JÜLICH
FORSCHUNGSZENTRUM

FMM goes GPU

A smooth trip or bumpy ride?

8. October, 2014 | B. Kohnke, I.Kabadshow | Jülich Supercomputing Centre

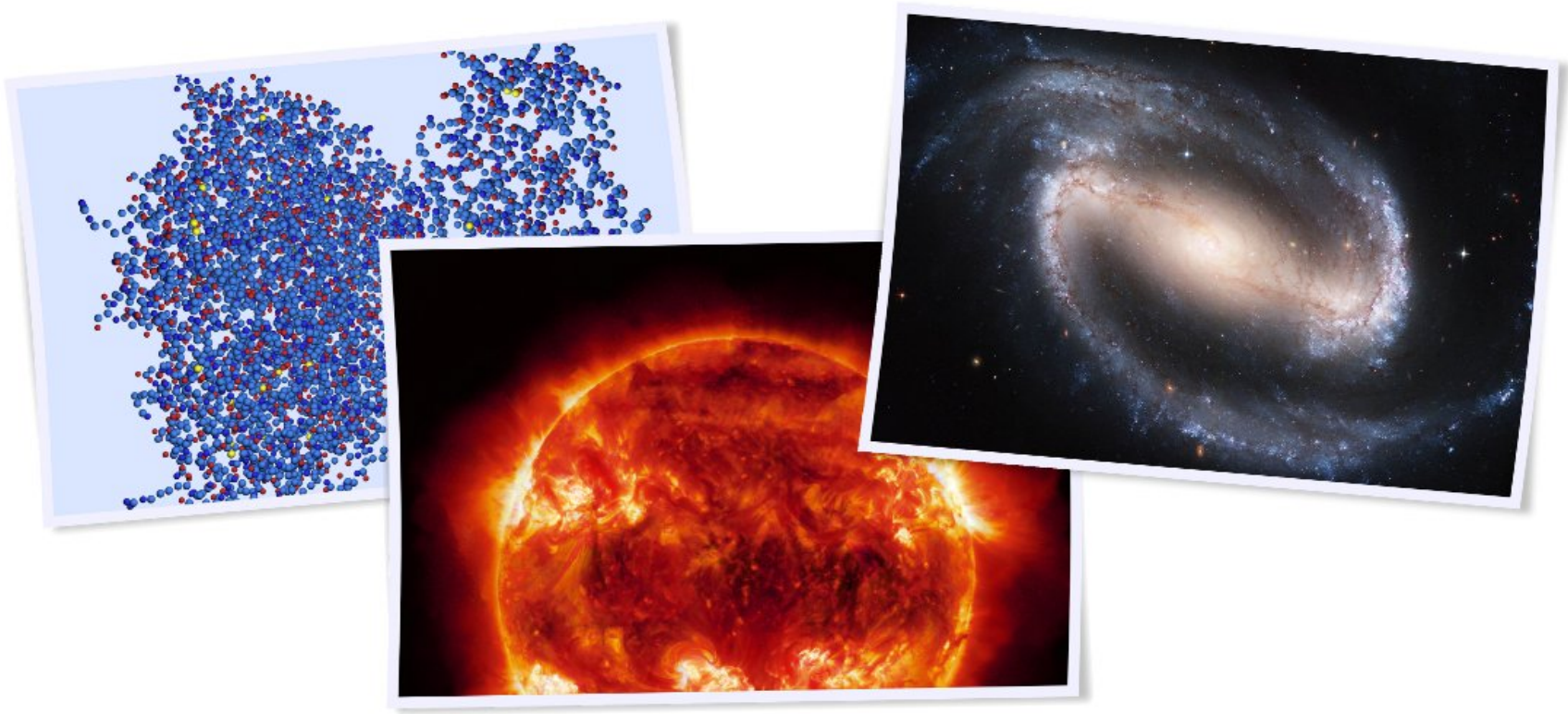
FMM goes GPU

A smooth trip or bumpy ride?

- Very short introduction to the FMM
- Memory allocation using unified memory
- FMM parallelization by example
  - Full parallelization of M2M operation
  - "Dynamic" approach
- Kernel comparison
- Conclusions

FMM Applications

$1/r$ Long-Range Interactions in $\mathcal{O}(N)$ instead of $\mathcal{O}(N^2)$



Molecular Dynamics | Plasma Physics | Astrophysics

Use FMM within GROMACS

DFG-funded SPPEXA Project: GROMEX

Partner

- MPI Göttingen (H. Grubmüller, C. Kutzner, Th. Ullmann, B. Kohnke)
- JSC (H. Dachsel, A. Beckmann, I. Kabadshow)
- KTH Stockholm (B. Hess)
- External GPU support: J. Kraus (NVidia)

Project goals

- FMM toolbox for exascale in MD
- Support heterogeneous platforms (CPU, GPU, MIC)
- λ-dynamics for MD

FMM goes GPU

Fast-forward mode

Catch-up with competitors

- Quickly find the real GPU bottlenecks
- Started with no initial modeling
- Coding $\rightarrow$ modeling $\rightarrow$ re-coding/re-design

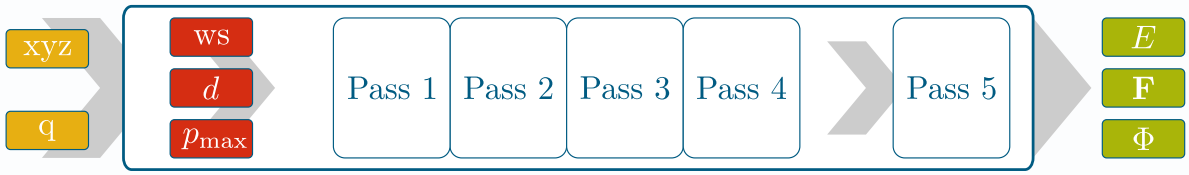
Code design

- Kernels written in templated C++
- NVIDIA GPU parallelization via CUDA
- No explicit on/offloading in first phase

Kick-Off was July 2014

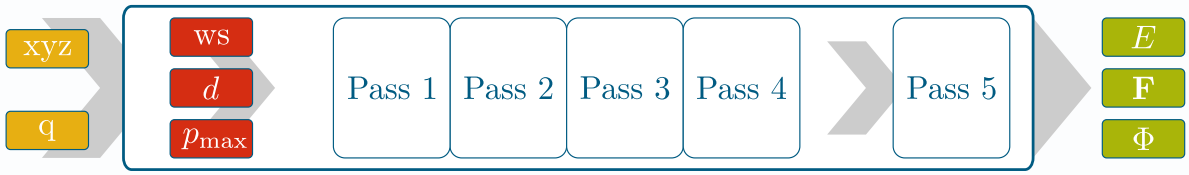
FMM Workflow

FMM Workflow



FMM Workflow

FMM Workflow



FMM in a Nutshell

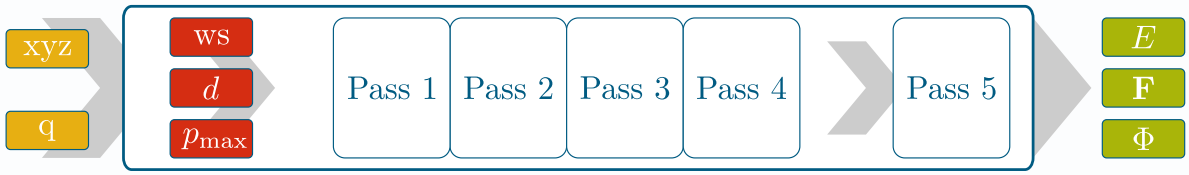
[do not try at home]

Many independent loops $\rightarrow$ intrinsic parallelism

$$E_C \approx \sum_{\text{level}} \sum_{\text{A}} \sum_{\text{B(A)}} \sum_{l=0}^{p} \sum_{m=-l}^{l} \sum_{j=0}^{p} \sum_{k=-j}^{j} (-1)^j \omega^{\text{A}}_{lm}(\mathbf{a}) M2L(\mathbf{R}) \omega^{\text{B}}_{jk}(\mathbf{b})$$

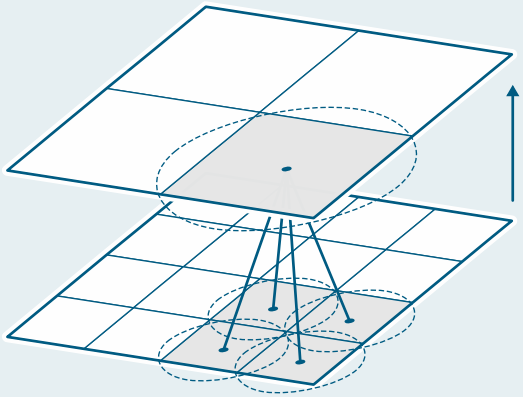
FMM Workflow

FMM Workflow



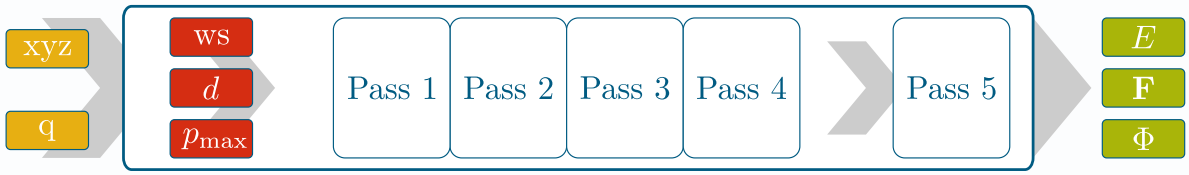
Pass 1

- Setup FMM tree and expand multipoles
- Shift multipoles to root node (M2M)



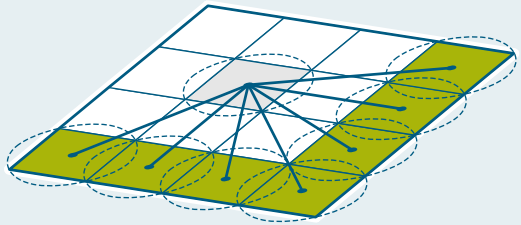
FMM Workflow

FMM Workflow



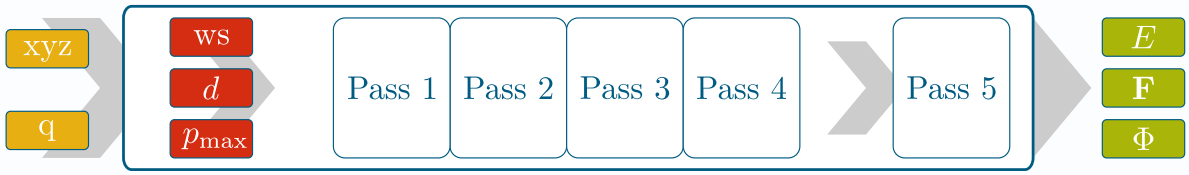
Pass 2

- Translate multipole moments into Taylor moments (M2L)



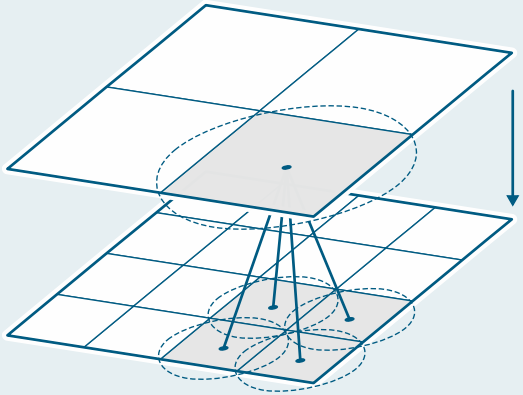
FMM Workflow

FMM Workflow



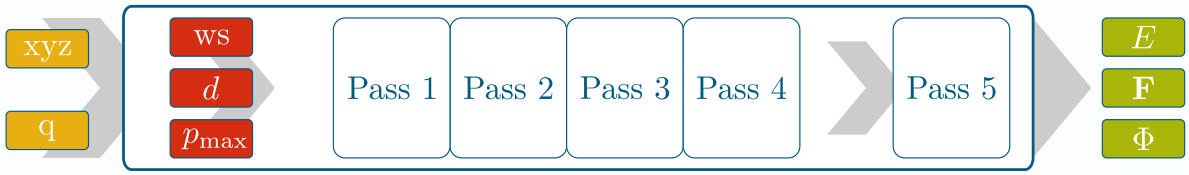
Pass 3

- Shift Taylor moments to the leaf nodes (L2L)



FMM Workflow

FMM Workflow

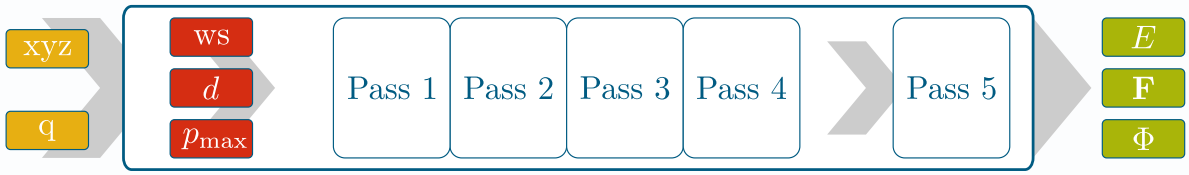


Pass 4

- Computation of the far field energy, forces, potentials

FMM Workflow

FMM Workflow



Pass 5

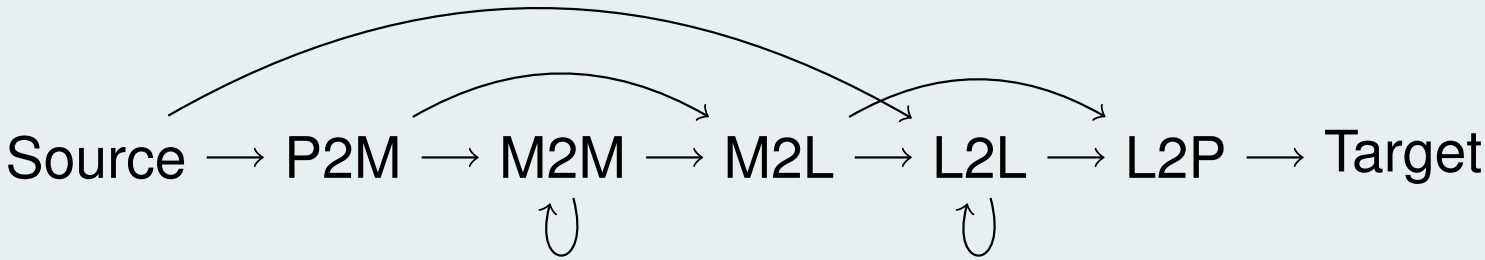
- Computation of the near field energy, forces, potentials

FMM Dependency Graph

Input, Output

- Input/Sources given by charge (q) and position (xyz)
- Output/Targets given by (xyz) and property (forces, potentials)

FMM Far field



FMM Near Field

Source → P2P → Target

FMM goes GPU

FMM goes GPU

A smooth trip or bumpy ride?

- Very short introduction to the FMM
- Memory allocation using unified memory
- FMM parallelization by example
  - Full parallelization of M2M operation
  - "Dynamic" approach
- Kernel comparison
- Conclusions

System Configuration

Juhydra @ JSC

- Intel Xeon CPU E5-2650 @ 2.00GHz
- NVIDIA Tesla K40m @ 745 MHz
  - 2880 CUDA cores
  - Single Precision Peak: 1.43 Tflops
  - Double Precision Peak: 4.29 Tflops

Unified Memory

Unified Memory

- Common address space for device and host
- One pointer shared by device and host
- Allocate memory only once

```
type *data;
cudaMallocManaged(&data,size);
host_function(data);
kernel<<<...>>>(data);
cudaDeviceSynchronize();
host_function(data);
```

Dynamic Parallelism

Dynamic Parallelism

- Dynamic Parallelism
- Starting a Kernel from a Kernel
- Dynamic computing grid resizing possible

FMM Datastructures

Farfield

Coefficient Matrix, Generalized Storage Type

- Multipole order p
- Matrix size $\mathcal{O}(p^2)$
- Triangular shape

Multipole Moments ω_{lm}

- Stored as coefficient matrix, size $\mathcal{O}(p^2)$

Operators, e.g. M2M

- Stored as coefficient matrix, size $\mathcal{O}(p^2)$

1-Element Shifting Operation

$$\omega_{lm}(\mathbf{a}+\mathbf{b}) = \sum_{j=0}^{l} \sum_{k=-j}^{j} \omega_{jk}(\mathbf{a}) O_{l-j,m-k}(b)$$

FMM – Preparing existing code for CUDA

Multipole moments – allocating memory

```
int num_boxes; //number of boxes in tree

//allocate memory for pointers
CoeffMatrix ** omega; = new CoeffMatrix *[num_boxes];
cudaMallocManaged(&omega, num_boxes*sizeof(CoeffMatrix*));

for (int id = 0; id < num_boxes; ++id) {
    int idx = compute_box_index(id);
    omega[idx] =  new CoeffMatrix(MULTIPOLEORDER);
}
```

- Using overloaded new operator
- Class CoeffMatrix inherits from CudaManaged class

Details on CudaManaged Class

Class CudaManaged

```
class CudaManaged{

    void *operator new (size_t len)
    {
        void *ptr;
        cudaMallocManaged(&ptr,len);
        return ptr;
    }

    void operator delete(void *ptr){
        cudaFree(ptr);
    }
};
```

FMM – Preparing existing code for CUDA

M2M-operator – allocating memory

```
int num_operators, depth; //tree depth
typedef M2M_Operator;
//allocate memory for pointers
M2M_Operator *** m2m_operators;
cudaMallocManaged(&m2m_operators,depth*sizeof(M2M_Operator*));
```

FMM – Preparing existing code for CUDA

M2M-operator – allocating memory

```
////allocate memory for operators at each depth
for (int d_c = depth; d_c > 0; --d_c) { //tree loop
    M2M_Operator **  m2m_operator;
    cudaMallocManaged(&m2m_operator,
                      num_operators*sizeof(M2M_Operator*));
    for (size_t ii = 0; ii <= 1; ++ii)
       for (size_t jj = 0; jj <= 1; ++jj)
          for (size_t kk = 0; kk <= 1; ++kk)
              int idx_op = compute_op_index(id);
              m2m_operator[idx_op] = new M2M_Operator(...);
    m2m_operators[d_c] = m2m_operator;
}
```

Memory allocation using Unified Memory

Pros

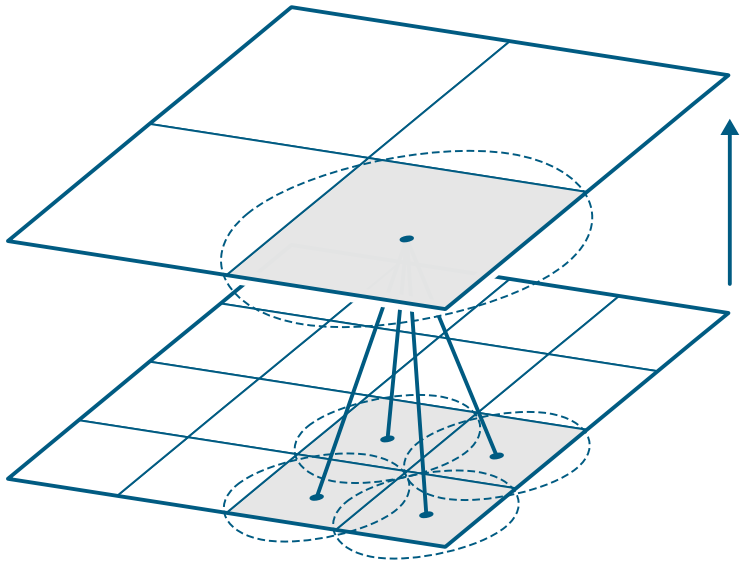
- Easy to use
- Fast porting of existing complex structures without a need to understand them

Cons

- Fast porting of existing complex structures without a need to understand them
- At the very beginning it can lead to superficial approach to problem

M2M Operation – Insights

- Going up the tree from lowest level
- Need to compute index of parent and child
- Need to compute index of translating operator



M2M Operation

$$\omega_{lm}(\mathbf{a}+\mathbf{b}) = \sum_{j=0}^{l} \sum_{k=-j}^{j} \omega_{jk}(\mathbf{a}) O_{l-j,m-k}(b)$$

M2M Operation – FMM Tree Loops

Parent – Child Loop structure

```
for (int d_c = depth; d_c > 0; --d_c) { // tree

 for (int i = 0; i < dim_p; ++i) { // parents
  for (int j = 0; j < dim_p; ++j) {
   for (int k = 0; k < dim_p; ++k) {

    int idx_p = parent_boxid(i, j, k);

     for (int ii = 0; ii <= 1; ++ii) { // children
      for (int jj = 0; jj <= 1; ++jj) {
       for (int kk = 0; kk <= 1; ++kk) {

        int idx_c = child_boxid(i, ii, j, jj, k, kk);
        int idx_op = operator_boxid(ii, jj, kk);

        // single p^4 operation
        M2M_operation(*omega[idx_c],
                      *m2m_operators[d_c][idx_op],
                      *omega[idx_p], p);
```

M2M Operation – Internal Structure

p^4 Loop structure

```
void M2M_operation(...){
  for (int l = 0; l <= p; ++l) {
    for (int m = 0; m <= l; ++m) {
      complex_type omega_l_m(0.);
      int j_max = f1(p, l);
      for (int j = 0; j <= j_max; ++j) {
        int k_min = f2(j, m, l);
        int k_max = f3(j, m, l);
        for (int k = k_min; k <= k_max; ++k) {
          omega_l_m += Operator.get(l - j, m - k) *
                       omega_in.get(j, k);
        }
      }
      omega_out(l, m) += omega_l_m;
    }
  }
}
```

M2M Operation – GPU Parallelization

Full parallel M2M-kernel

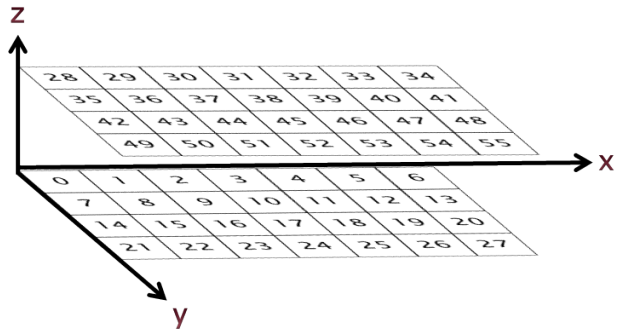
10 for-loops parallelized

- Loop over parent boxes: i, j, k
- Loop over child boxes: ii, jj, kk
- Inside M2M step ($\mathcal{O}(p^4)$ loops): l, m, j, k

Index Computation

- Compute a distinct position in the tree
- Child id and indices l,m,j,k from global thread index q

```
x = q%dim_x
y = q/dim_x
z = q/(dim_x*dim_y)
```



M2M Operation – GPU Parallelization

Index Computation

```
int x =  blockIdx.x * blockDim.x + threadIdx.x;
int y =  blockIdx.y * blockDim.y + threadIdx.y;
int z =  blockIdx.z * blockDim.z + threadIdx.z;

int z_range = gridDim.z*blockDim.z;
int y_range = gridDim.y*blockDim.y;

// compute global one-dimensional thread index
int q = x*y_range*z_range + y*z_range + z;
```

M2M Operation – Internal Structure (again)

p^4 Loop structure

```
// M2M_operation
  for (int l = 0; l <= p; ++l) {
    for (int m = 0; m <= l; ++m) {
      complex_type omega_l_m(0.);
      int j_max = f1(p, l);
      for (int j = 0; j <= j_max; ++j) {
        int k_min = f2(j, m, l);
        int k_max = f3(j, m, l);
        for (int k = k_min; k <= k_max; ++k) {
          omega_l_m += Operator.get(l - j, m - k) *
                       omega_in.get(j, k);
        }
      }
      omega_out(l, m) += omega_l_m;
    }
  }
```

M2M Operation – Internal Structure (again)

p^4 Loop structure

```
// M2M_operation
  for (int l = 0; l <= p; ++l) {
    for (int m = 0; m <= l; ++m) {
      complex_type omega_l_m(0.);
      int j_max = f1(p, l);
      for (int j = 0; j <= j_max; ++j) {
        int k_min = f2(j, m, l);
        int k_max = f3(j, m, l);
        int k=q%p; if(k>=k_min && k<=k_max) {
          omega_l_m += Operator.get(l - j, m - k) *
                       omega_in.get(j, k);
        }
      }
      omega_out(l, m) += omega_l_m;
    }
  }
```

M2M Operation – Internal Structure (again)

p^4 Loop structure

```
// M2M_operation
  for (int l = 0; l <= p; ++l) {
    for (int m = 0; m <= l; ++m) {
      complex_type omega_l_m(0.);
      int j_max = f1(p, l);
      for (int j = 0; j <= j_max; ++j) {
        int k_min = f2(j, m, l);
        int k_max = f3(j, m, l);
        int k=q%p; if(k>=k_min && k<=k_max) {
          omega_l_m += Operator.get(l - j, m - k) *
                       omega_in.get(j, k);
        }
      }
      omega_out(l, m) += omega_l_m;
    }
  }
```

M2M Operation – Internal Structure (again)

p^4 Loop structure

```
// M2M_operation
  for (int l = 0; l <= p; ++l) {
    for (int m = 0; m <= l; ++m) {
      complex_type omega_l_m(0.);
      int j_max = f1(p, l);
      int j = (q/p)%p; if(j<=j_max) {
        int k_min = f2(j, m, l);
        int k_max = f3(j, m, l);
        int k=q%p; if(k>=k_min && k<=k_max) {
          omega_l_m += Operator.get(l - j, m - k) *
                       omega_in.get(j, k);
        }
      }
      omega_out(l, m) += omega_l_m;
    }
  }
```

M2M Operation – Internal Structure (again)

p^4 Loop structure

```
// M2M_operation
  for (int l = 0; l <= p; ++l) {
    for (int m = 0; m <= l; ++m) {
      complex_type omega_l_m(0.);
      int j_max = f1(p, l);
      int j = (q/p)%p; if(j<=j_max) {
        int k_min = f2(j, m, l);
        int k_max = f3(j, m, l);
        int k=q%p; if(k>=k_min && k<=k_max) {
          omega_l_m += Operator.get(l - j, m - k) *
                       omega_in.get(j, k);
        }
      }
      omega_out(l, m) += omega_l_m;
    }
  }
```

M2M Operation – Internal Structure (again)

p^4 Loop structure

```
// M2M_operation
  int l = (q/p^3)%p; if(l<=p) {
    int m = (q/p^2)%p; if(m<=l) {
      complex_type omega_l_m(0.);
      int j_max = f1(p, l);
      int j = (q/p)%p; if(j<=j_max) {
        int k_min = f2(j, m, l);
        int k_max = f3(j, m, l);
        int k=q%p; if(k>=k_min && k<=k_max) {
          omega_l_m += Operator.get(l - j, m - k) *
                       omega_in.get(j, k);
        }
      }
      omega_out(l, m) += omega_l_m;
    }
  }
```

M2M Operation – Internal Structure (again)

p^4 Loop structure

```
// M2M_operation
  int l = (q/p^3)%p; if(l<=p) {
    int m = (q/p^2)%p; if(m<=l) {
      complex_type omega_l_m(0.);
      int j_max = f1(p, l);
      int j = (q/p)%p; if(j<=j_max) {
        int k_min = f2(j, m, l);
        int k_max = f3(j, m, l);
        int k=q%p; if(k>=k_min && k<=k_max) {
          omega_l_m += Operator.get(l - j, m - k) *
                       omega_in.get(j, k);
        }
      }
      omega_out(l, m) += omega_l_m; // atomic operation
    }
  }
```

M2M Operation Inside the Tree

Loop over child and parent boxes

```
for (int d_c = depth; d_c > 0; --d_c) { //tree

 for (int i = 0; i < d_p; ++i) {
  for (int j = 0; j < d_p; ++j) {
   for (int k = 0; k < d_p; ++k) {

    int idx_p = parent_boxid(i, j, k);

     for (int ii = 0; ii <= 1; ++ii) {
      for (int jj = 0; jj <= 1; ++jj) {
       for (int kk = 0; kk <= 1; ++kk) {

        int idx_c = child_boxid(i, ii, j, jj, k, kk);
        int idx_op = operator_boxid(ii, jj, kk);

        // p  multipole order (p^4 operation)
        M2M_operation ...;
```

M2M Operation Inside the Tree

Loop over child and parent boxes

```
for (int d_c = depth; d_c > 0; --d_c) { //tree

 int i = (q/p^4*8*d_p^3)%d_p {
  int j = (q/p^4*8*d_p^2)%d_p {
   int k = (q/p^4*8*d_p)%d_p {

    int idx_p = parent_boxid(i, j, k);

     int ii = (q/p^4*4)%2 {
      int jj = (q/p^4*2)%2 {
       int kk = (q/p^4)%2 {

        int idx_c = child_boxid(i, ii, j, jj, k, kk);
        int idx_op = operator_boxid(ii, jj, kk);

        // p  multipole order (p^4 operation)
        M2M_operation ...;
```

M2M Operation – Advanced Features

Reduction: Map $k_{min} - k_{max}$ loop to threadIdx.x

Classical Reduction

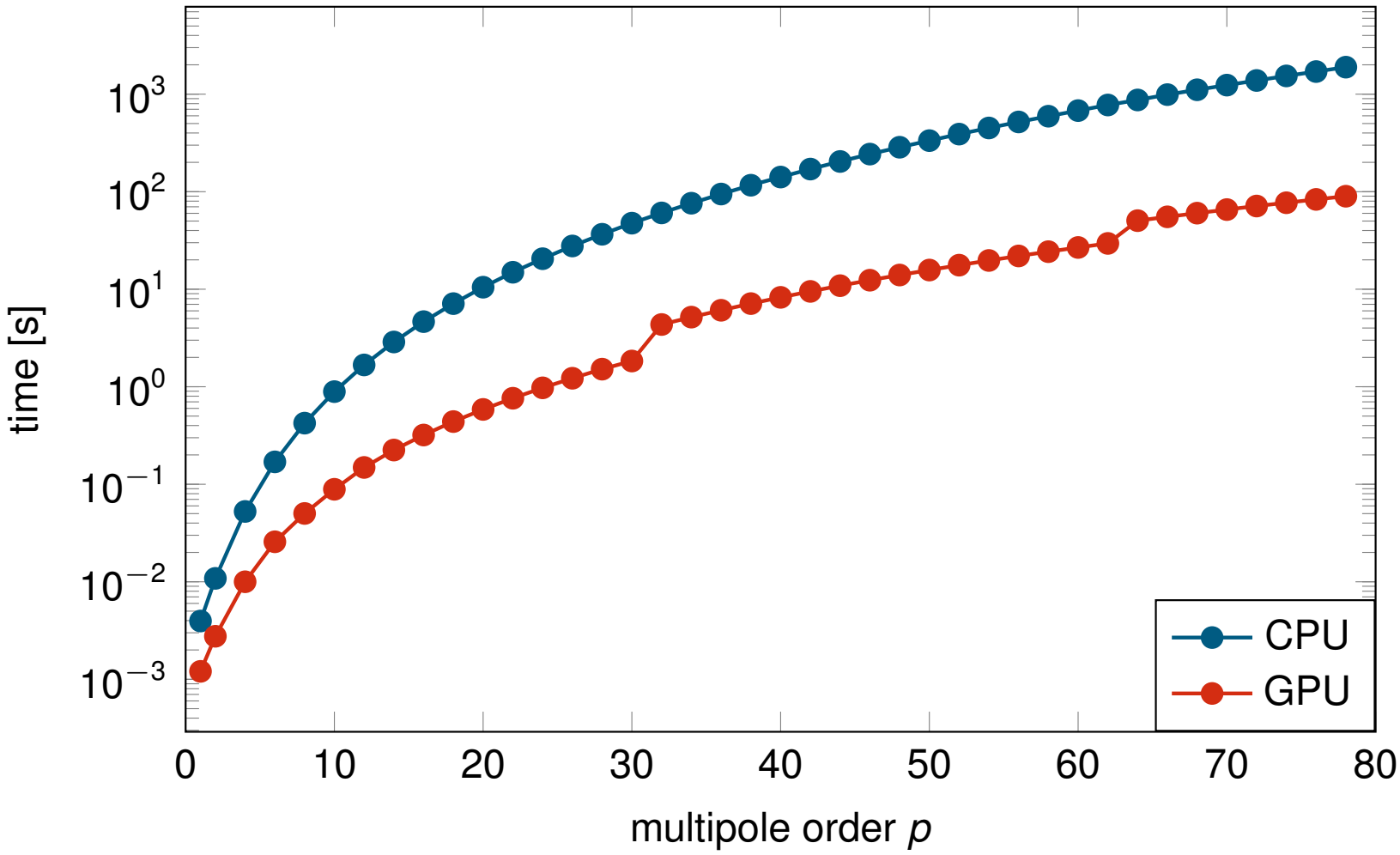
```
if(k>=k_min && k<=k_max){
    omega_l_m += Operator.get(l - j, m - k){
                 *
                 omega_in.get(j, k);
}}
```

Warp Reduction

```
if(k <= k_max){
  my_val = Operator.get(l - j, m - k)* omega_in.get(j, k);
}
omega_temp = WarpReduce(temp[warp_id]).Sum(my_val);

if(lane_id%32==0)
*omega_out(l, m) += omega_temp;
```

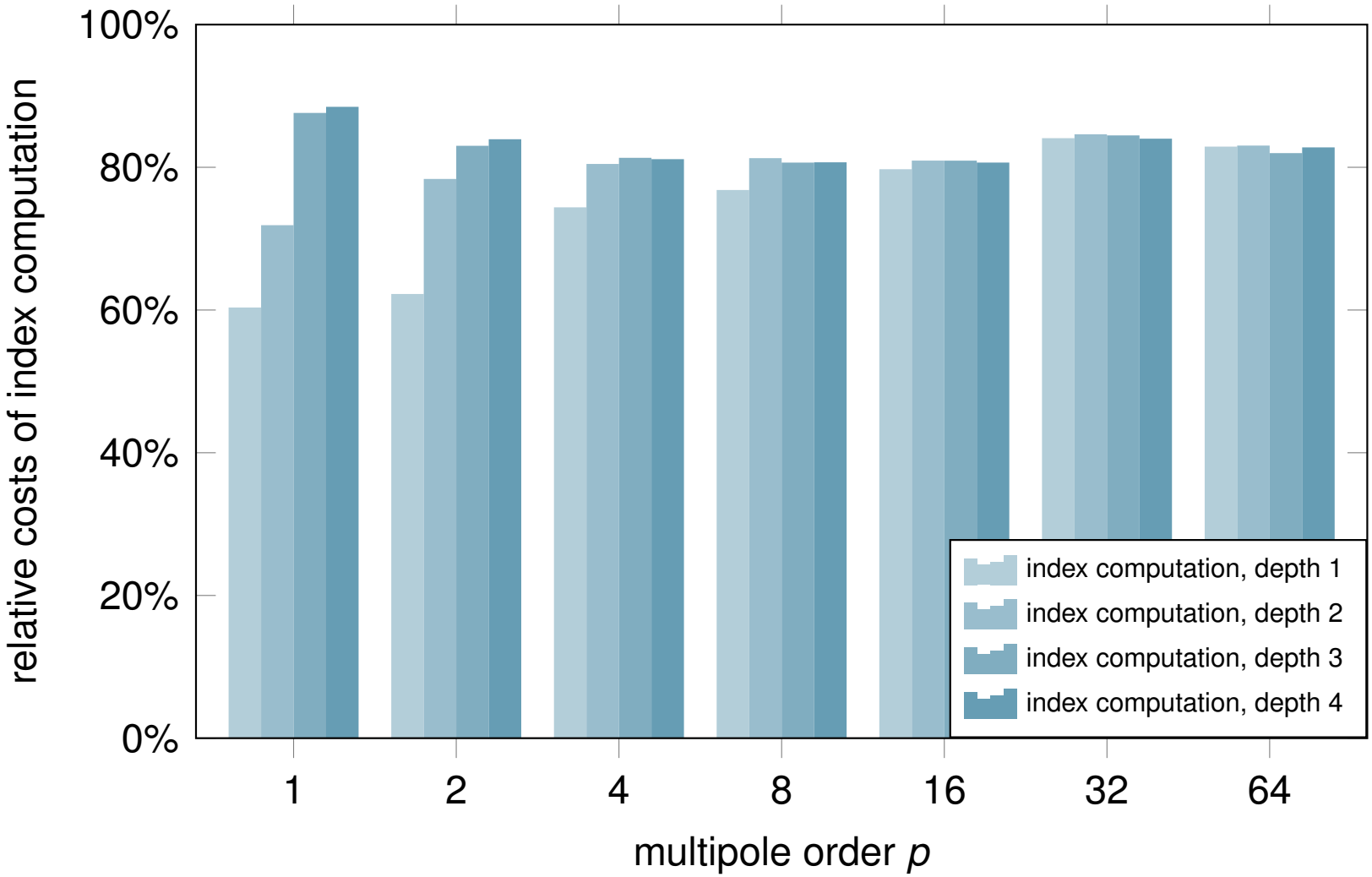
M2M – CPU vs. GPU

First Version, M2M Full Parallel, Depth 4



Additional Parallelization Costs

Relative costs of index computing operations for M2M full parallel kernel



Dynamic Parallelization

Problem – Redundancy

- each thread of parallelized p^4 does index computation
- time consuming
- redundant

Idea

- Use dynamic parallelism to avoid redundant index computation
- Pass computed index to the child kernel
- Child kernel performes the p^4 operation

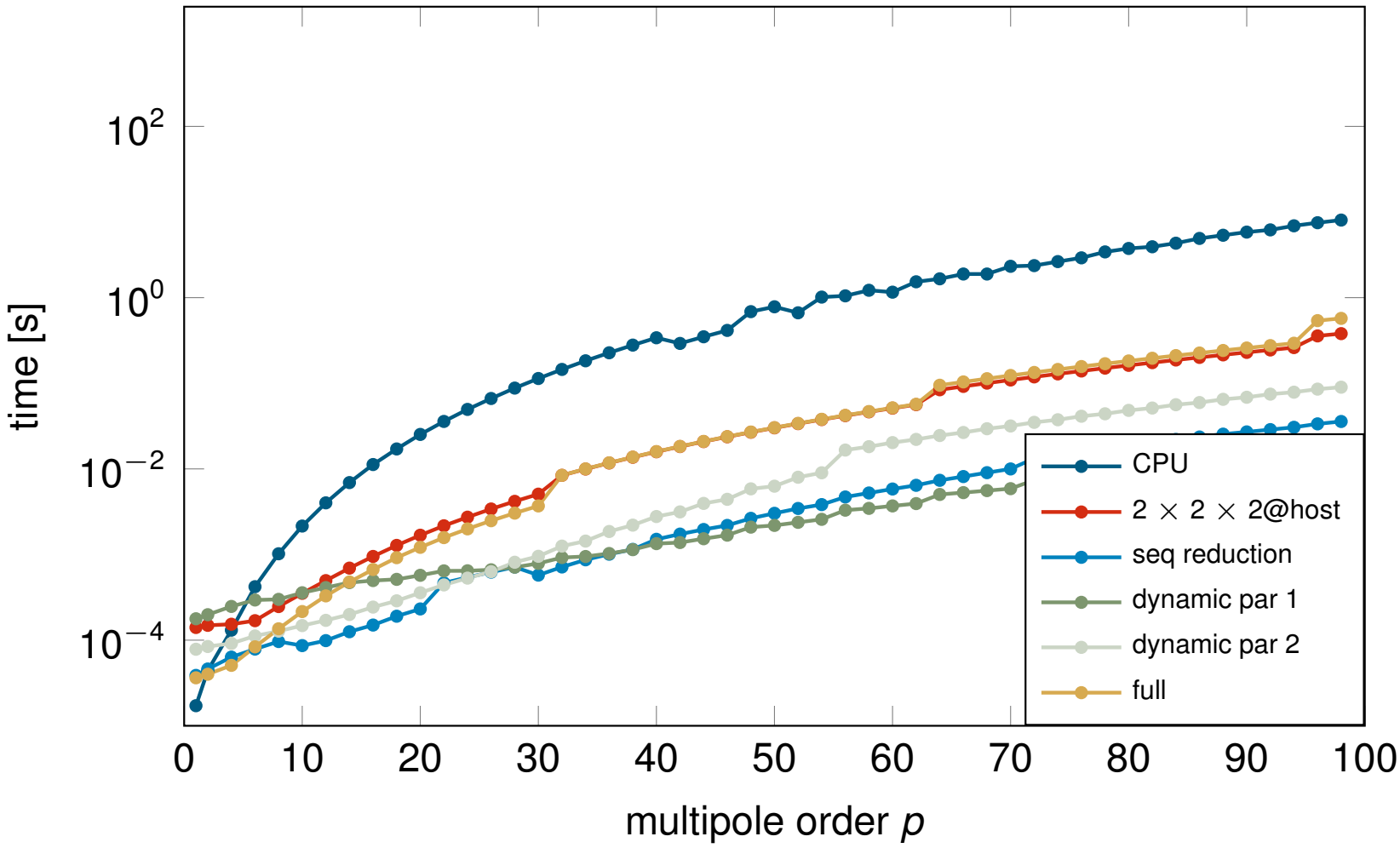
Dynamic Parallelization

M2M dynamic parent kernel

```
//compute parent i,j,k position in tree

int idx_p = make_parent_boxid(i,j,k);

//loop over nearest neighbors
for (int ii = 0; ii <= 1; ++ii) {
  for (int jj = 0; jj <= 1; ++jj) {
    for (int kk = 0; kk <= 1; ++kk) {

      int idx_c =  make_child_boxid(i, ii,j, jj, k, kk);
      int idx_op = make_op_boxid(ii,jj,kk);

      dim3 blockDim(...);
      dim3 gridDim(...);

      M2Mkernel_child
      <<<gridDim,blockDim>>>
      (Operator[depth],omega,idx_p,idx_c,idx_op);
```

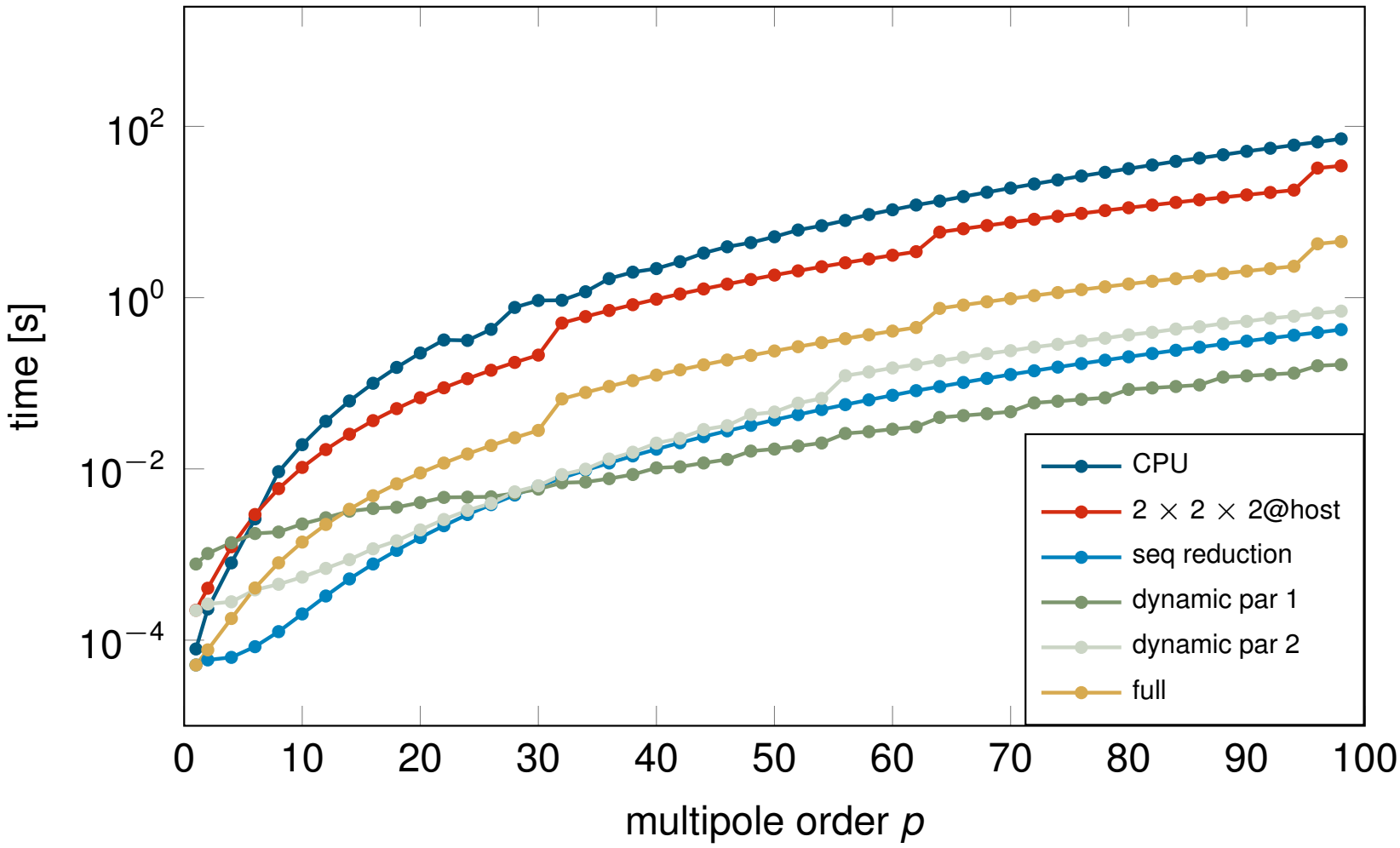
M2M – Different Parallelization Strategies

Depth 1



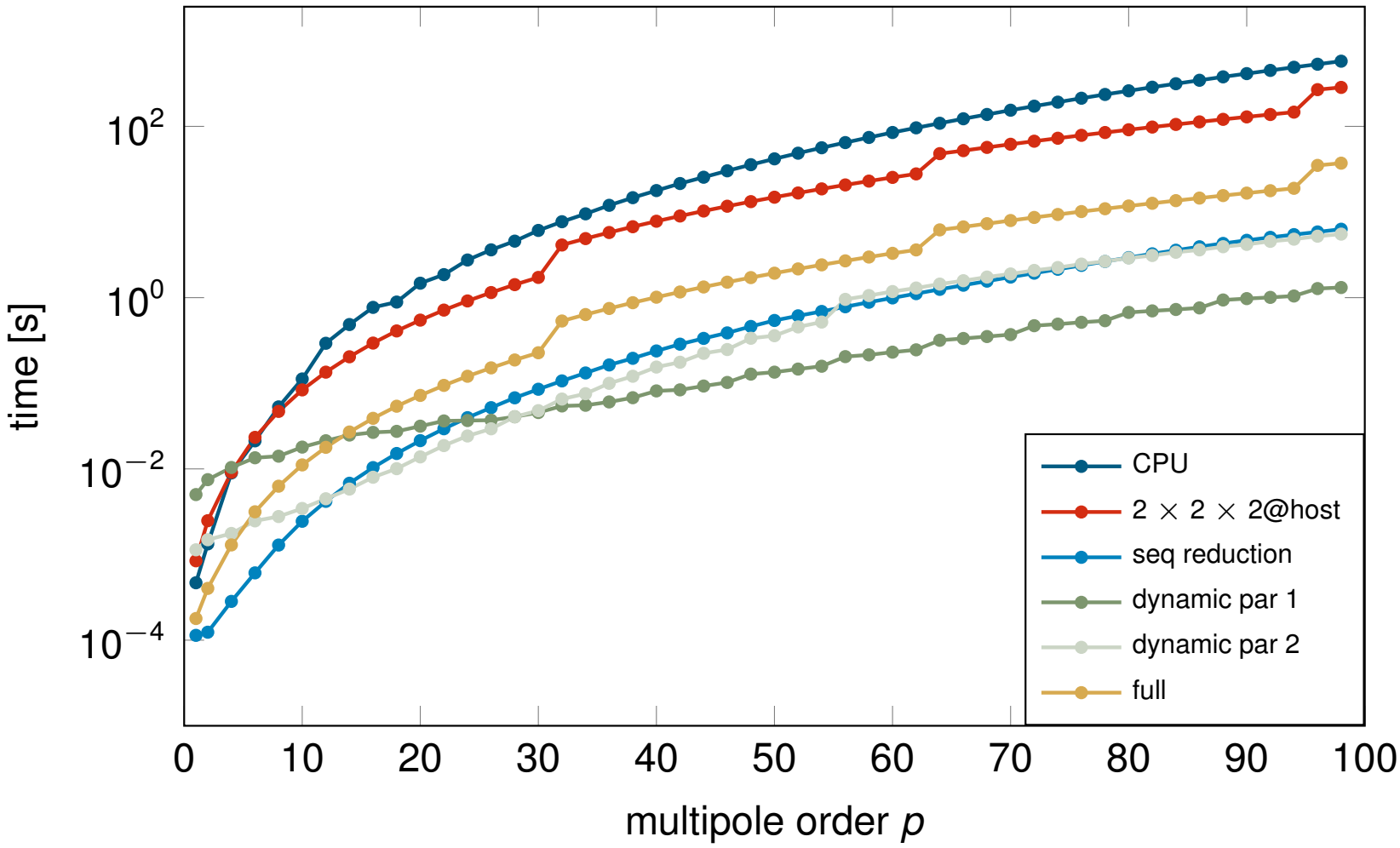
M2M – Different Parallelization Strategies

Depth 2



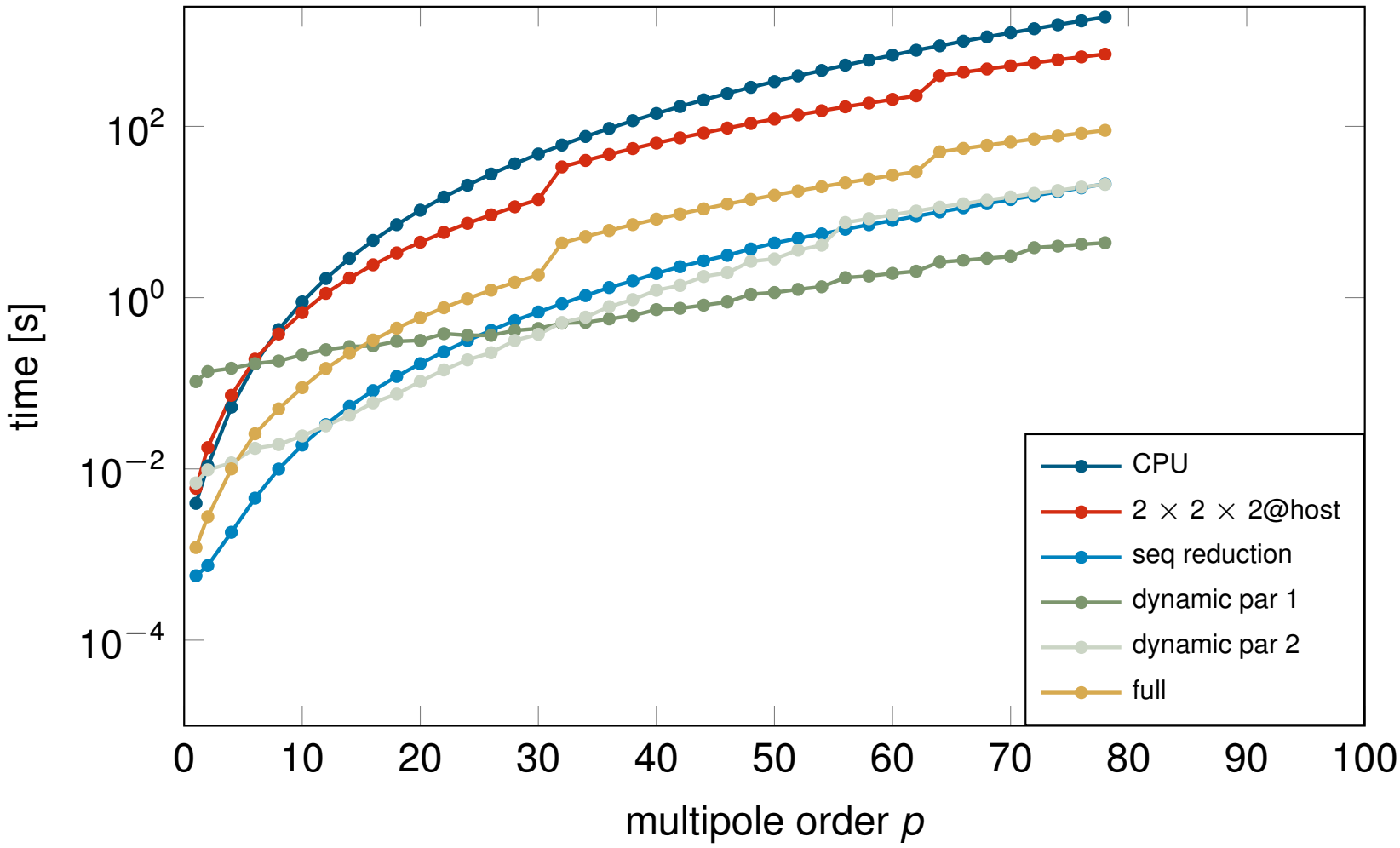
M2M – Different Parallelization Strategies

Depth 3



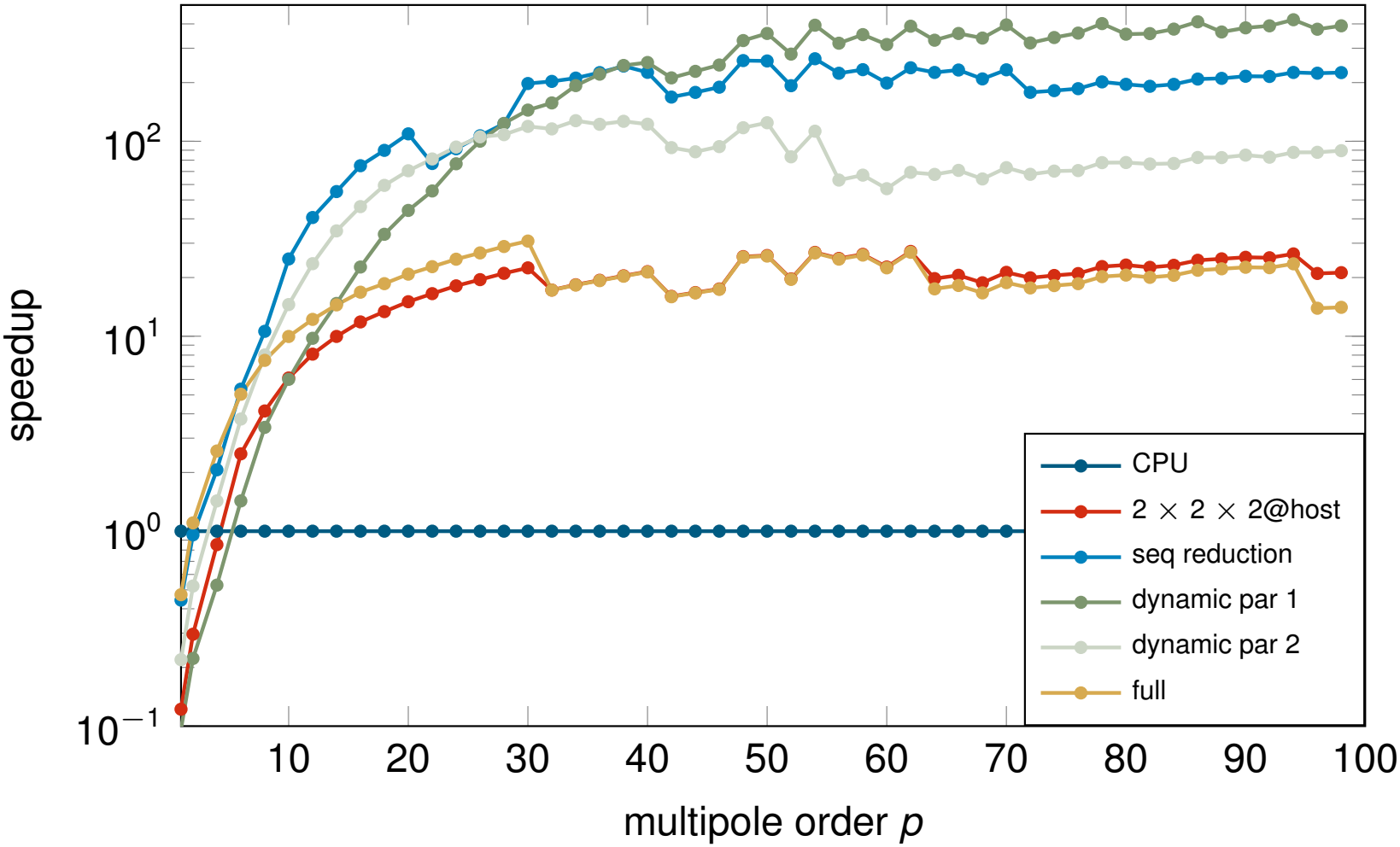
M2M – Different Parallelization Strategies

Depth 4



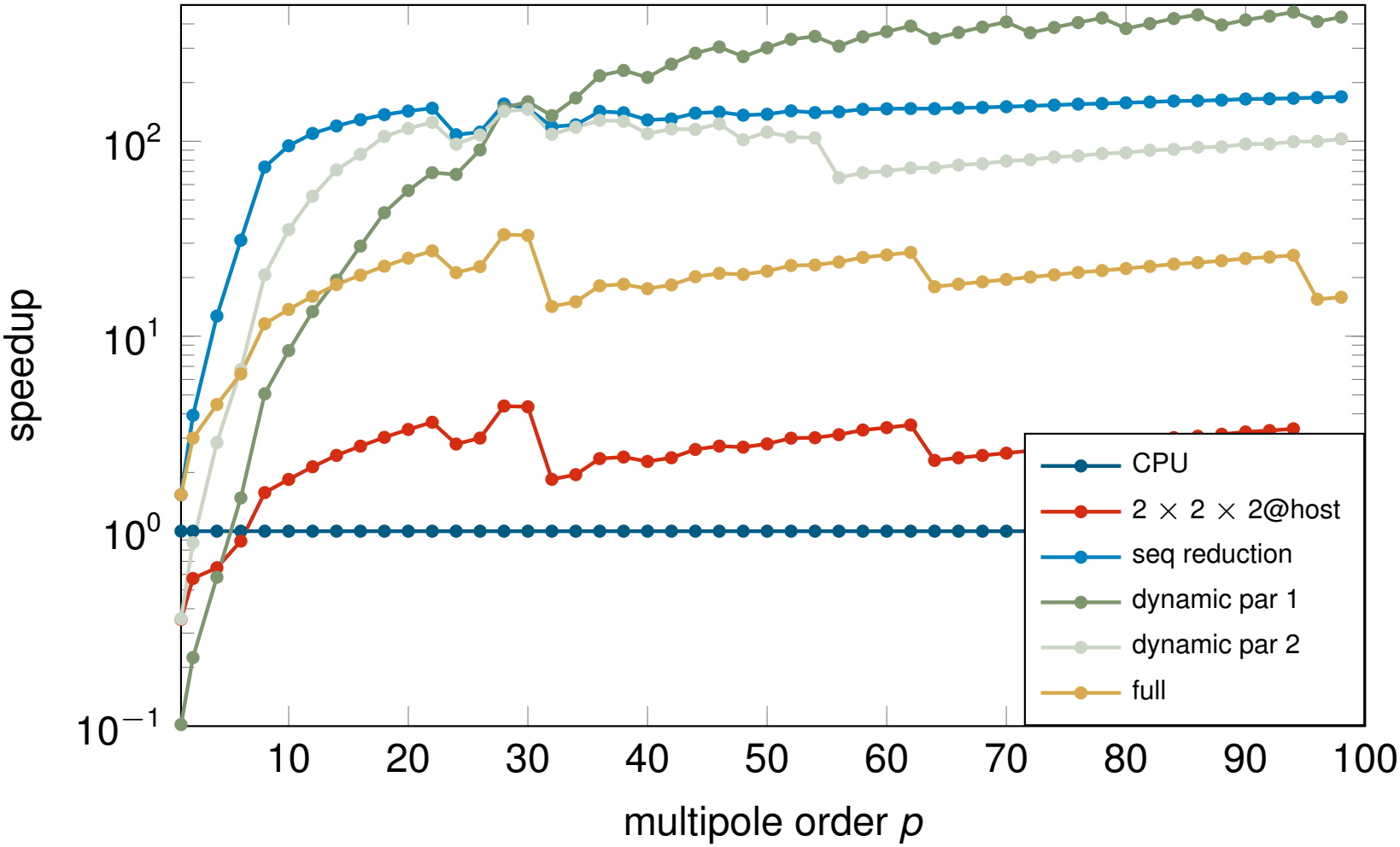
M2M – Different Parallelization Strategies

Depth 1, Speedup 1 GPU vs. 1 CPU core



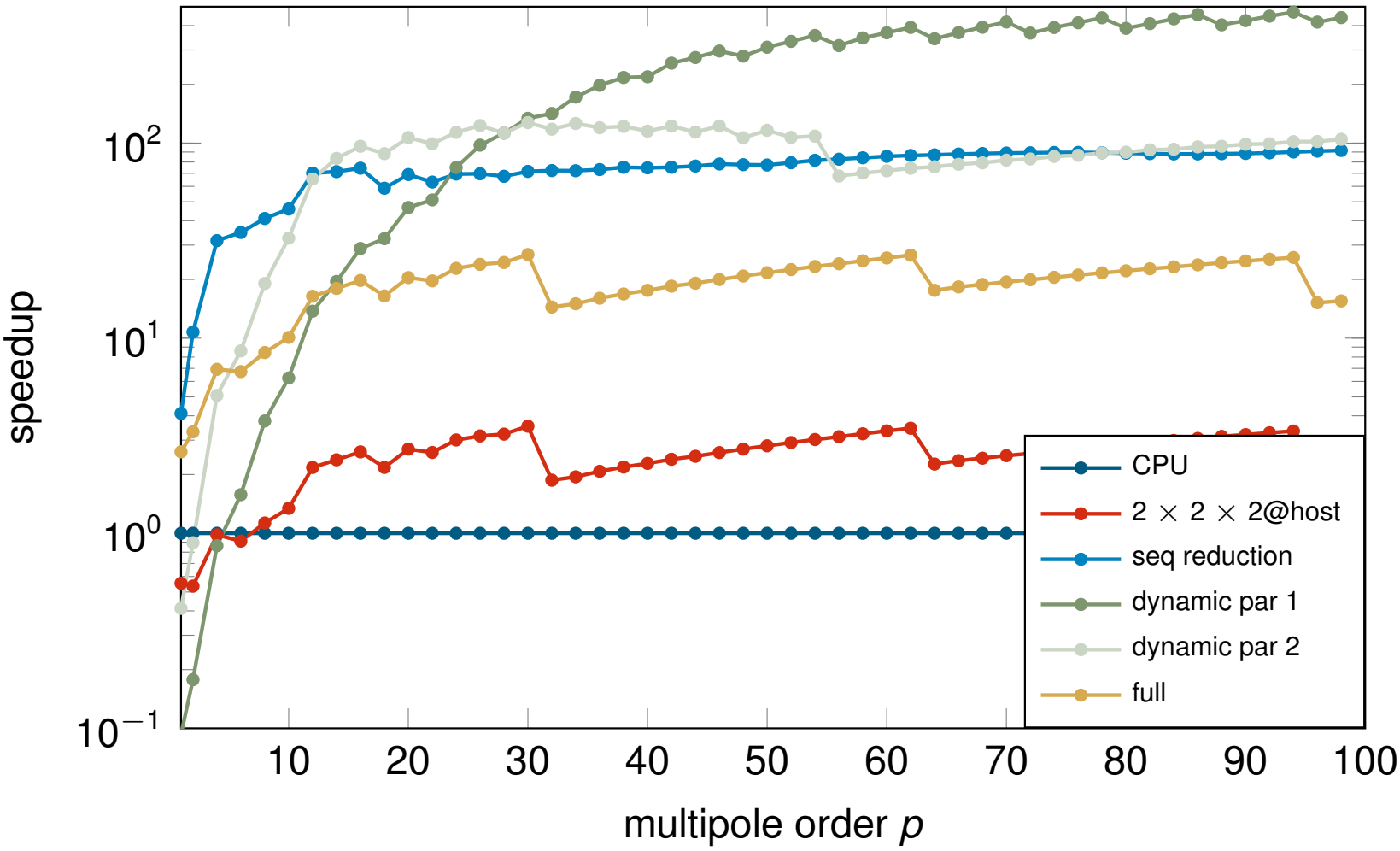
M2M – Different Parallelization Strategies

Depth 2, Speedup 1 GPU vs. 1 CPU core



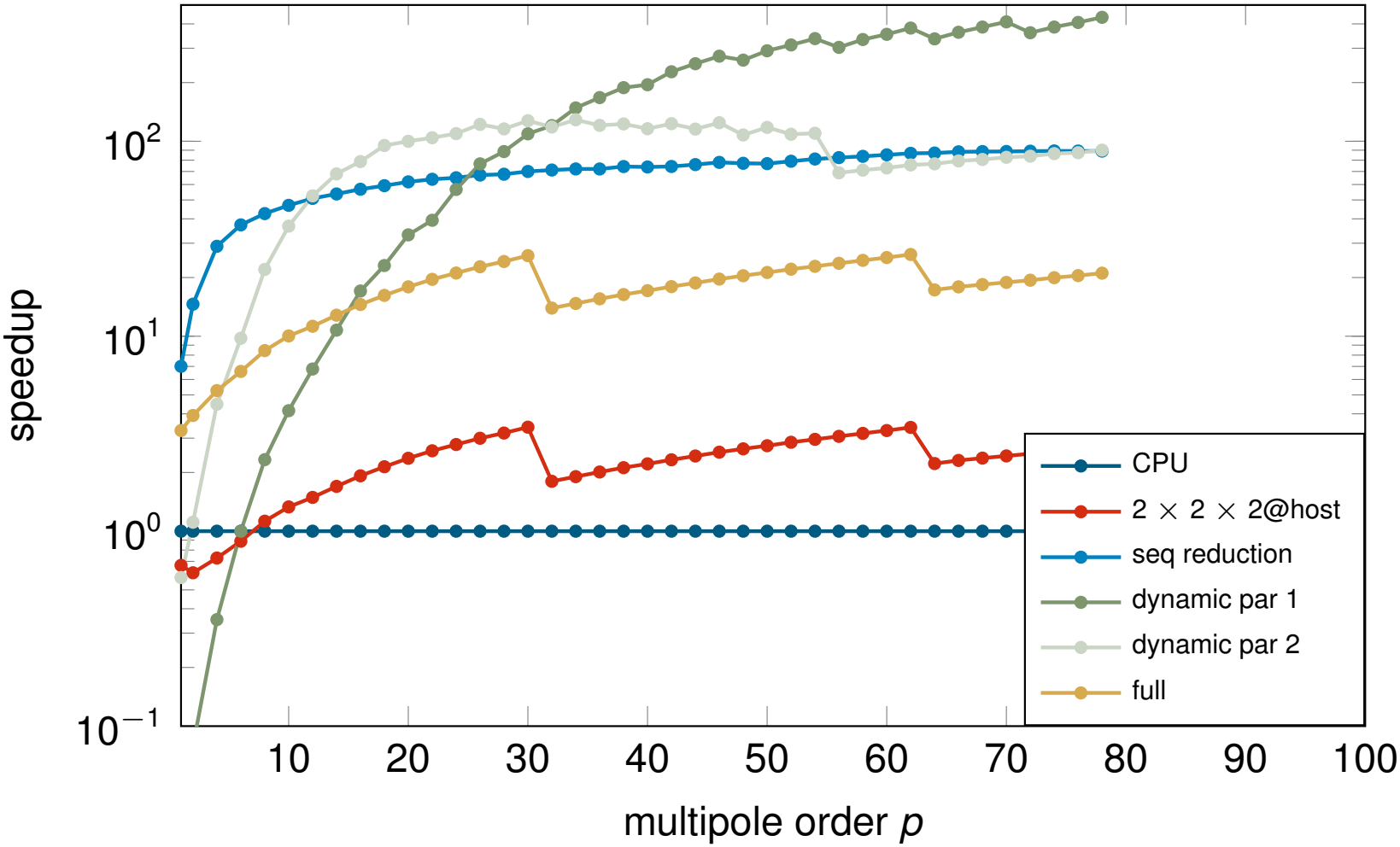
M2M – Different Parallelization Strategies

Depth 3, Speedup 1 GPU vs. 1 CPU core



M2M – Different Parallelization Strategies

Depth 4, Speedup 1 GPU vs. 1 CPU core



Conclusion

The Smooth Part

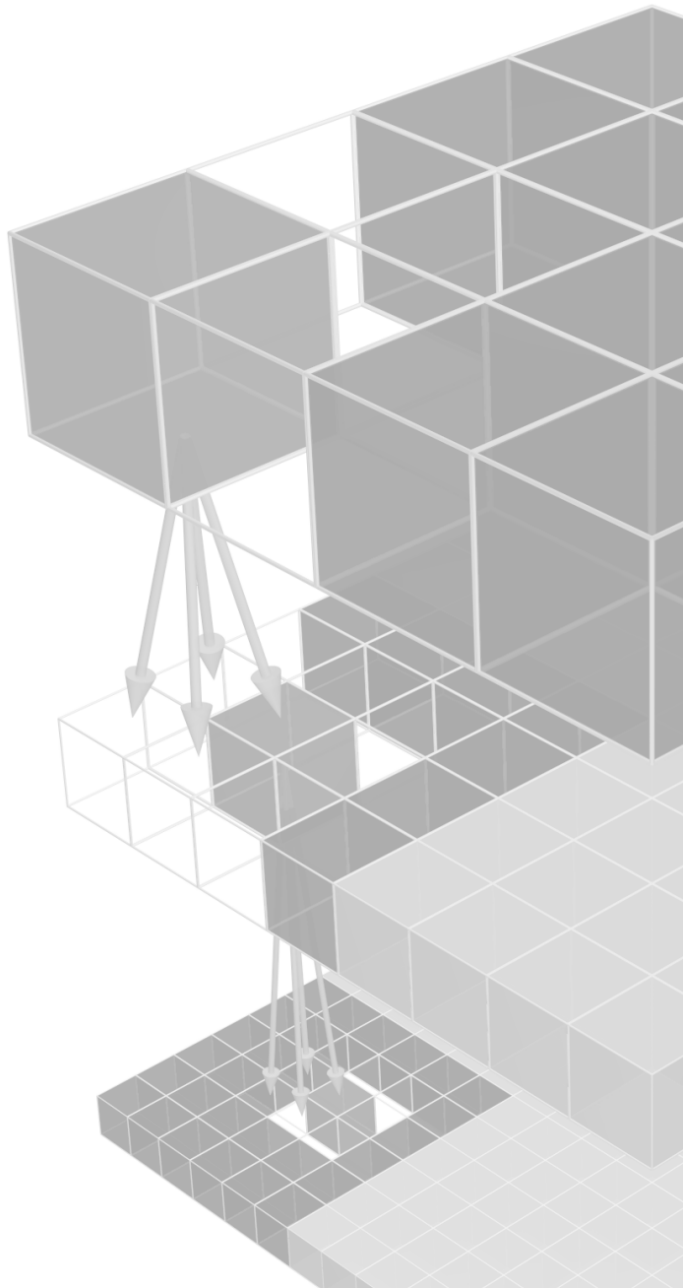
- Easy memory management with unified memory
- Dynamic parallelization helps a lot
- Easy FMM tree parallelization
- $\mathcal{O}(p^4)$ operator
  - operators (M2M, M2L, L2L) are simple and alike
  - inner kernel: only one complex multiplication
  - outer kernel: four independent loops
  - kernel: $\mathcal{O}(p^2)$ independent reductions

The Bumpy Part

- index computation takes time (developer and HW)
- $\mathcal{O}(p^3)$ vs. $\mathcal{O}(p^4)$ operator GPU parallelization benefit?

JÜLICH
FORSCHUNGSZENTRUM

Questions?



JÜLICH
FORSCHUNGSZENTRUM

FMM goes GPU

A smooth trip or bumpy ride?

8. October, 2014 | B. Kohnke, I.Kabadshow | Jülich Supercomputing Centre