Towards an Efficient CPU-GPU Code Hybridization:
a Simple Guideline for Code Optimizations on Modern Architecture
with OpenACC and CUDA

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

Acks.: CEA/DIF, IDRIS, GENCI, NVIDIA, Région Lorraine.
CFD at ONERA

One simulation = thousands of CPU hours

From http://www.onera.fr
Why optimize?

**Cost** of a CPU hour ~$0.05 → $0.10 [Walker (2009)].

**Example:** 1000 cores for 1 month ~$36,000 → $72,000.

**Dividing** application runtime allows to:

→ be more **cost-efficient**,

→ get **faster** results,

→ **increase** the problem size.
CPU-GPU differences

**CPU:** few # of cores, vector machines, hard to manage registers,

**GPU:** huge # of cores, vector machines, easy to manage registers (--ptxas-options=-v with nvcc).

**Question:**
Due to vector approaches, could GPU and CPU be considered as similar devices?

→ Could the GPU be used to optimize the CPU code?
A prototype flow

Yee's vortex with the full Navier-Stokes equations,

2 dimensional structured mesh,

**Numerical method:**

→ **Space:** 2\textsuperscript{nd} order Discontinuous Galerkin,

→ **Time:** 3\textsuperscript{rd} order TVD Runge-Kutta,

→ **Boundaries:** X and Y-periodic.
Test case configuration

2001x2001 mesh, 100 time-steps.

**CPU:** Intel Xeon E5-1650v3 (6 cores),

**Compiler:**
icc 2016, -O3 -march=native

**GPU:** K20Xm (2688 CUDA cores)

**Compilers:**
nvcc CUDA-8.0, -O3
pgc++ 16.10, -O3 -acc
   -ta=tesla:cuda8.0,nollvm,nordc
Experimental protocol

Developments with the full CUDA version

GPU-CUDA

```
int tidx=threadIdx.x+...;
int tidy=threadIdx.y+...;
if(tidx<Nx && tidy<Ny)
{
    ...
}
```

CPU-OpenMP

```
#pragma omp for
for(int tidy=0;tidy<Ny;tidy++)
{
    #pragma simd
    for(int tidx=0;tidx<Nx;tidx++)
    {
        ...
    }
}
```
Experimental protocol

Developments with the full CUDA version

GPU-CUDA

```c
int tidx = threadIdx.x + ...;
int tidy = threadIdx.y + ...;
if (tidx < Nx && tidy < Ny)
{
...}
```

Copy

GPU-OpenACC

```c
#pragma acc parallel
{
#pragma acc loop gang independent
for (int tidy = 0; tidy < Ny; tidy++)
{
#pragma acc loop vector independent
for (int tidx = 0; tidx < Nx; tidx++)
{
...
}
}
}```

Paste
Step 0: Code adaptation

### GPU-CUDA
- Host → Device transfers
- cudaMemcpy(...)
- Time loop + CUDA kernels
- Device → Host transfers
- cudaMemcpy(...)

### CPU-OpenMP
- #pragma omp parallel
  - Time loop + omp pragmas
- } // end of the omp region

### GPU-OpenACC
- #pragma acc data
copy(...)
copyin(...)
create(...)
- Time loop + acc pragmas
- } // end of the acc region
Step 0: Code adaptation

\[ P(\text{cell updt. per sec. [cus]}) = \frac{\# \text{of points} \times \text{number of it.}}{\text{time (sec)}} \]

- CPU: E5-1650v3 (6 cores)
- GPU: K20Xm

Bar chart showing performance comparison:
- CPU Reference: \(10^5\)
- CPU-OpenMP: \(15\times10^5\)
- GPU-OpenACC: \(17\times10^5\)
- GPU-CUDA: \(17\times10^5\)
Step 1: Reduce remote accesses

**Load** reused remote variables in registers (time locality),

**Use** `__restrict__` keyword on pointers.

```c
__global__ void func(double *results, int N) {
    int tidx=threadIdx.x+...;
    ...
    for(int i=0;i<N;i++)
    {
        ...
        results[tidx]+=...
    }
}

__global__ void func(double *__restrict__ results, int N) {
    int tidx=threadIdx.x+...;
    ...
    double loc_result=results[tidx];
    for(int i=0;i<N;i++)
    {
        ...
        loc_result+=...
    }
    results[tidx]=loc_result;
}
```
Step 1: Reduce remote accesses

\[ P(\text{cell update per sec. [cus]}) = \# \text{ of points} \times \text{number of it.} / \text{time (sec)} \]

- CPU: E5-1650v3 (6 cores)
- GPU: K20Xm

**Comparison:**
- **CPU-Reference**
- **CPU-OpenMP**
- **GPU-OpenACC**
- **GPU-CUDA**

- **CPU-Reference**
  - \( \times 9.6 \)
  - \(+36.2\%\)

- **CPU-OpenMP**
  - \( \times 34 \)
  - \(+127.0\%\)

- **GPU-CUDA**
  - \( \times 68 \)
  - \(+297.1\%\)
Step 2: Merge similar kernels

Merge kernels which share similar memory patterns.

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. Fluxes in $X$,
   c. Fluxes in $Y$,
   d. Integral+Runge-Kutta.
Step 2: Merge similar kernels

\[ P(\text{cell updt. per sec. [cus]}) = \# \text{of points} \times \text{number of it. / time (sec)} \]

- **CPU**: E5-1650v3 (6 cores)
- **GPU**: K20Xm

Diagram:
- **CPU-Reference**: 10^6 × 13
- **CPU-OpenMP**: 10^6 × 42 × +36.5%
- **GPU-OpenACC**: 10^6 × 42 × +23.7%
- **GPU-CUDA**: 10^6 × 119 × +75.0%
Step 3: Factor computations

**Transform** divisions into multiplications,

**Control** whether or not a 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)*...
    ...
}
```

```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*...
    ...
}
```
Step 3: Factor computations

\[ P(\text{cell updt. per sec. [cus]}) = \# \text{of points} \times \text{number of it.} / \text{time (sec)} \]

- **CPU**: E5-1650v3 (6 cores)
- **GPU**: K20Xm

### Graph

- **CPU** - Reference
- **CPU** - OpenMP
- **GPU** - OpenACC
- **GPU** - CUDA

- CPU: E5-1650v3 (6 cores)
- GPU: K20Xm

- CPU speedup: \( \times 24 \) (82.9%)
- GPU speedup: \( \times 65 \) (54.1%)
- Total speedup: \( \times 145 \) (21.2%)
Step 4: Align data (+tiling for CPU)

**GPU:**

Align data on the size of a warp → coalescence.

**CPU:**

Tiling (Block) algorithm + fixed block size.

![Tiling algorithm for OpenMP threads.](image)
Step 4: Align data (+tiling for CPU)

$P(\text{cell updt. per sec. [cus]}) = \#\text{of points} \times \text{number of it.} / \text{time (sec)}$

- CPU: E5-1650v3 (6 cores)
- GPU: K20Xm

![Diagram showing performance comparison between CPU and GPU]
Step 5: Miscellaneous

OpenACC PGI 16.10:

- O1 / O2 / O3 / O4 compilation → registers overflow.
- O0 = +82% performances (corrected in v17).

CPU:

Introduce MPI to support several nodes and to improve data locality on each one.

#pragma simd on huge loops creates huge register pressure:

→ not suitable for CPU vectorization.
→ split SIMD loops into smaller loops
→ adjust vectorization with #pragma vector always.
Step 5: Miscellaneous

\[ P(\text{cell updt. per sec. [cus]}) = \frac{\# \text{of points} \times \text{number of it.}}{\text{time (sec)}} \times 10^6 \]

CPU: E5-1650v3 (6 cores)
GPU: K20Xm

CPU–Reference
CPU–OpenMP
GPU–OpenACC
GPU–CUDA

+0.0%  \times 149
+82.5%  \times 119
+3.1%  \times 36
Running on various devices

\[ P(\text{cell updt. per sec. [cus]}) = \# \text{of points} \times \text{number of it.} / \text{time (sec)} \]

- **Xeon Phi**
  - cuS/Watts \( \times 10^4 \)
  - cuS \( \times 10^6 \) [OpenMP]
  - cuS \( \times 10^6 \) [OpenACC]
  - cuS \( \times 10^6 \) [CUDA]

**CPU**

**GPU**

- **P100**
  - Nvlink 720GB/s HBM2
  - 540GB/s HBM2

*TDP evaluated from constructor's documentsations*
Vectorized OpenACC code

Run on 2xE52680v4 (CPU):
OpenACC code for GPU with pgc++ and
-ta=multicore -tp=x64 (CPU execution)
→ 2.72 x 10^6 cus → ~25% performances of the Intel version.

CPU code with pgc++ and
-ta=multicore -tp=x64 (CPU execution)
→ 5.4 x 10^6 cus → ~50% performances of the Intel version.

Therefore:
OpenACC for CPUs must be developed on smaller loops than GPU kernels.
Add tiling algorithms to improve CPU cache efficiency.
What's next?

→ **GPUs can be** used to efficiently optimize CPU codes
→ this optimization strategy leads to highly similar kernels.

Could it be possible to use the **same kernels for both CPU and GPU** within a single version of the code?
A unified development approach

Use C++ templates and pre-compilation variables to define which version should be used.

<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></td>
</tr>
<tr>
<td>#include &quot;kernel.h&quot;</td>
<td>ifdef DEFGPU</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>device</em>_</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>{</td>
<td>const int tidx)</td>
<td>{</td>
</tr>
<tr>
<td>//Thread indexes</td>
<td>//Vectorized loop</td>
<td>//CPU loop</td>
</tr>
<tr>
<td>int tidx = ...;</td>
<td>#pragma vector always</td>
<td>for(int tidx=0;</td>
</tr>
<tr>
<td>//Logical condition</td>
<td>#pragma unroll</td>
<td>tidx&lt;Nx;</td>
</tr>
<tr>
<td>//to make computations</td>
<td>for(vec=0; vec&lt;VSIZ; vec++)</td>
<td>tidx+=VSIZ)</td>
</tr>
<tr>
<td>if(tidx&lt;Nx)</td>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>{</td>
<td>kernel&lt;VSIZ&gt;(val,tidx);</td>
<td>kernel&lt;VSIZ&gt;(val,tidx);</td>
</tr>
<tr>
<td>...</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>18 }</td>
<td>18</td>
<td>18</td>
</tr>
</tbody>
</table>

Compiler directives (ignored when not using the right compiler),
Template parameter to control the vector size (VSIZ=1 for GPU, VSIZ ≥ 1 for CPU),
Inline keyword to avoid conflicts in function names.
Sketch of the hybridized code

GPU PART managed by MPI process 0 of the node.

CPU PART for all other MPI processes of the node.

Does the node have a GPU?

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

No: CPU initialization

Main

Is the MPI process the first of the node?

Yes: MPI communications (+ I/O)

No: CPU Time loop (+tiling)

CPU template configuration (vector size ≥ 1)

GPU template configuration (vector size = 1)

GPU Time loop + GPU(s) communications

GPU finalization

Common Vectorized kernels

End program

CPU finalization
Work-balancing

Poor balance between performances of CPUs and GPUs.

→ Micro-benchmark inside the code to balance work between processes:

\[ W_i = \frac{P_i}{\sum_{i=0}^{n_p} P_i} \]

Performances of MPI process \( i \) with the actual work-balance.

Work for MPI process \( i \).

Total work to be done.

Sum of performances with the actual work-balance.
Strong scalability

Efficiency (%) = \frac{\text{effective performance}}{\text{max. performance}} \times 100

Devices (1 node):
2xE5–2680v4 +GTX1060

Devices (2 nodes):
4xE5–2680v4 +2xGTX1060

Devices (3 nodes):
6xE5–2680v4 +2xGTX1060 +P100(540GB/s)
Conclusion

→ **GPUs can be** used to efficiently optimize CPU codes,

→ **4 steps** to optimize:
  
  **Reduce** access to remote memories,
  
  **Merge** kernels with similar data access pattern,
  
  **Factor** your computations,
  
  **Align** your data.

**Future work:** move to unstructured meshes

→ ranking on engine's performances may change.

To go deeper into GPU optimization: [Woolley (2013)].
Thank you for your attention.

Contact: ludomir.oteski@onera.fr
Cache roofline (Intel Advisor 2017)

Single threaded E5-2680v4

Performances of the application
~11 GFlops

Storage function
## Optimization steps

### Ref. Case: 4785s

<table>
<thead>
<tr>
<th>Modif.</th>
<th>CPU (6th)</th>
<th>Speedup</th>
<th>OpenACC</th>
<th>Speedup</th>
<th>CUDA</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptation</td>
<td>677s</td>
<td>7.06</td>
<td>320s</td>
<td>14</td>
<td>278s</td>
<td>17.2</td>
</tr>
<tr>
<td>Reduce accesses</td>
<td>497s</td>
<td>9.63</td>
<td>141s</td>
<td>33.9</td>
<td>70s</td>
<td>68.4</td>
</tr>
<tr>
<td>Merge kernels</td>
<td>364s</td>
<td>13.1</td>
<td>114s</td>
<td>42</td>
<td>40s</td>
<td>119</td>
</tr>
<tr>
<td>Factor. Comput.</td>
<td>199s</td>
<td>24</td>
<td>74s</td>
<td>64.7</td>
<td>33s</td>
<td>145</td>
</tr>
<tr>
<td>Tiling</td>
<td>161s</td>
<td>29.7</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Alignment</td>
<td>135s</td>
<td>35.4</td>
<td>73s</td>
<td>65.5</td>
<td>32s</td>
<td>149</td>
</tr>
<tr>
<td>OpenMP +MPI</td>
<td>131s</td>
<td>35.5</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>PGI -O0 compil.</td>
<td>--</td>
<td>--</td>
<td>40s</td>
<td>119</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>