Scientific computing on accelerator-based supercomputers

Xing Cai

Simula Research Laboratory & University of Oslo

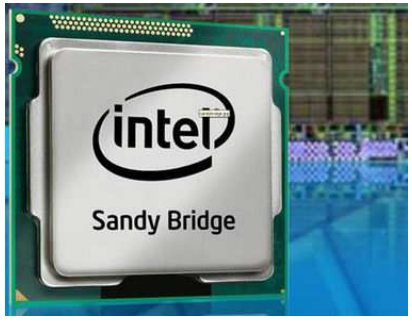
FFI, 2013.09.20

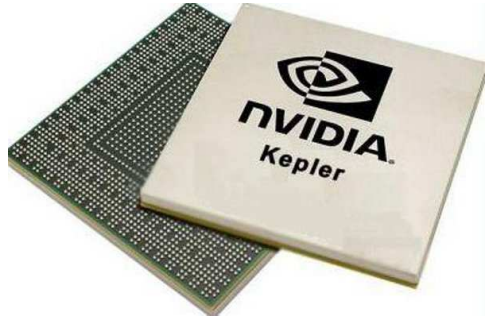
Outline

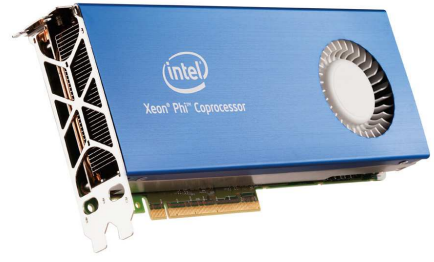
- Motivation
- Bits & pieces
  - Using GPUs
  - Using Xeon Phi coprocessors
- Realistic applications

Why using accelerators?

Hardware	Peak DP rate	Peak memory BW
Intel Xeon E5-2650 8-core GPU	128 GFlop/s	51 GB/s
NVIDIA Kepler GK110 GPU	1170 GFlop/s	208 GB/s
Intel Xeon Phi 5110P coprocessor	1011 GFlop/s	320 GB/s

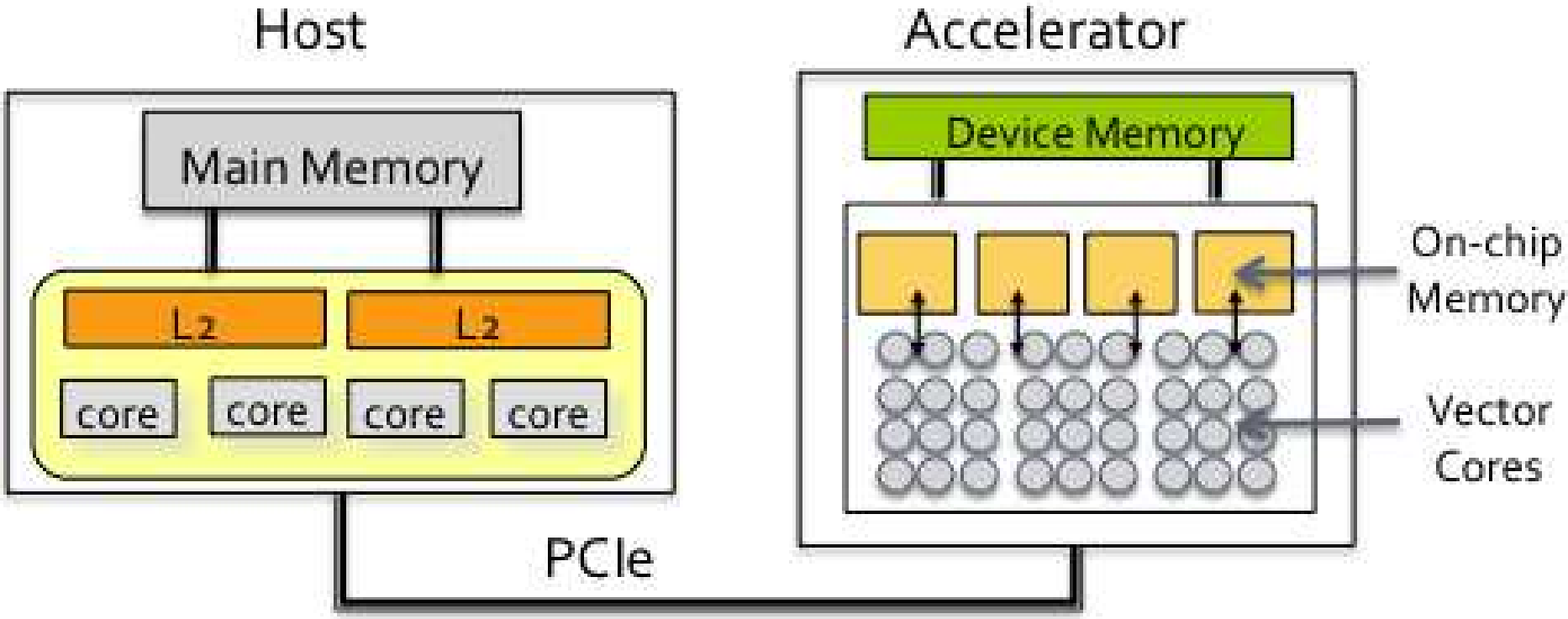






- Accelerators have tremendous computing power, but requires careful usage

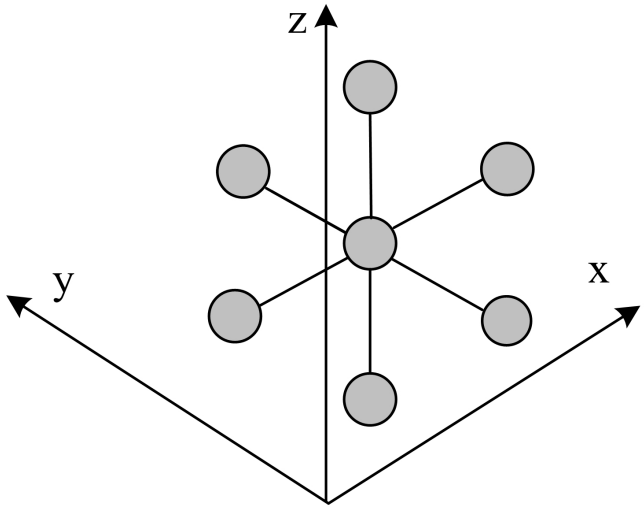
A conceptual picture



A simple numerical benchmark

Solving a 3D heat equation by explicit finite differences:

$$\frac{u_{i,j,k}^{\text{new}} - u_{i,j,k}^{\text{old}}}{\Delta t} = \frac{u_{i,j,k-1}^{\text{old}} + u_{i,j-1,k}^{\text{old}} + u_{i-1,j,k}^{\text{old}} - 6u_{i,j,k}^{\text{old}} + u_{i+1,j,k}^{\text{old}} + u_{i,j+1,k}^{\text{old}} + u_{i,j,k+1}^{\text{old}}}{h^2}$$



Heat 3D 7 point stencil

```
while(iters < max_ iters)
{
    for (k=1; k<z-1; k++)
        for (j=1; j<y-1; j++)
            for (i=1; i<x-1; i++)
            {
                u_new[k][j][i] = alpha * u_old[k][j][i] + beta * (
                                + u_old[k][j][i-1] + u_old[k][j][i+1]
                                + u_old[k][j-1][i] + u_old[k][j+1][i]
                                + u_old[k+1][j][i] + u_old[k-1][j][i]);
            }
}
```

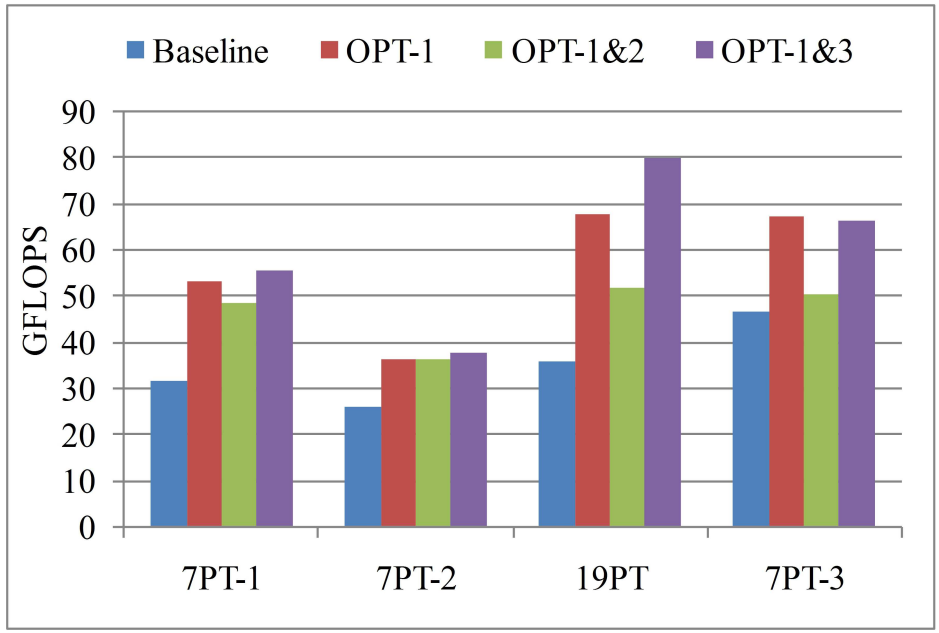
Baseline CUDA implementation

```
__global__ void stencil (double * device_u, double * device_u_new,
                         double alpha, double beta, int Nx, int Ny)
{
    int gid_x = blockIdx.x*blockDim.x + threadIdx.x + 1;
    int gid_y = blockIdx.y*blockDim.y + threadIdx.y + 1;
    int gid_z = blockIdx.z*blockDim.z + threadIdx.z + 1;
    double (*in)[Ny][Nx];
    double (*out)[Ny][Nx];
    in  = (double (*)[Ny][Nx])device_u;
    out = (double (*)[Ny][Nx])device_u_new;
    out[gid_z][gid_y][gid_x]=(alpha*in[gid_z][gid_y][gid_x])+
                             beta*(in[gid_z][gid_y][gid_x-1]
                                  +in[gid_z][gid_y][gid_x+1]
                                  +in[gid_z][gid_y-1][gid_x]
                                  +in[gid_z][gid_y+1][gid_x]
                                  +in[gid_z-1][gid_y][gid_x]
                                  +in[gid_z+1][gid_y][gid_x]);
}
```

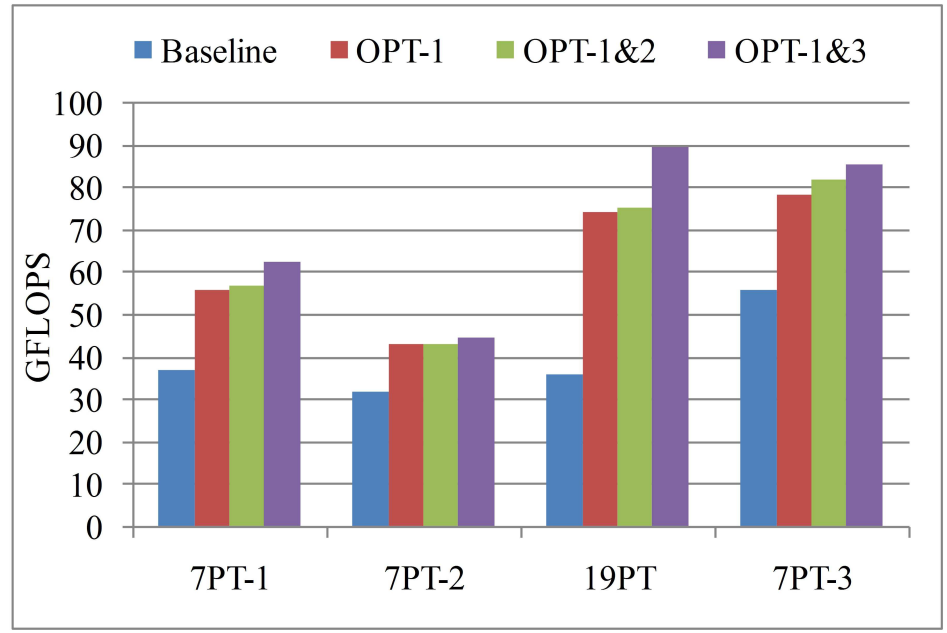
- One mesh point is computed by one extremely lightweight thread

Improving the GPU performance

- Typical performance-enhancing strategies:
  - OPT-1: Let each thread compute a z-column of mesh points
  - OPT-2: Use the on-chip shared memory
  - OPT-3: Chunking in the y-direction



GTX590 GPU



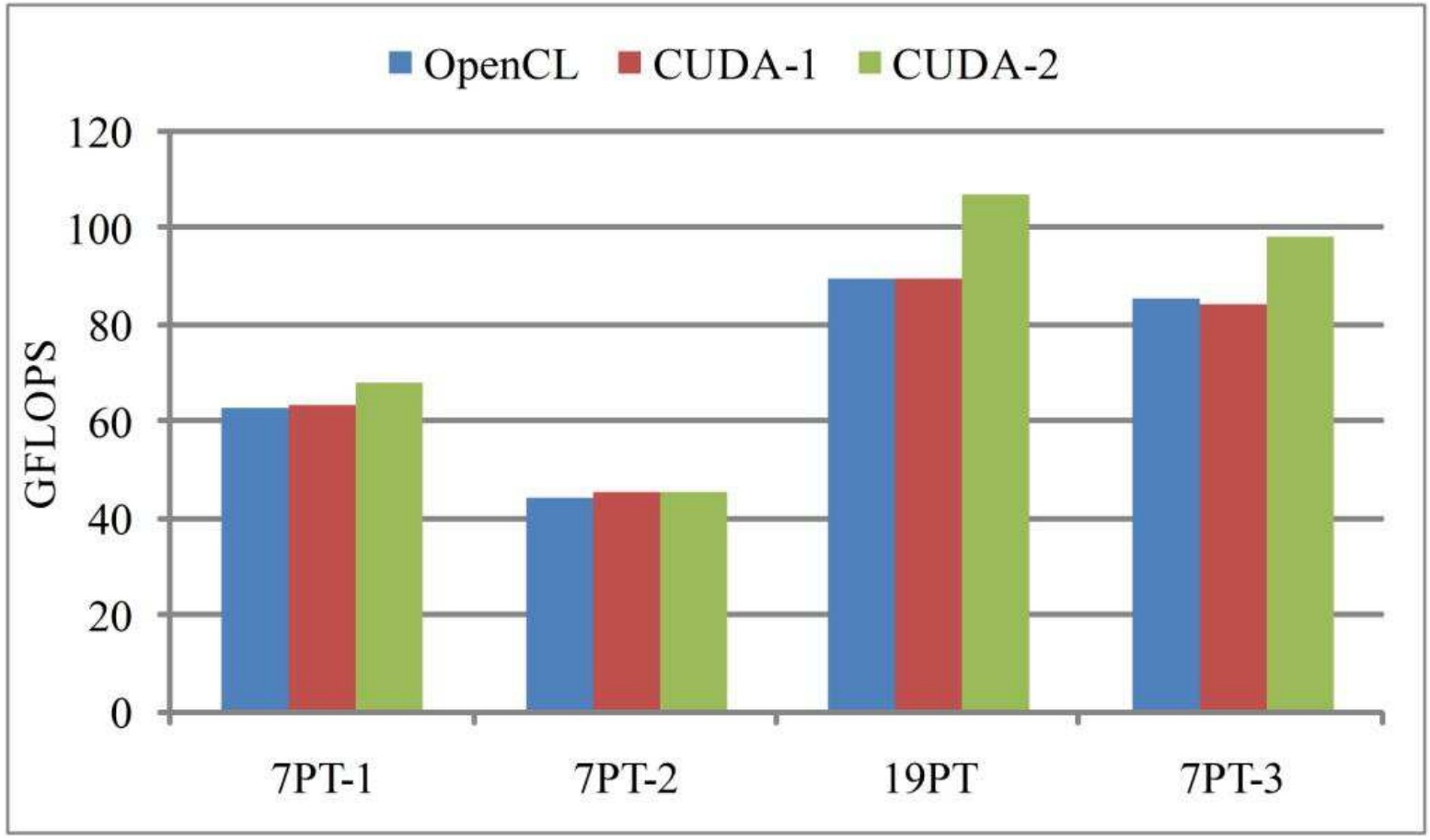
K20 GPU

OpenCL programming

```
__kernel void stencil(__global double * device_u,
                      __global double * device_u_new,
                      double alpha, double beta,
                      int Nx, int Ny)
{
    int gid_x = get_global_id(0)+1;
    int gid_y = get_global_id(1)+1;
    int gid_z = get_global_id(2)+1;
    __global double (*in)[Ny][Nx];
    __global double (*out)[Ny][Nx];
    in  = (__global double (*)[Ny][Nx])device_u;
    out = (__global double (*)[Ny][Nx])device_u_new;
    out[gid_z][gid_y][gid_x]=(alpha*in[gid_z][gid_y][gid_x])+
                             beta*(in[gid_z][gid_y][gid_x-1]
                                  +in[gid_z][gid_y][gid_x+1]
                                  +in[gid_z][gid_y-1][gid_x]
                                  +in[gid_z][gid_y+1][gid_x]
                                  +in[gid_z-1][gid_y][gid_x]
                                  +in[gid_z+1][gid_y][gid_x]);
}
```

- OpenCL programming is very similar to CUDA programming

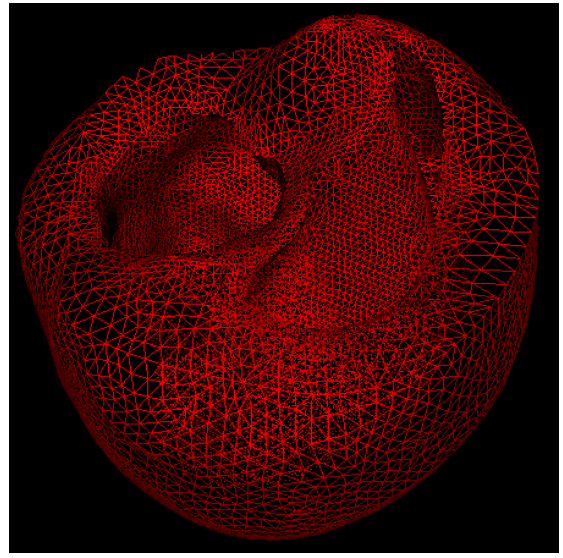
CUDA vs. OpenCL



Performance comparison between OpenCL and CUDA on a K20 GPU

- OpenCL can give fully comparable performance against CUDA
- Advantage of using CUDA on Kepler GPUs: read-only data cache

GPU performance & unstructured mesh



- Example: cell-centered FVM on a 3D tetrahedral mesh
  - 11 floating-point operations per tetrahedron

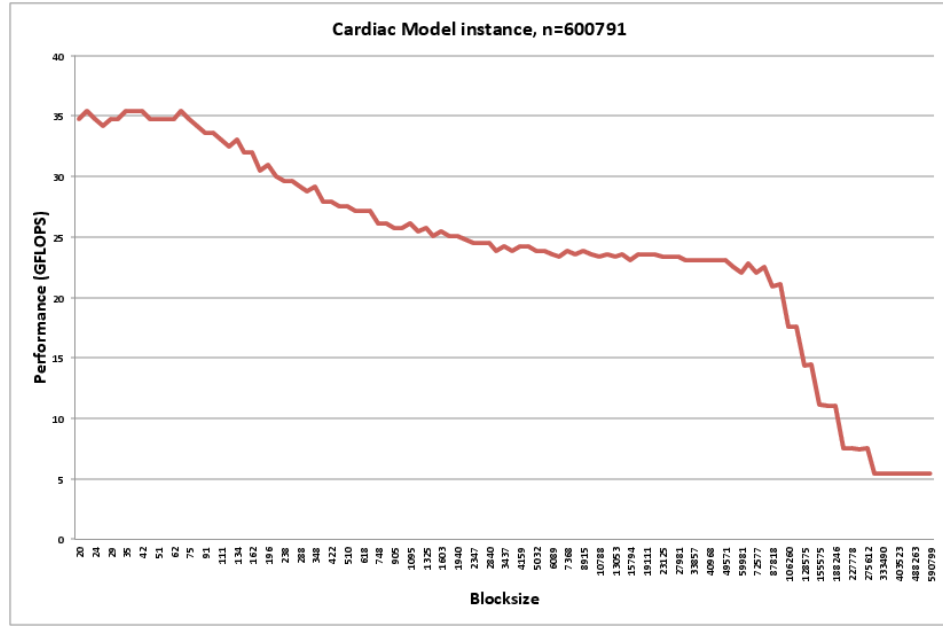
$$y(i) = \sum_{j=1}^{4} A(i,j)\,(x(\mathcal{I}(i,j)) - x(i))$$

  - Minimum amount of data load from global memory: 56 bytes
  - Theoretical peak performance on K20 GPU:

$$\frac{11 \text{ FLOP}}{56 \text{ B}} \times 208 \text{ GB/s} = 40.86 \text{ GFLOP/s}$$

Importance of tetrahedrons ordering

- Theoretical peak performance relies on
  - perfect data pre-fetch, perfect pipelining and perfect caching
- In reality:
  - can be more than 56 B/tet read from global memory
  - data traffic from the L2 cache: possible bottleneck
- Use of K20's shared memory and read-only data cache: important
- A reasonably good numbering of the tetrahedrons: important

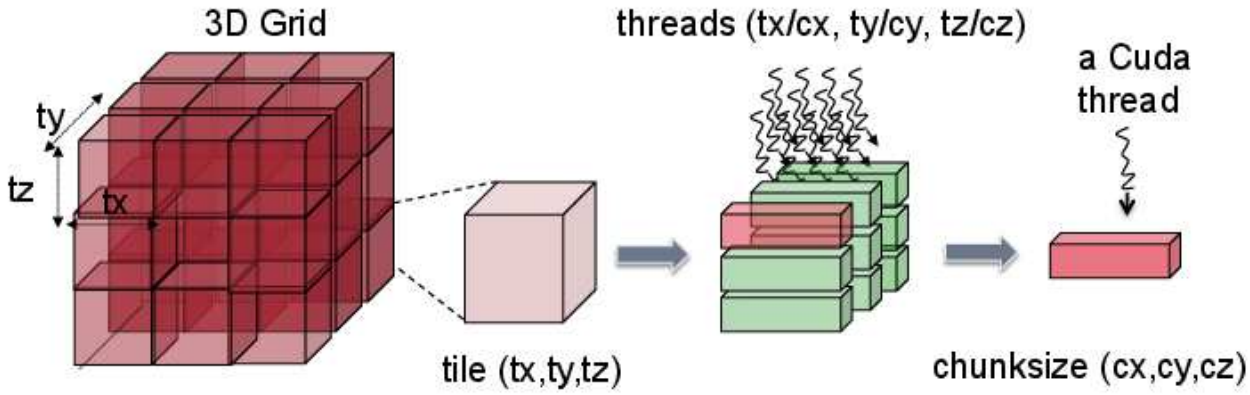


GPU programming is cumbersome

- GPU computing needs a CPU host, explicit data shuffles needed
- CUDA example of copying a 3D array from host to device:

```
cudaError_t stat_dev_1_u_old = make_cudaExtent((n) * sizeof(double ),(n),(
cudaPitchedPtr dev_1_u_old = cudaMalloc3D(&dev_1_u_old,ext_dev_1_u_old);
cudaMemcpy3DParms param_1_dev_1_u_old = {0};
param_1_dev_1_u_old.srcPtr = make_cudaPitchedPtr(((void *)u_old[0][0]),(n)
param_1_dev_1_u_old.dstPtr = dev_1_u_old;
param_1_dev_1_u_old.extent = ext_dev_1_u_old;
param_1_dev_1_u_old.kind = cudaMemcpyHostToDevice;
stat_dev_1_u_old = cudaMemcpy3D(&param_1_dev_1_u_old);
```

GPU programming is cumbersome (cont'd)



- In CUDA programs, threads are organized in a hierarchy
- Mapping is needed by each thread to "find its designated work"

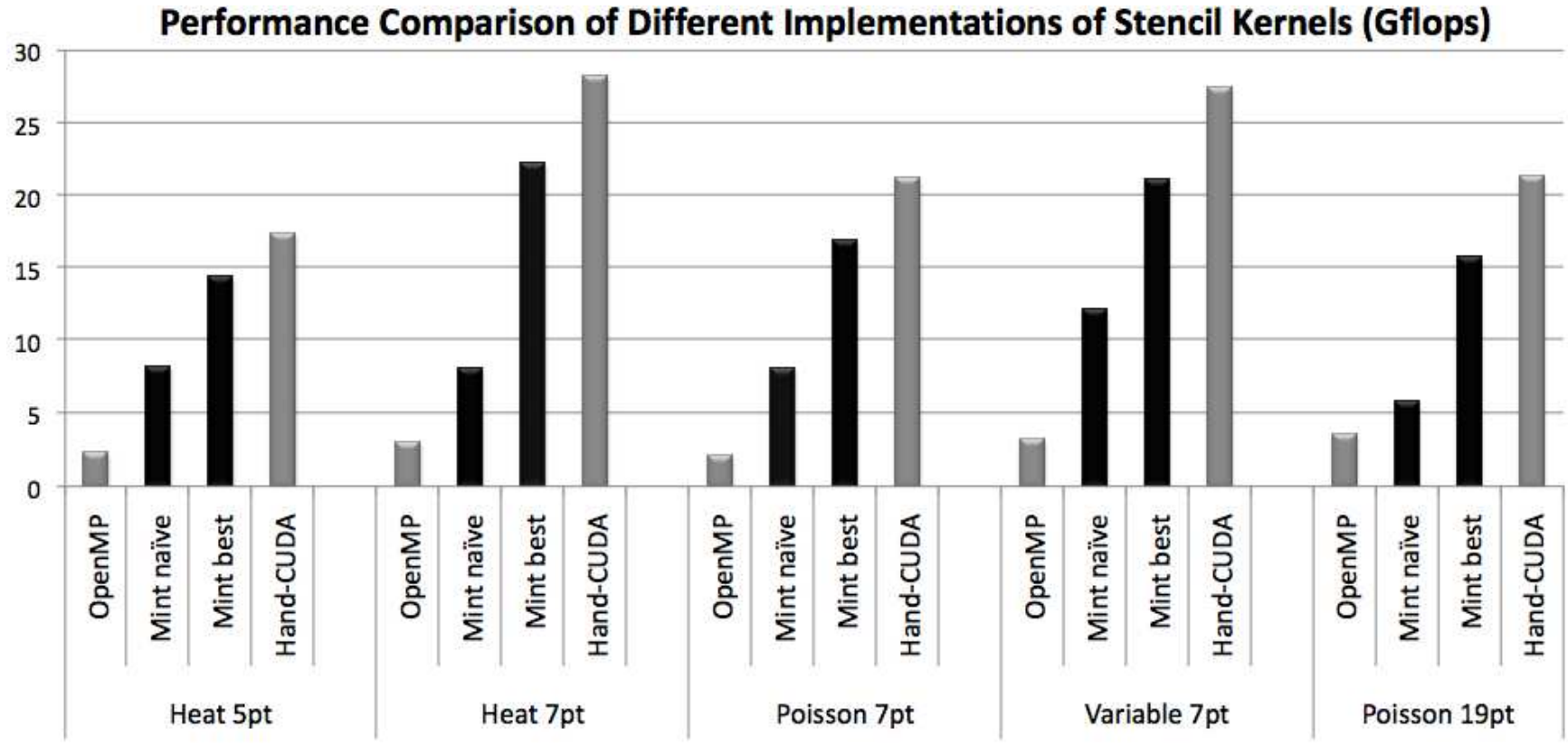
```
int _idx = threadIdx.x + 1;
int _gidx = _idx + blockDim.x * blockIdx.x;
int _idy = threadIdx.y + 1;
int _gidy = _idy + blockDim.y * 1 * blockIdx.y;
int _idz = threadIdx.z + 1;
int blockIdxz = blockIdx.y * invBlocksInY;
int blockIdxy = blockIdx.y - blockIdxz * blocksInY;
_gidy = _idy + blockIdxy * blockDim.y;
int _gidz = _idz + blockIdxz * blockDim.z;
int _index3D = _gidx + _gidy * _width + _gidz * _slice;
...
```

Mint

- Automated C-to-CUDA code generator and optimizer
  - Domain-specific targeting stencil computations
  - User only needs to annotate a serial C code with Mint pragmas
  - https://sites.google.com/site/mintmodel/

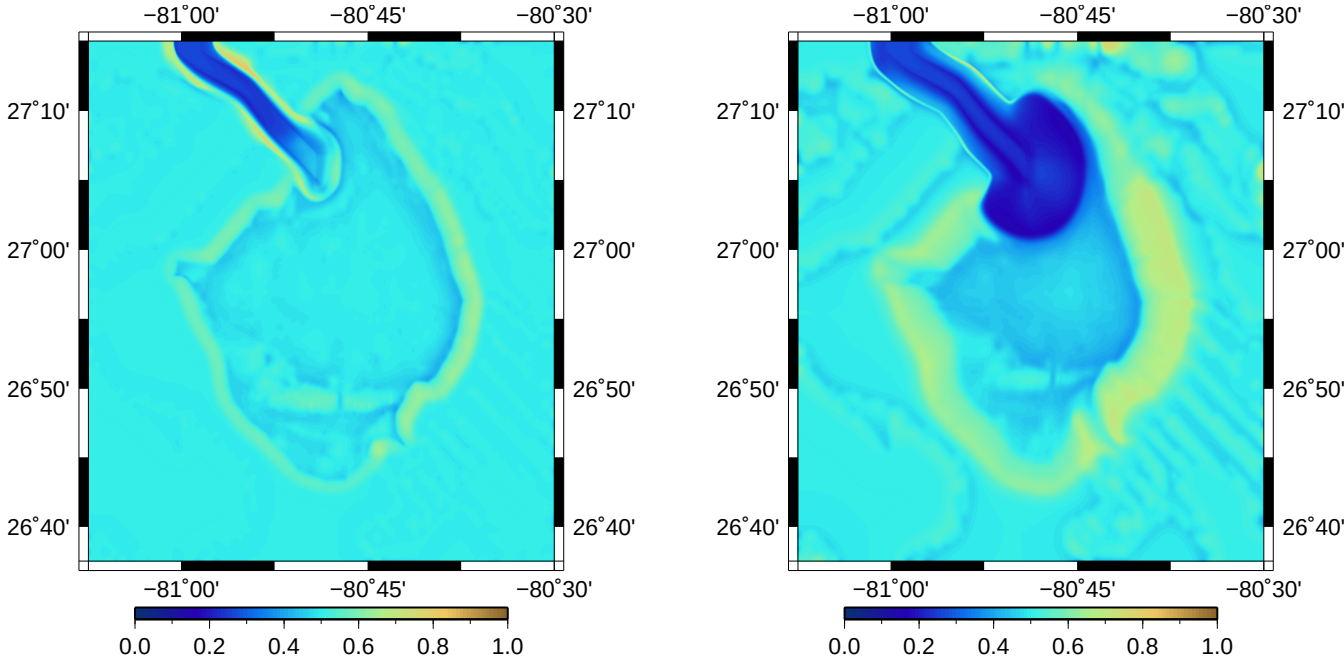
```
#pragma mint copy(U,toDevice,(n+2),(m+2),(k+2))
#pragma mint copy(Unew,toDevice,(n+2),(m+2),(k+2))

#pragma mint parallel default(shared) {
#pragma mint for nest(all) tile(16,16,1)
   for (int z=1; z<= k; z++)
    for (int y=1; y<= m; y++)
     for (int x=1; x<= n; x++)
      Unew[z][y][x] = c0 *  U[z][y][x] +
                  c1 * (U[z][y][x-1] + U[z][y][x+1] +
                  U[z][y-1][x] + U[z][y+1][x] + U[z-1][y][x] + U[z+1][y][x]);
}

#pragma mint copy(Unew,toHost,(n+2),(m+2),(k+2))
```

Mint performance



CPU: Nehalem E5504 quad-core, GPU: Tesla C1060

A realistic case of GPU computing



- 2D simulations of sedimentary basin filling
- A coupled system of two nonlinear PDEs:

$$\frac{\partial h}{\partial t} = \frac{1}{C_s}\nabla \cdot (\alpha s \nabla h) + \frac{1}{C_m}\nabla \cdot (\beta(1-s)\nabla h), \tag{1}$$

$$A\frac{\partial s}{\partial t} + s\frac{\partial h}{\partial t} = \frac{1}{C_s}\nabla \cdot (\alpha s \nabla h). \tag{2}$$

- Explicit finite difference based numerical strategy
- Two CUDA kernels: one for Eq. (1), the other for Eq. (2)

Performance on K20

- Three performance optimizations:
  - Using Kepler's read-only data cache
  - Using on-chip shared memory $\rightarrow$ avoid duplicated computations
  - Using halo threads $\rightarrow$ avoid if-tests

Measurements for the h-kernel

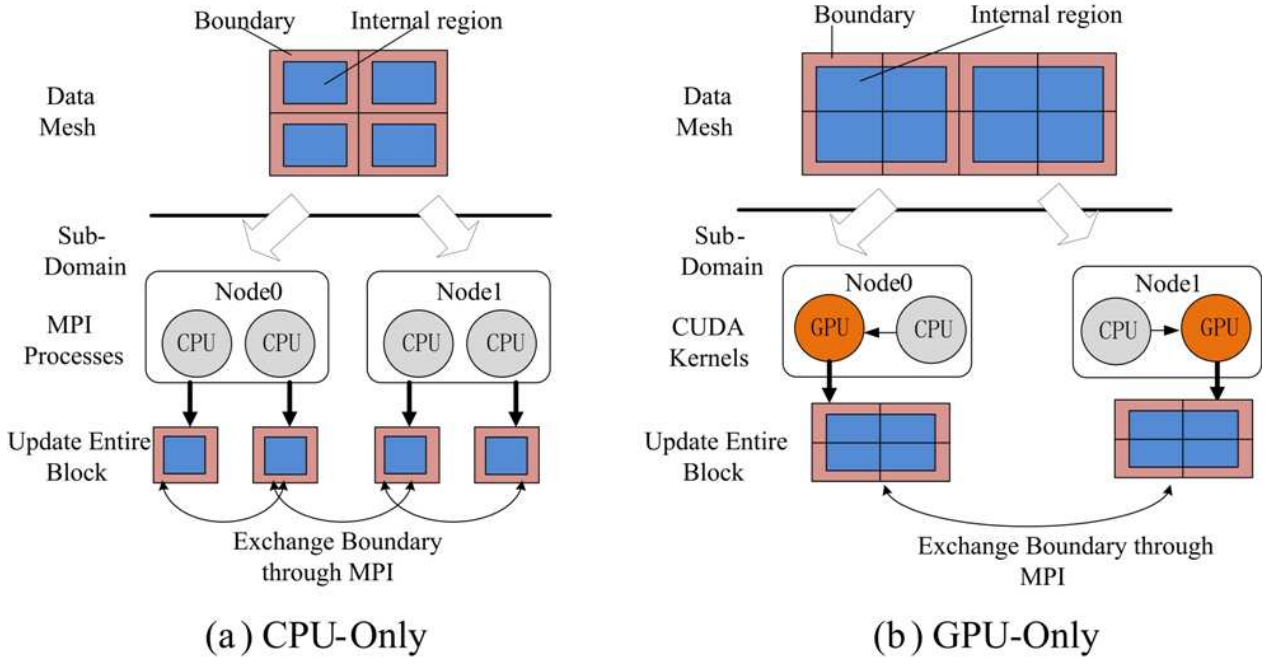
Code version	Thread block	GFlop/s	Registers/thread	Occupancy
Baseline	32×4	88.43	55	0.562
Read-only cache	32×4	178.69	52	0.562
Shared memory	32×4	182.36	33	0.750
Halo threads	34×6	190.45	34	0.656

Measurements for the s-kernel

Code version	Thread block	GFlop/s	Registers/thread	Occupancy
Baseline	32×4	67.99	39	0.750
Read-only cache	32×4	122.78	40	0.750
Shared memory	32×4	112.84	38	0.750
Halo threads	34×6	110.55	33	0.656

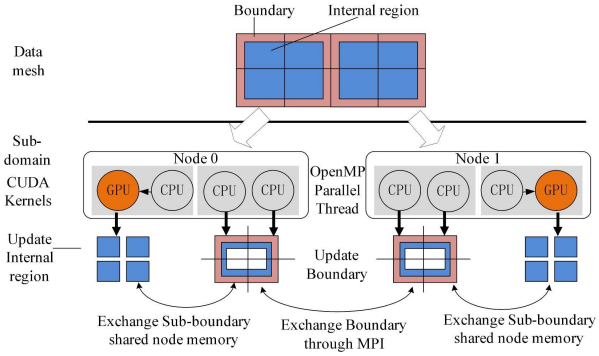
Using more GPUs

- Inter-GPU MPI communication has to go through the host CPUs

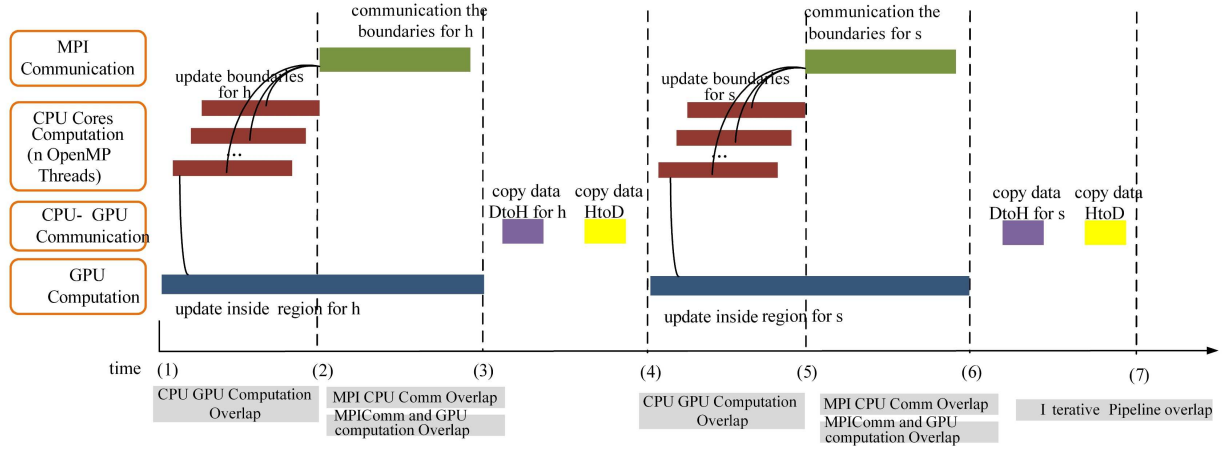


Hybrid computing

- Use host CPUs also for computation, in addition to MPI communication
- “Cut” an outer stripe per subdomain, and give it to the host CPU
- Possibility of pipelining of computation and communication, by using OpenMP threads



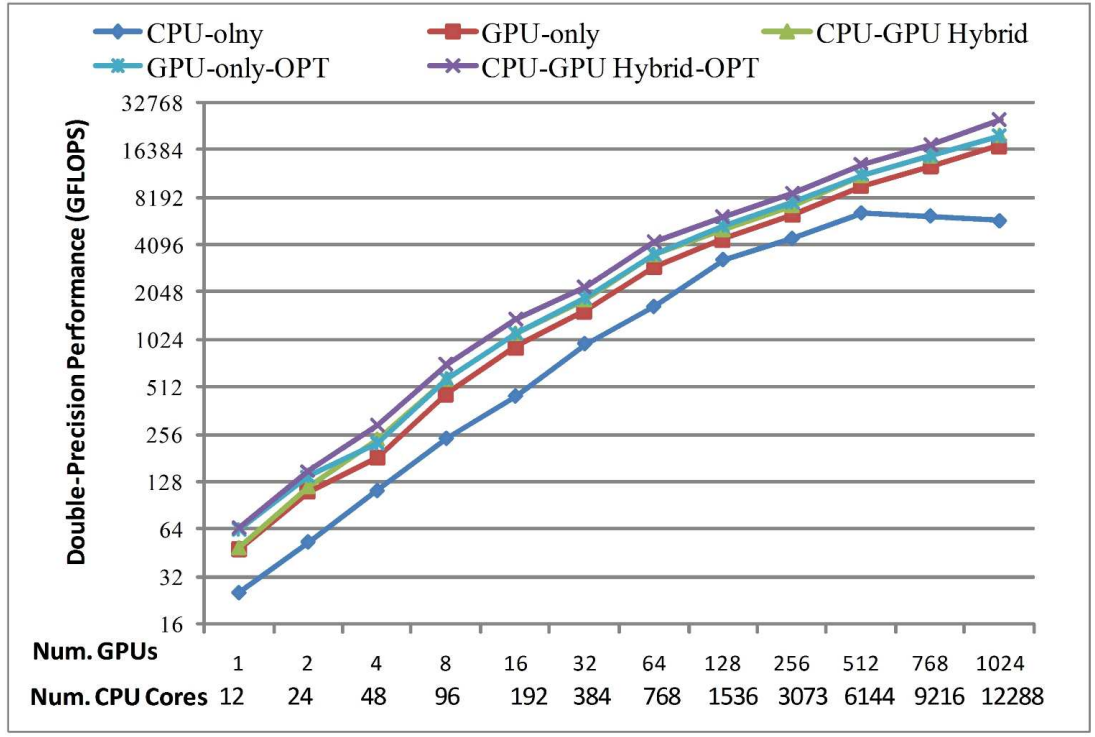
(a) CPU-GPU Hybrid: MPI/OpenMP/CUDA mode



(b) Timing Scheme of CPU-GPU Hybrid MPI/OpenMP/CUDA mode

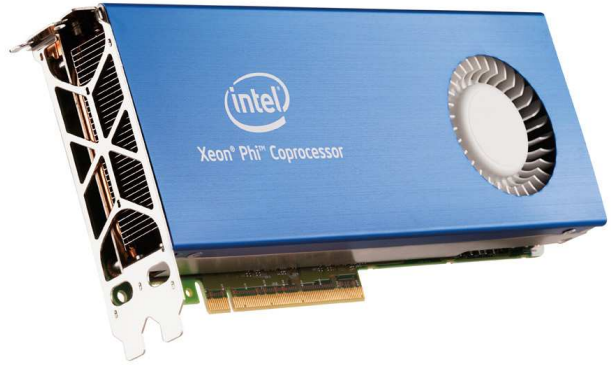
Measurements on Tianhe-1A

- Tianhe-1A: the world's largest supercomputer in June 2010
- Each node: Tesla M2050 GPU + two Xeon 6-core X5670 CPUs



Global 2D mesh: 16384×16384

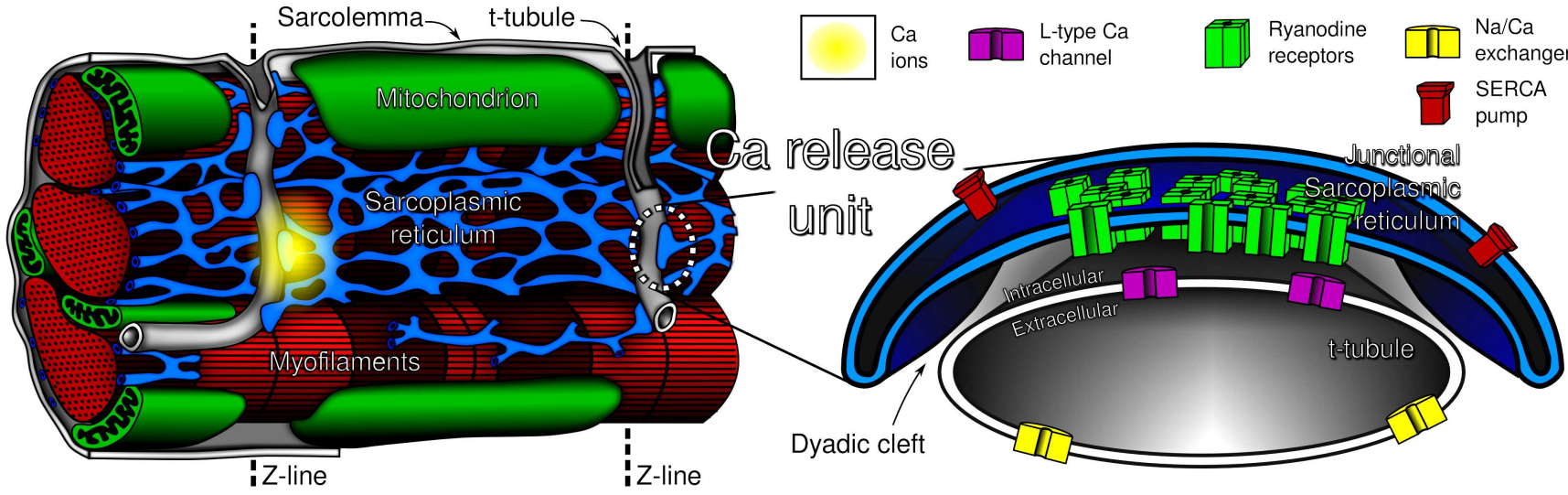
Xeon Phi coprocessor



- Many-integrated-core (MIC) architecture from Intel
- Tremendous theoretical peak DP performance $>$ 1 TFLOP/s
- $57 \sim 61$ cores per chip
- 4 threads per core
- Private L1 cache per core, shared L2 cache
- CPU-like versatile programmability
  - easy to get started
  - difficult to achieve good performance

Realistic application 2

- Subcellular calcium diffusion
- A coupled system of multiple 3D reaction-diffusion equations
- Towards nanometer mesh resolution

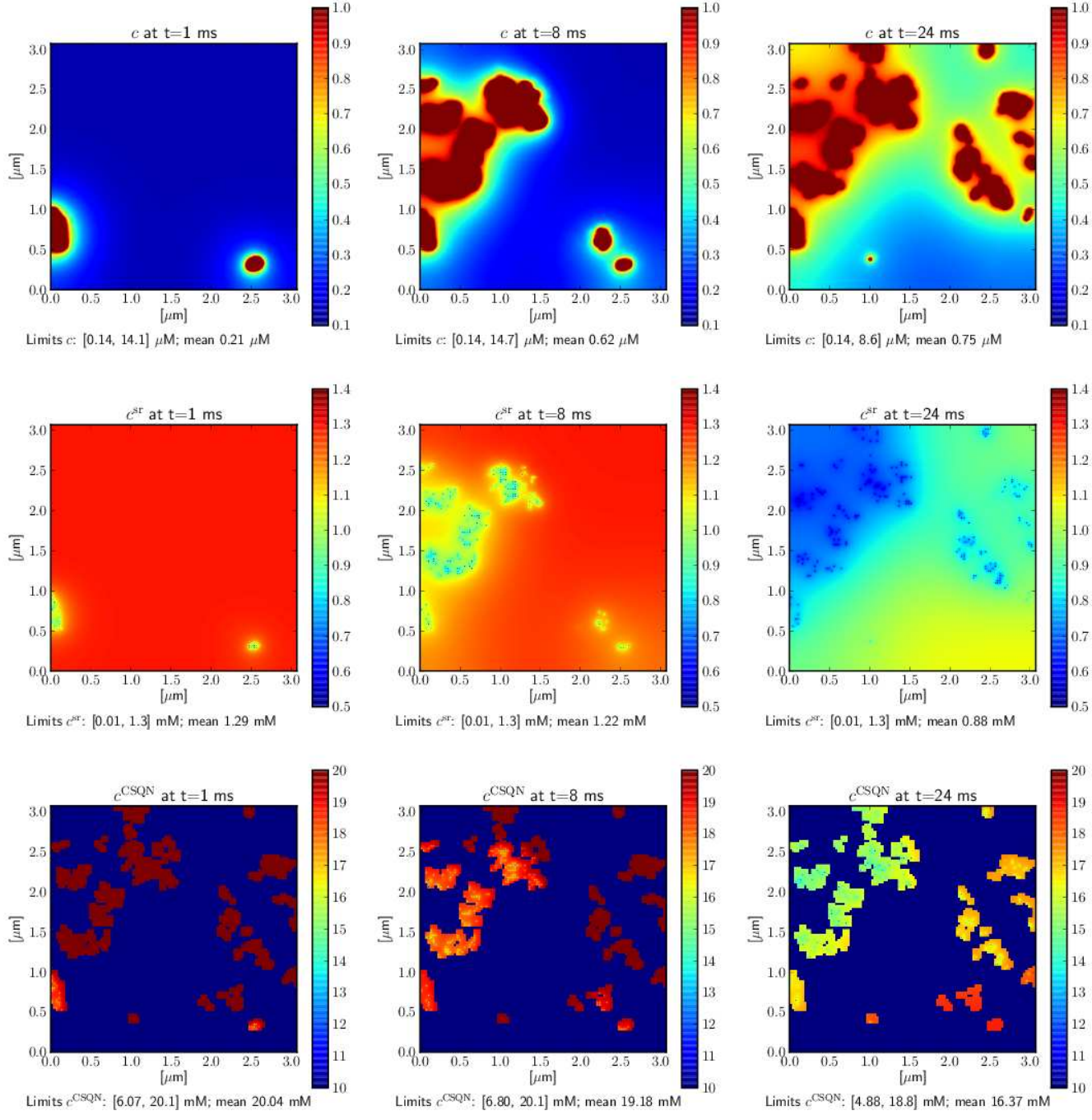


Mathematical model

$$
\begin{aligned}
\frac{\partial c}{\partial t} &= \mathrm{D}_{\mathrm{Ca}}^{\mathrm{cyt}} \nabla^2 c + R_{\mathrm{SR}}(c, c^{\mathrm{sr}}) - \sum_i R_i(c, c^{B_i}), \\
\frac{\partial c^{\mathrm{sr}}}{\partial t} &= \mathrm{D}_{\mathrm{Ca}}^{\mathrm{sr}} \nabla^2 c^{\mathrm{sr}} - \frac{R_{\mathrm{SR}}(c, c^{\mathrm{sr}})}{\gamma} - R_{\mathrm{CSQN}}(c^{\mathrm{sr}}, c^{B_{\mathrm{CSQN}}}), \\
\frac{\partial c^{B_{\mathrm{ATP}}}}{\partial t} &= \mathrm{D}_{\mathrm{ATP}}^{\mathrm{cyt}} \nabla^2 c^{B_{\mathrm{ATP}}} + R_{\mathrm{ATP}}(c, c^{B_{\mathrm{ATP}}}), \\
\frac{\partial c^{B_{\mathrm{CMDN}}}}{\partial t} &= \mathrm{D}_{\mathrm{CMDN}}^{\mathrm{cyt}} \nabla^2 c^{B_{\mathrm{CMDN}}} + R_{\mathrm{CMDN}}(c, c^{B_{\mathrm{CMDN}}}), \\
\frac{\partial c^{B_{\mathrm{Fluo}}}}{\partial t} &= \mathrm{D}_{\mathrm{Fluo}}^{\mathrm{cyt}} \nabla^2 c^{B_{\mathrm{Fluo}}} + R_{\mathrm{Fluo}}(c, c^{B_{\mathrm{Fluo}}}), \\
\frac{\mathrm{d}c^{B_{\mathrm{TRPN}}}}{\mathrm{d}t} &= R_{\mathrm{TRPN}}(c, c^{B_{\mathrm{TRPN}}}), \\
\frac{\mathrm{d}c^{B_{\mathrm{CSQN}}}}{\mathrm{d}t} &= R_{\mathrm{CSQN}}(c^{\mathrm{sr}}, c^{B_{\mathrm{CSQN}}}),
\end{aligned}
$$

- Five reaction-diffusion equations
- Two ordinary differential equations

Simulation snapshots



Limits c: [0.14, 14.1] μM; mean 0.21 μM

Limits c: [0.14, 14.7] μM; mean 0.62 μM

Limits c: [0.14, 8.6] μM; mean 0.75 μM

Limits c^{sr}: [0.01, 1.3] mM; mean 1.29 mM

Limits c^{sr}: [0.01, 1.3] mM; mean 1.22 mM

Limits c^{sr}: [0.01, 1.3] mM; mean 0.88 mM

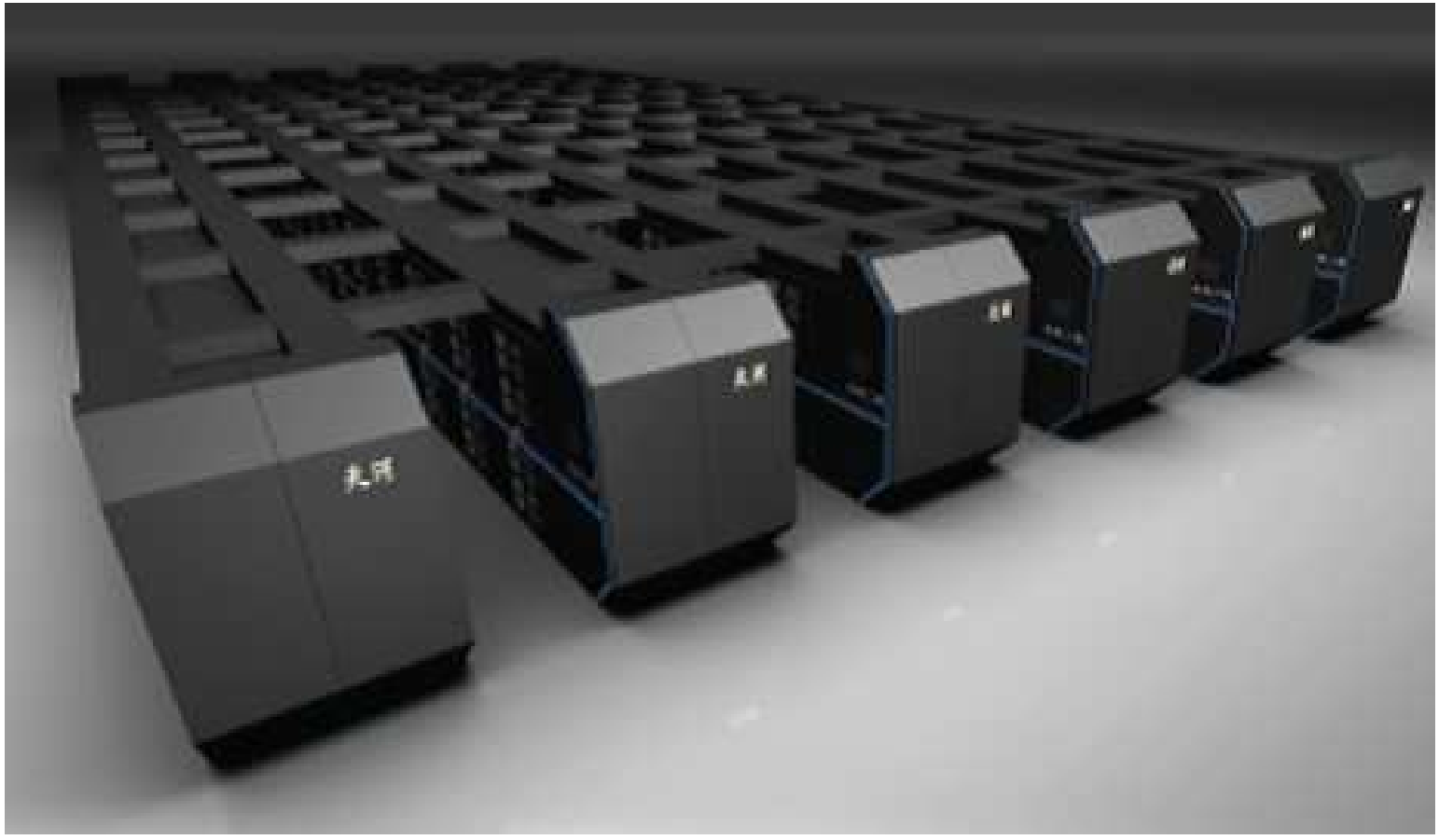
Limits c^{CSQN}: [6.07, 20.1] mM; mean 20.04 mM

Limits c^{CSQN}: [6.80, 20.1] mM; mean 19.18 mM

Limits c^{CSQN}: [4.88, 18.8] mM; mean 16.37 mM

Using Tianhe-2

- Tianhe-2: currently No. 1 supercomputer
- 16,000 compute nodes
  - Each node has three Xeon Phi coprocessors



Single-MIC performance

- Offload programming mode (initiated by host CPU)
  - `#pragma offload target(mic)`
- Spawning of $4 \times 56 = 224$ OpenMP threads on each Xeon Phi
  - `#pragma omp for collapse(2)` as parallelization
- Optimization techniques:
  - First touch + thread binding
  - Loop fusion
  - Hierarchical loop blocking
  - Explicit vectorization
  - Vector registers reuse
- Achieved 138 GB/s per MIC (90% of realistic memory bandwidth)
- Achieved 118 GFLOP/s per MIC (11.8% of theoretical DP peak)

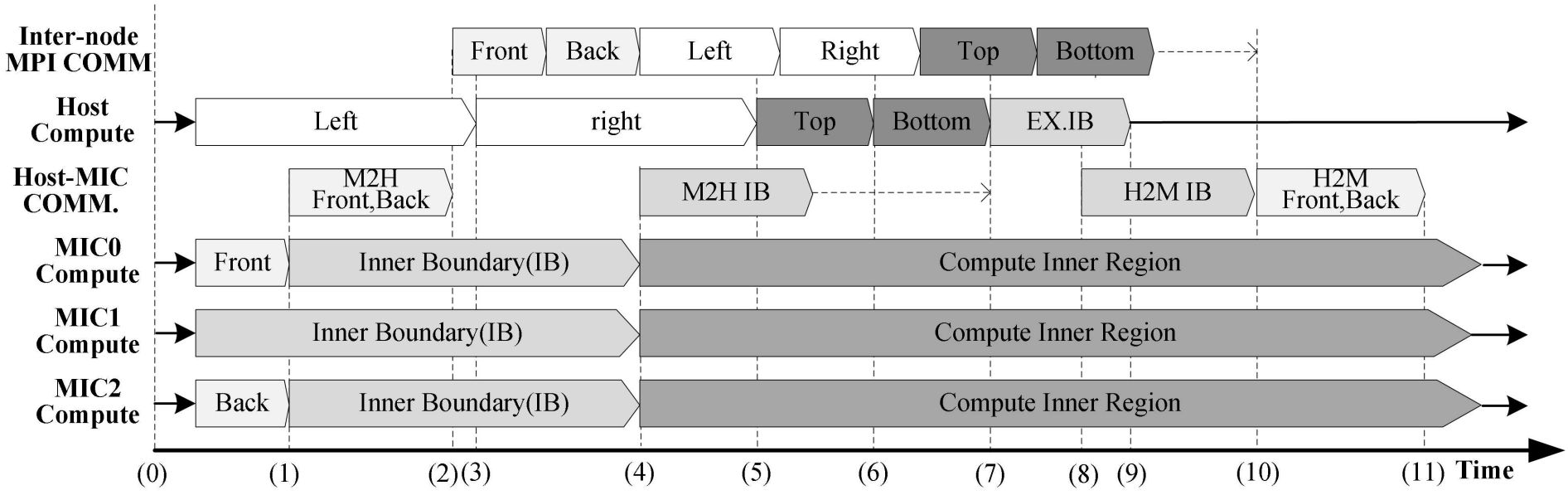
Single-node performance

- Three MICs lie side-by-side (in the y-direction)
- Offload programming mode
- Different sub-tasks done by different threads on host CPU:
  - invokes computation in the three MICs
  - does additional computation
  - issues host-MIC and MIC-host data exchange

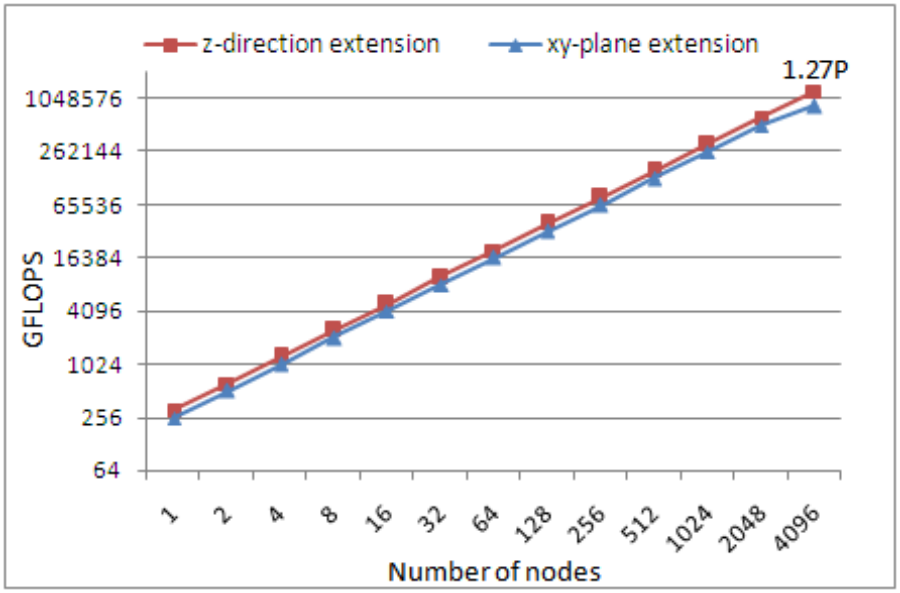
# MICs	Mesh points per MIC	Gflop/s
1	$142 \times 1200 \times 112$	111
2	$142 \times 600 \times 112$	226
3	$142 \times 400 \times 112$	326

Multi-node performance

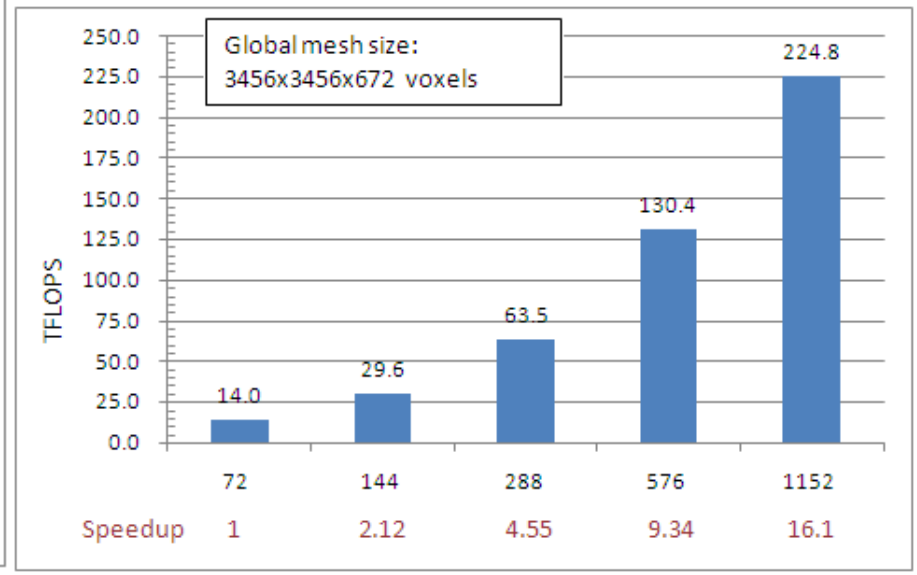
- MPI communication between nodes
  - One MPI process per node
  - Host CPU is responsible for communication and offloading
- Latency hiding through pipelining



Tianhe-2 performance



Weak-scalability tests



Strong-scalability tests

Some concluding remarks

- Accelerators provide new computing capabilities
  - ... new headaches at the same time
- Effective use of accelerators requires complicated programming
- Hybrid computing (using both CPUs and accelerators) is even more challenging
- Some programming tasks can possibly be generalized
  - automated code generation targeting specific computation domains
  - ongoing 4-year FriNatek project: **User-friendly programming of GPU-enhanced clusters via automated code translation and optimization**