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

Ludomir Oteski, Guillaume Colin de Verdiere, Sylvain Contassot-Vivier, Stephane Vialle, Juliette Ryan

To cite this version:

Ludomir Oteski, Guillaume Colin de Verdiere, Sylvain Contassot-Vivier, Stephane Vialle, Juliette Ryan. Towards a Unified CPU-GPU code hybridization: A GPU Based Optimization Strategy Efficient on Other Modern Architectures. Parallel Computing Conference (Parco 2017), Sep 2017, Bologne, Italy. hal-01651961

HAL Id: hal-01651961
https://hal-centralesupelec.archives-ouvertes.fr/hal-01651961

Submitted on 29 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Towards a Unified CPU-GPU code hybridization: A GPU Based Optimization Strategy Efficient on Other Modern Architectures

L. Oteski, G. Colin de Verdière, S. Contassot-Vivier, S. Vialle, J. Ryan

Acks.: CEA/DIF, IDRIS, GENCI, NVIDIA, Région Lorraine
Introduction and context

Most scientific simulations
  • Require intensive computations
  • Are frequently based on parallel iterative processes

Classical development process:

1. Sequential (optimized) version
2. CPU multi-threaded version
3. GPU version

We propose to invert the parallel development process:
  • Begin with the GPU version
  • To rapidly derive efficient CPU versions: vectors + threads (+ MPI)
Test case application: Discontinuous Galerkin schemes for Computational Fluid Dynamics

Representative of some compute-intensive applications
We consider the 2D time-dependant compressible Navier-Stokes equations

Example:
*Yee's vortex* with the full NS equations

Numerical method:
- **Space:** 2nd order Discontinuous Galerkin,
- **Time:** 3rd order TVD Runge-Kutta,
- **Boundaries:** X and Y-periodic.
Test case application: Discontinuous Galerkin schemes for Computational Fluid Dynamics

Representative of some compute-intensive applications
We consider the 2D time-dependant compressible Navier-Stokes equations

- Storage space per array of variables: $N_{\text{coefs}} \times N_{\text{eq}} \times N_{\text{cells}}$
  - $N_{\text{coefs}}$: number of polynomial coefficients (6 for 2D second order polynomials)
  - $N_{\text{eq}}$: number of equations (4 in our case)
  - $N_{\text{cells}} = N_x \times N_y$: number of cells in the $x$ and $y$ directions

- 3 double precision arrays: current state, previous time-step and time-derivatives

- Memory cost $\approx 3 \times (6 \times 4 \times N_{\text{cells}}) \times 8$ Bytes
  - Ex: $2001 \times 2001$ mesh $\geq 2.3$ GBytes
    $2731 \times 2731$ mesh $\geq 4.3$ GBytes
Unified Development Approach

GPU and CPU have different architectures, **BUT**:

Today both use **vectorization**:

- **Vectors** for CPU
- **Warps** for GPU

→ Efficient computing scheme on GPU should provide:
  - Compliant CPU scheme with minor adaptations
  - Efficient CPU execution

**Optimized GPU programming**:

- is low-level (« close to the metal »)
- low-level API (CUDA, OpenCL) allows accurate ctrl of computations

→ Clear impact of optimization attempts on GPU
→ Interest of using GPU as the first development step
Unified Development Approach

Example of GPU kernel translated into a vectorized CPU function

2D Grid of blocks of CUDA threads $\rightarrow$ CPU nested loops
Optimization guideline overview

4 crucial GPU optimizations applied to the CPU version
1. Reduce accesses to distant memories
2. Merge kernels which share the same memory patterns
3. Simplify and factor computations
4. Align data (including tiling: CPU specific optimization)

+ CPU specific optimizations
a. Vectorization tuning
b. Data locality improvement
Optimization guideline: GPU → CPU (1)

1. Reduce accesses to distant memories

Transfer redundant accesses to global/DRAM memory into registers
→ GPU avoids unnecessary accesses to global memory
→ CPU improves data cache locality

Ex: GPU perf ×3.97, CPU perf ×1.36
Optimization guideline: GPU → CPU (2)

2. Merge kernels which share the same memory patterns
   → GPU limits its number of accesses to distant memories by improving the re-use of distant variables
   → CPU increases its cache re-use

1. Compute the time-step
2. For s Runge-Kutta step:
   a. Convective fluxes in X
   b. Convective fluxes in Y
   c. Convective integral
   d. Compute local viscosity
   e. Viscous fluxes in X
   f. Viscous fluxes in Y
   g. Viscous integral
   h. Runge-Kutta propagation

1. Compute the time-step
2. For s Runge-Kutta step:
   a. Compute local viscosity
   b. All fluxes in X
   c. All fluxes in Y
   d. Integral+Runge-Kutta

Ex: GPU perf ×1.75, CPU perf ×1.37
Optimization guideline: GPU → CPU (3)

3. **Simplify and factor computations**
   - Suppress non essential variables
   - Store recurrent computations in intermediate vars (mult by const, const square roots,...)
   - Control whether or not a static loop should be unrolled

```c
#pragma unroll
for(int i=0; i<SIZE; i++) {
    ...
    double val = ...
    double c0 = 1/(sqrt(0.5)*val)*...
    double c1 = 1/(3*val)*...
    ...
}
```

→ The most time-consuming development step

```c
double isqrt0p5 = 1/sqrt(0.5);
double inv3 = 1/3;
#pragma unroll //Keep it ?
for(int i=0; i<SIZE; i++) {
    ...
    double val = ...
    double ival = 1/val;
    double c0 = isqrt0p5*ival*...
    double c1 = inv3*ival*...
    ...
}
```

Ex: GPU perf ×1.21, CPU perf ×1.82
Optimization guideline: GPU → CPU (4)

4. Align data

- GPU: align data access on warp size: 32 (static size)
  → ensures coalescent memory accesses
  → GPU perf ×1.03 on our problem

- CPU:
  1. Add a tiling algorithm inside each thread computations
     → CPU perf ×1.23
  2. Use static tile size
     → CPU perf ×1.19

Static tile size → static nb of loop iterations → better vectorization
Optimization guideline: CPU specific (1)

a. Vectorization tuning

Previous optimizations steps tend to create huge vectorized loops!

→ high increase of register pressure
→ not suitable for CPU vectorization...

So ... we adjusted our CPU vectorization strategy

1. by splitting SIMD loops into smaller loops
2. #pragma simd → #pragma vector always

* tells the compiler to perform auto-vectorization if the loop does not carry dependencies

→ CPU perf ×1.03

Compromise *distant memory accesses reduction / register pressure* for optimal CPU vectorization
Optimization guideline: CPU specific (2)

b. Data locality improvement

MPI-OpenMP version of the CPU code $\rightarrow$ to improve data locality

• Memory locations of data used in (only) one MPI process and its threads will be close to their associated cores

• This optimization is well suited to NUMA architectures (and many machines are becoming NUMA machines !)

$\rightarrow$ CPU perf $\times 1.03$

*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*
# Experimental steps & performances (1)

<table>
<thead>
<tr>
<th>Code versions</th>
<th>CPU E5-1650v3 (6th)</th>
<th>Speedup vs seq. CPU</th>
<th>Progressive speedup</th>
<th>GPU K20Xm</th>
<th>Speedup vs seq. CPU</th>
<th>Progressive speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq. CPU v (1th)</td>
<td>8.35 x10^4 cus</td>
<td>1.00</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU v1 (OMP+simd pragmas)</td>
<td>0.59 x10^6 cus</td>
<td>7.01</td>
<td>7.01</td>
<td>1.44 x10^6 cus</td>
<td>17.21</td>
<td>17.21</td>
</tr>
</tbody>
</table>

**GPU v1 (CUDA)**

**Note:**
- **cus:** Cell Update/s
- **2001 × 2001 cells**
- **100 steps**
## Experimental steps & performances (2)

<table>
<thead>
<tr>
<th>Code versions</th>
<th>CPU E5-1650v3 (6th)</th>
<th>Speedup vs seq. CPU</th>
<th>Progressive speedup</th>
<th>GPU K20Xm</th>
<th>Speedup vs seq. CPU</th>
<th>Progressive speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq. CPU v (1th)</td>
<td>8.35 x10^4 cus</td>
<td>1.00</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU v1 (OMP+simd pragmas)</td>
<td>0.59 x10^6 cus</td>
<td>7.01</td>
<td>7.01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU v1 (CUDA)</td>
<td>0.81 x10^6 cus</td>
<td>9.63</td>
<td>1.36</td>
<td>1.44 x10^6 cus</td>
<td>17.21</td>
<td>17.21</td>
</tr>
<tr>
<td>Reduced accesses to distant mem.</td>
<td>1.09 x10^6 cus</td>
<td>13.05</td>
<td>1.37</td>
<td>5.71 x10^6 cus</td>
<td>68.36</td>
<td>3.97</td>
</tr>
<tr>
<td>Merged funcs with same mem pattern</td>
<td>2.01 x10^6 cus</td>
<td>24.05</td>
<td>1.82</td>
<td>10.0 x10^6 cus</td>
<td>119.62</td>
<td>1.85</td>
</tr>
<tr>
<td>Simplification of computations</td>
<td>2.48 x10^6 cus</td>
<td>29.72</td>
<td>1.23</td>
<td>12.1 x10^6 cus</td>
<td>145.00</td>
<td>1.21</td>
</tr>
<tr>
<td>CPU Only: Tiling (cache optim)</td>
<td>2.86 x10^6 cus</td>
<td>34.25</td>
<td>1.19</td>
<td>12.5 x10^6 cus</td>
<td>149.53</td>
<td>1.03</td>
</tr>
</tbody>
</table>

**cus:** Cell Update/s

2001 × 2001 cells

100 steps

**Parco - September 2017 – Bologna, Italy**
### Experimental steps & performances (3)

<table>
<thead>
<tr>
<th>Code versions</th>
<th>CPU E5-1650v3 (6th)</th>
<th>Speedup vs seq. CPU</th>
<th>Progressive speedup</th>
<th>GPU K20Xm</th>
<th>Speedup vs seq. CPU</th>
<th>Progressive speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq. CPU v (1th)</td>
<td>8.35 x10^4 cus</td>
<td>1.00</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU v1 (OMP+simd pragmas)</td>
<td>0.59 x10^6 cus</td>
<td>7.01</td>
<td>7.01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU v1 (CUDA)</td>
<td></td>
<td></td>
<td></td>
<td>1.44 x10^6 cus</td>
<td>17.21</td>
<td>17.21</td>
</tr>
<tr>
<td>Reduced accesses to distant mem.</td>
<td>0.81 x10^6 cus</td>
<td>9.63</td>
<td>1.36</td>
<td>5.71 x10^6 cus</td>
<td>68.36</td>
<td>3.97</td>
</tr>
<tr>
<td>Merged funcs with same mem pattern</td>
<td>1.09 x10^6 cus</td>
<td>13.05</td>
<td>1.37</td>
<td>10.0 x10^6 cus</td>
<td>119.62</td>
<td>1.85</td>
</tr>
<tr>
<td>Simplification of computations</td>
<td>2.01 x10^6 cus</td>
<td>24.05</td>
<td>1.82</td>
<td>12.1 x10^6 cus</td>
<td>145.00</td>
<td>1.21</td>
</tr>
<tr>
<td>CPU Only: Tiling (cache optim)</td>
<td>2.48 x10^6 cus</td>
<td>29.72</td>
<td>1.23</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data alignment</td>
<td>2.86 x10^6 cus</td>
<td>34.25</td>
<td>1.19</td>
<td>12.5 x10^6 cus</td>
<td>149.53</td>
<td>1.03</td>
</tr>
<tr>
<td>CPU Only: Tuning on vectorization</td>
<td>2.96 x10^6 cus</td>
<td>35.44</td>
<td>1.03</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CPU Only: MPI + OMP (3 proc x 2th)</td>
<td>3.05 x10^6 cus</td>
<td>36.52</td>
<td>1.03</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Note:**
- `cus`: Cell Update/s
- `2001 × 2001 cells 100 steps`
Experimental testbed

Hardware:
- Bi-socket E5-2698v3
- Bi-socket E5-2680v4
- KNL 7250
- K20Xm
- K40
- P100 540GB/s
- P100 720GB/s

Application:
- 2D Discontinuous Galerkin
  (NS equations)
- Grid size: 2731×2731

Operating System:
- Linux

Compilers:
- CUDA 8
- ICC

ICC options:
- -O3
- -march=native
- -fma
- -align
Experimental final performances (1)

- KNL 7250 with 16GB MCDRAM (~500 GB/s)
  - a little bit faster than GPU K20Xm and GPU K40

Applying our optimisation guideline
Experimental final performances (2)

But GPU P100 great winner!!

P100

P100 720GB/s
HBM2, NVLINK

P100 540GB/s
HBM2

CPU

GPU

<table>
<thead>
<tr>
<th>Graph bars</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>cus/Watts (×10^4)</td>
<td>cus (×10^6) [OpenMP+MPI]</td>
</tr>
<tr>
<td>cus (×10^6) [CUDA]</td>
<td></td>
</tr>
</tbody>
</table>
Experimental single-threaded cache roofline profile

Perf (GFLOPS) on one E5-2680v4 core (measured with Intel Advisor 2017)

86.6

Time consuming loops
Above L2 BdW limit (good)

0.36

0.016

Arithmetic Intensity (FLOPS/Byte)

1.65

Global application

Storage function (not significant)
Foundation of Hybrid solution

**GPU**
A large block of $n_{th}$ light threads

An array of $n$ data

$n = n_{th} \times \text{VSIZ}$

Ex: $12 = 12 \times 1$

**CPU**
One CPU thread with VSIZ vector units

An array of $n$ data

$n = n_{th} \times n_{steps} \times \text{VSIZ}$

Ex: $12 = 1 \times 3 \times 4$
Hybrid implementation

### Common src code for CPU & GPU

<table>
<thead>
<tr>
<th>2D CUDA-GPU code (GPU.cu)</th>
<th>Common vectorized kernel (kernel.h)</th>
<th>2D CPU code (CPU.cpp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define VSIZ 1</td>
<td>#template&lt;const int VSIZ&gt;</td>
<td>#define VSIZ 32</td>
</tr>
<tr>
<td>//Include the kernel</td>
<td>//Conditional compilation</td>
<td>//Include the kernel</td>
</tr>
<tr>
<td>#define DEFGPU</td>
<td>#ifdef DEFGPU</td>
<td>#include &quot;kernel.h&quot;</td>
</tr>
<tr>
<td>#include &quot;kernel.h&quot;</td>
<td><strong>device</strong></td>
<td></td>
</tr>
<tr>
<td>void <strong>global</strong> gpu_function(</td>
<td>inline void kernel(</td>
<td>void cpu_function(</td>
</tr>
<tr>
<td>double *<strong>restrict</strong> val)</td>
<td>double *<strong>restrict</strong> val,</td>
<td>double *<strong>restrict</strong> val)</td>
</tr>
<tr>
<td>7</td>
<td>const int tidx</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>//Vectorized loop</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>#pragma vector always</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>#pragma unroll</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>for(vec=0; vec&lt;VSIZ; vec++)</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>kernel&lt;VSIZ&gt;(val,tidx);</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>...</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>val[VSIZ+tidx+vec] = ...;</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>18</td>
</tr>
</tbody>
</table>

- **C++ template** code, parametered with the nb of elements processed by each thread
- **Compiler directives** (ignored when not using the right compiler and architecture)
Hybrid system

MPI Process 0

GPU 0

GPU 1

CPU 0

CPU 1

MPI Process 1

CPU 2

CPU 3

MPI Process 2

CPU 4

CPU 5

MPI Process 3
Sketch of hybridized code

CPU(s) managed by others MPI processes

GPU(s) managed by MPI process 0

Does the node have a GPU?

Yes

No

Yes

No

Process 0 of the node manages all GPUs of the node + GPU initialization

CPU initialization

Main

Is the MPI process the first of the node?

Yes

No

MPI communications (+ I / O)

GPU Time loop + GPU(s) communications

GPU finalization

End program

GPU template configuration (vector size = 1)

Common Vectorized kernels

CPU template configuration (vector size ≥ 1)

CPU Time loop (+tiling)

CPU finalization
## Experimental Hybrid Performance

<table>
<thead>
<tr>
<th>Devices</th>
<th>CPU run: 2xE5-2680v4</th>
<th>GPU run: P100 (540GB/s)</th>
<th>Hybrid run: 2xE5-2680v4 + P100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perf (x10^6 cus)</td>
<td>10.4</td>
<td>37.0</td>
<td>46.0</td>
</tr>
</tbody>
</table>

Maximum perf: 10.4 + 37.0 = \textbf{47.4}

Close to ideal (hybrid) performances
Conclusion & Perspective

- General programming guideline for modern computing devices
  CPU, NUMA CPU, GPU, Xeon-Phi KNL
- Highly efficient with the Discontinuous Galerkin problem
  Should also be efficient on other highly vectorizable problems
- Common (hybrid) computing kernel for all devices
  Inserted in a hybrid source code (for CPU+GPU machines)

→ GPU development also valuable for other (vector) targets
→ Increases the interest of developing first on GPU

Perspective:
(Half-) Automatic code generation for CPU from the GPU one
Towards a Unified CPU-GPU code hybridization: A GPU Based Optimization Strategy Efficient on Other Modern Architectures

Questions?

We propose to invert the parallel development process:
• Begin with the GPU version
• To rapidly derive efficient CPU versions: vectors + threads (+ MPI)