versions from a GPU one. This concentrates most of the implementation effort on a single code.

This method has proved to be highly beneficial in the case of a two-dimensional discontinuous Galerkin solver for Computational Fluid Dynamics and allowed us to introduce a methodology to develop an hybridized application which uses the same computational kernels for both CPUs and GPUs. As the vector support inside GPU, Xeon-Phi and CPU are quite similar, we believe that this approach should provide similar results for other scientific problems.

As a consequence, the interest of GPU development should be reconsidered, as the development cost for such devices can be amortized by the inexpensive deduction of optimized codes for other computing devices. Also, we have observed that adaptation and integration of the GPU code into CPU/Xeon-Phi codes can be performed, at least in part, automatically. This will be the subject of further work.

Acknowledgments

Authors want to acknowledge the CEA/DIF, IDRIS, GENCI and NVIDIA for accesses to the compute resources used for benchmark and Region Lorraine for its constant support to our research.

References

[1] M. Borrel and J. Ryan. The Elastoplast Discontinuous Galerkin (Edg) Method For The Navier-Stokes Equations. *Journal of Computational Physics*, 231(1), 2012.

[2] L. Einkemmer. A Mixed Precision Semi-Lagrangian Algorithm and its Performance on Accelerators. In *International Conference on High Performance Computing Simulation (HPCS)*, Innsbruck, Austria, July 2016.

[3] Sigal Gottlieb and Chi-Wang Shu. Total Variation Diminishing Runge-Kutta Schemes. *Mathematics of Computation of the American Mathematical Society*, 67(221):73–85, 1998.

[4] Alexander Heinecke, Alexander Breuer, Michael Bader, and Pradeep Dubey. High Order Seismic Simulations on the Intel Xeon Phi Processor (Knights Landing). In *31st International Conference on High Performance Computing (ISC)*, Frankfurt, Germany, June 2016.

[5] Andreas Klöckner, Timothy Warburton, and Jan S. Hesthaven. High-Order Discontinuous Galerkin Methods by GPU Metaprogramming. In D.A. Yuen, L. Wang, X. Chi, L. Johnsson, W. Ge, and S. Yaolin, editors, *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, 2013.

[6] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[7] Mark Sabahi. A Guide to Auto-vectorization with Intel C++ Compilers. 2012.

[8] Avinash Sodani. Knights Landing (KNL): 2nd Generation Intel Xeon Phi processor. In *IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, 2015.

[9] Eleuterio F Toro, Michael Spruce, and William Speares. Restoration of the Contact Surface in the HLL-Riemann Solver. *Shock waves*, 4(1):25–34, 1994.