Towards a Unified CPU/GPU Codebase
For A Linear Scaling FMM Coulomb Solver

April 7th, 2016 | Albert García | Ivo Kabadshow | Jülich Supercomputing Centre
Outline

- Theoretical Background
- Previous Application Layout
- Unified Application Layout
- Results & Conclusion
Alternatives to Direct Summation

- **Direct Summation**
  - Machine precision $O(N^2)$
  - Error bounds up to $O(N)$
Alternatives to Direct Summation

- Direct Summation: $O(N^2)$, no error bounds.
- Cut-Off Schemes: $O(N)$ error bounds up to $O(N)$.
Alternatives to Direct Summation

- **Direct Summation**
  - Machine precision $O(N^2)$
  - No error bounds $O(N)$
- **Cut-Off Schemes**
  - Error bounds up to $O(N)$
- **Fast Summation Methods**
  - Error bounds up to $O(N)$
Overview of Fast Summation Methods

- **FFT-based**
  - Ewald: $O(N^{3/2})$
  - PME: $O(N \log N)$
  - P3M: $O(N \log N)$
  - P2NFFT: $O(N \log N)$

- **PDE-based**
  - Multigrid: $O(N)$

- **Multipole-based**
  - Treecode: $O(N \log N)$
  - FMM: $O(N)$
Overview of Fast Summation Methods

- **FFT-based**
  - Ewald: $O(N^{3/2})$
  - PME: $O(N \log N)$
  - P3M: $O(N \log N)$
  - P2NFFT: $O(N \log N)$

- **PDE-based**
  - Multigrid: $O(N)$

- **Multipole-based**
  - Treecode: $O(N \log N)$
  - FMM: $O(N)$
The Fast Multipole Method in a Nutshell

Core concepts: reduced interaction scheme

(a) direct interaction

(b) FMM interaction
The Fast Multipole Method in a Nutshell
Core concepts: particle grouping
The Fast Multipole Method in a Nutshell

Core concepts: spatial subdivision with octree

(a) depth = 0  
(b) depth = 1  
(c) depth = 2
The Fast Multipole Method in a Nutshell

Core concepts: well separateness (ws)

(a) $ws = 1$

(b) $ws = 2$

(c) $ws = 3$
Classical Sequential FMM Workflow

User needs to setup three algorithmic parameters

- Tree depth $d$
- Expansion order $p$
- Separation criterion $ws$

⚠️ Optimal parameter set not known analytically

→ Run-time and error bound strongly depend on these parameters
The Fast Multipole Method in a Nutshell

Workflow

Pass 1: Particle-to-Multipole $\mathcal{O}(Np^2)$

P2M
The Fast Multipole Method in a Nutshell

Workflow

Pass 1: Multipole-to-Multipole (M2M) $\mathcal{O}(p^4) \| \mathcal{O}(p^3)$
The Fast Multipole Method in a Nutshell

Workflow

Pass 2: Multipole-to-Local (M2L) $O(p^4)\|O(p^3)$
The Fast Multipole Method in a Nutshell

Workflow

Pass 3: Local-to-Local (L2L) $O(p^4)\|O(p^3)$
The Fast Multipole Method in a Nutshell

Workflow

Pass 4: Local-to-Particle $\mathcal{O}(Np^2)$
The Fast Multipole Method in a Nutshell

Workflow

Pass 5: Particle-to-Particle $O(M^2)$

P2P
The Fast Multipole Method in a Nutshell
Rotation-based operators to reduce complexity from $O(p^4)$ to $O(p^3)$

(a) interaction set

(b) rotation
The Fast Multipole Method in a Nutshell
Rotation-based operators to reduce complexity from $O(p^4)$ to $O(p^3)$

(a) translation

(b) rotation backwards
Goals

- Explore the possibilities of deploying $O(p^3)$ operators on a GPU
- Minimize the impact on the project, keep single codebase
- Make use of existing C++11 features in the current code
- Set up abstraction layers to choose CPU/GPU code
Current CPU $O(p^4) - O(p^3)$ Layout

- FMM C++11
  - $<p^3/p^4>$
  - M2M → M2L → L2L

- Data Structures
  - $<\text{float/double}>$
  - $\omega$ → $R$ → POD

- Allocator
  - $<\text{std::allocator}>$

- Memory
CPU Scaling $\mathcal{O}(p^4) - \mathcal{O}(p^3)$ on JURECA

Full M2M over lowest level of the tree, $d = 4$, double precision
**Existing GPU Implementation of \( O(p^4) \) Operators**

Conclusions of the work presented at GTC2015 by Bartosz Kohnke

<table>
<thead>
<tr>
<th><strong>The Smooth Part</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Fairly fast start possible due to unified memory</td>
</tr>
<tr>
<td>- Dynamic parallelization avoids subsequent index computations (developer and hardware)</td>
</tr>
<tr>
<td>- Dynamic parallelization allows to achieve good performance (only for some predefined parameter setup)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>The Bumpy Part</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Finding the best parallelization for every setup is not trivial</td>
</tr>
<tr>
<td>- One ‘fits all’ kernel not possible (hybrid approach necessary)</td>
</tr>
<tr>
<td>- ( O(p^3) ) vs. ( O(p^4) ) operator GPU parallelization benefit?</td>
</tr>
</tbody>
</table>
Custom CUDA Managed Allocator
Inheritance from std::allocator and overriding allocate/deallocate

```cpp
pointer allocate(
    size_type n,
    cuda_managed_allocator<void>::const_pointer /*hint*/ = 0)
{
    pointer p;
    cudaError_t ret = cudaMallocManaged(&p, n * sizeof(T));
    // Error checking and exception throwing...
    return p;
}

void deallocate(pointer p, size_type /*n*/)
{
    cudaFree(p);
}
```
CPU-GPU with Unified Memory Layout

FMM C++11

algorithm

CPU  M2M  M2L  L2L  M2M  M2L  L2L  GPU

data structures

π  ω  P0D

allocator

<std::::allocator>

memory

CPU  unified memory  GPU
Unified Memory: Allocation Calls Limit

- Allocation Calls
- Chunk Size (B)
- K40m (12 GiB)
- K20Xm (6 GiB)
Unified Memory: Total Memory Limit

Memory Allocated (B)

- 16 Gi
- 256 Mi
- 4 Mi
- 64 Ki
- 1 Ki

Chunk Size (B)

- $2^1$
- $2^4$
- $2^7$
- $2^{10}$
- $2^{13}$
- $2^{16}$
- $2^{19}$

- K40m (12 GiB)
- K20Xm (6 GiB)
Unified Memory with Simple Allocation Scheme

Memory limit due to maximum allocations

\[ m \text{KiB} \]

\[ e_0 \]

\[ \vdots \]

\[ e_{k+0} \]

\[ e_{k+1} \]

\[ \vdots \]

\[ e_{k+l-1} \]

\[ e_{k+l+0} \]

\[ \text{alloc} \]

\[ 0 \]

\[ \text{memory} \]

\[ m \cdot (k + l) - 1 \]

\[ \text{bad_alloc} \]

\[ \text{bad_alloc} \]

\[ \text{bad_alloc} \]

\[ \text{out of memory} \]
Unified Memory with Pool Allocator Scheme

Memory limit due to maximum capacity

\[m \cdot k - 1\]

\[e_0 \rightarrow \text{alloc} \rightarrow 2m \text{KiB} \rightarrow b_0 \rightarrow \text{alloc} \rightarrow 0\]

\[e_1 \rightarrow \text{alloc} \rightarrow \text{pool allocator} \rightarrow b_0 \rightarrow \text{alloc} \rightarrow \text{memory} \approx m \cdot k - 1\]

\[\vdots\]

\[e_{k-1} \rightarrow \text{alloc} \rightarrow \text{pool allocator} \rightarrow b_n \rightarrow \text{alloc} \rightarrow b_n+1 \rightarrow \text{bad_alloc} \rightarrow \text{out of memory}\]

\[e_{k+0} \rightarrow \text{alloc} \rightarrow \text{pool allocator} \rightarrow b_n \rightarrow \text{alloc} \rightarrow b_{n+1} \rightarrow \text{bad_alloc} \rightarrow \text{out of memory}\]

\[e_{k+1} \rightarrow \text{bad_alloc} \rightarrow \text{out of memory}\]

\[e_{k+2} \rightarrow \text{bad_alloc} \rightarrow \text{out of memory}\]
CPU-GPU with Pool Allocator Layout

FMM C++11

algorithm
CPU | M2M → M2L → L2L | M2M → M2L → L2L | GPU

data structures

pool allocator

allocator

memory

CUDA pool allocator

R

ω

POD

CUDA

std

unified memory

GPU

std

cuda
Effects of the Pool Allocator on JUHYDRA
Running full CPU $\mathcal{O}(p^3)$ M2M operator, $d = 4$, double precision

- Runtime (s)
- Poles

- Using pool allocator
- Not using pool allocator
CPU-GPU Independent Codebases

- Algorithm: CPU → M2M → M2L → L2L → M2M → M2L → L2L → GPU
- Data structures: ω → R → POD
- Pool allocator: std → CUDA
- Memory: CPU → unified memory → GPU
CPU and GPU Independent Codebases

Multipole rotation CPU implementation

\[ \omega'_{l,m}(a, \alpha, \beta) = \sum_{k=-l}^{l} \frac{\sqrt{(l-k)!}(l+k)!}{\sqrt{(l-m)!}(l+m)!} d_{l}^{m,k}(\theta) e^{ik\phi} \omega_{l,k} \quad l \in \{0, \ldots, p\}, m \in \{0, \ldots, l\} \]

for (index l = 0; l <= p; ++l){
    for (index m = 0; m <= l; ++m){
        complex omg = d_f(l,m,0)[0] * omega_in.get_upper(l,0);

        for (index k = 1; k <= l; ++k){
            // Computations with d_f and omg...
        }

        omega_out(l,m) = omg;
    }
}
CPU and GPU Independent Codebases

Multipole rotation GPU implementation

typedef cub::WarpReduce<complex_type> WarpReduce;
__shared__ typename WarpReduce::TempStorage omega_storage[NUM_WARPS];

for (index = l_start; l <= p; l += l_stride) {
    for (index m = m_start; m <= l; m += m_stride) {
        complex omg(0.0);
        if (0 == threadIdx.x)
            omg = d_f(l, m, 0)[0] * omega_in.get_upper(l, 0);
        for (index k = k_start; k <= l; k += k_stride) {
            // Computations with d_f and omg...
        }
        complex omega = WarpReduce(omega_storage[warp_id]).Sum(omg);
        if (0 == threadIdx.x)
            omega_out(l, m) = omega;
    }
}
CPU-GPU Merged Codebases

FMM C++11

algorithm

data structures

pool allocator

allocator

memory

CPU &r

M2M → M2L → L2L

GPU *p + #macros

POD

std

cuda

unified memory
Results on JUHYDRA

Full $\mathcal{O}(p^3)$ M2M operator over the lowest level of a $d = 4$ octree, double precision

![Graph showing runtime vs. number of poles for CPU and GPU. The graph illustrates the performance comparison between CPU (Intel Xeon E5-2650) and GPU (NVIDIA Tesla K40m) for different numbers of poles, demonstrating a speedup of $4.0 \times$ at the crossover point $p = 7$.](image-url)
FMM Time Distribution

$N = 103000$, $p = 10$, $d = 4$ parallelized $O(p^4)$ kernels on Tesla K40m

- P2P: 39.61%
- P2M: 3.95%
- M2M: 2.50%
- Forces: 3.52%
- L2L: 2.18%
- M2L: 48.24%
Exposing Parallelism
Coefficient matrix partitioning $p = 15$

Multipole moment: $\omega_{l,m}$

or Local moment: $\mu_{l,m}$

$a_{l,m} + ib_{l,m}$

more precision
less precision
Exposing Parallelism

Block distribution, one per each column
Exposing Parallelism

Example of block work distribution for $b_{10}$
Exposing Parallelism

Block configuration: $32 \times 4$ threads

<table>
<thead>
<tr>
<th>w3</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
<th>...</th>
<th>t29</th>
<th>t30</th>
<th>t31</th>
</tr>
</thead>
<tbody>
<tr>
<td>w2</td>
<td>t0</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>t4</td>
<td>t5</td>
<td>t6</td>
<td>t7</td>
<td>t8</td>
<td>...</td>
<td>t29</td>
<td>t30</td>
<td>t31</td>
</tr>
<tr>
<td>w1</td>
<td>t0</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>t4</td>
<td>t5</td>
<td>t6</td>
<td>t7</td>
<td>t8</td>
<td>...</td>
<td>t29</td>
<td>t30</td>
<td>t31</td>
</tr>
<tr>
<td>w0</td>
<td>t0</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>t4</td>
<td>t5</td>
<td>t6</td>
<td>t7</td>
<td>t8</td>
<td>...</td>
<td>t29</td>
<td>t30</td>
<td>t31</td>
</tr>
</tbody>
</table>
Exposing Parallelism

Warp distribution, warps compute individual elements
Exposing Parallelism

Example of warp work distribution for $b_{10}$ and $w_0$
Exposing Parallelism

Warp divergence

\[ w_0 \quad w_1 \quad w_2 \quad w_3 \]

\[ w_0 \quad w_1 \quad w_2 \quad w_3 \]

\[ w_0 \quad w_1 \quad w_2 \quad w_3 \]

\[ w_0 \quad w_1 \quad w_2 \quad w_3 \]

\[ b_0 \quad b_1 \quad b_2 \quad b_3 \quad b_4 \quad b_5 \quad b_6 \quad b_7 \quad b_8 \quad b_9 \quad b_{10} \quad b_{11} \quad b_{12} \quad b_{13} \quad b_{14} \quad b_{15} \]
Exposing Parallelism

Warp divergence

\[ \begin{array}{c}
\text{w}_0 \\
\text{w}_1 \\
\text{w}_2 \\
\text{w}_3 \\
\text{w}_0 \\
\text{w}_1 \\
\text{w}_2 \\
\text{w}_3 \\
\text{w}_0 \\
\text{w}_1 \\
\text{w}_2 \\
\text{w}_3 \\
\text{b}_0 \\
\text{b}_1 \\
\text{b}_2 \\
\text{b}_3 \\
\text{b}_4 \\
\text{b}_5 \\
\text{b}_6 \\
\text{b}_7 \\
\text{b}_8 \\
\text{b}_9 \\
\text{b}_{10} \\
\text{b}_{11} \\
\text{b}_{12} \\
\text{b}_{13} \\
\text{b}_{14} \\
\text{b}_{15}
\end{array} \]
Main CUDA Features
Basic techniques and advanced optimizations

- Flexible kernels with grid-strided loops
- Fully coalesced global memory accesses
- Simple first dimension launch bounds
- Warp reduction using CUB
- Precomputed factors
- 32-bit index types
But hey, good news!
Full $O(p^3)$ M2M operator over the lowest level of a $d = 4$ octree, float

![Graph showing runtime vs. poles for CPU and GPU]
Oh wait...

Full $O(p^3)$ M2M operator over the lowest level of a $d = 4$ octree, float
Denormalization

Full $\mathcal{O}(p^3)$ M2M operator over the lowest level of a $d = 4$ octree, float

![Graph showing runtime versus multipole order for different CPU and GPU configurations. The graph indicates that GPU (NVIDIA Tesla K40m) is significantly faster than CPU (Intel Xeon E5-2650) with a speedup of 25.0x at a crossover multipole order p = 11.]
Outlook & Future Directions

Achievements
- M2M operator running on GPU with CUDA
- 4× speedup after many optimizations
- Minimum impact on existing code

Conclusions
- 100× speedups are often caused by a bad CPU implementation

Future work
- Deal with performance bottlenecks, e.g., register usage
- Explore the usage of shared memory in the kernels
- Architecture specific tuning
Thank you for your attention.

Questions?

Contact: Albert García
a.garcia@fz-juelich.de
Towards a Unified CPU/GPU Codebase
For A Linear Scaling FMM Coulomb Solver

April 7th, 2016 | Albert García | Ivo Kabadshow | Jülich Supercomputing Centre