

Introduction to CUDA Programming

Hemant Shukla

hshukla@lbl.gov

Trends

Scientific Data Deluge

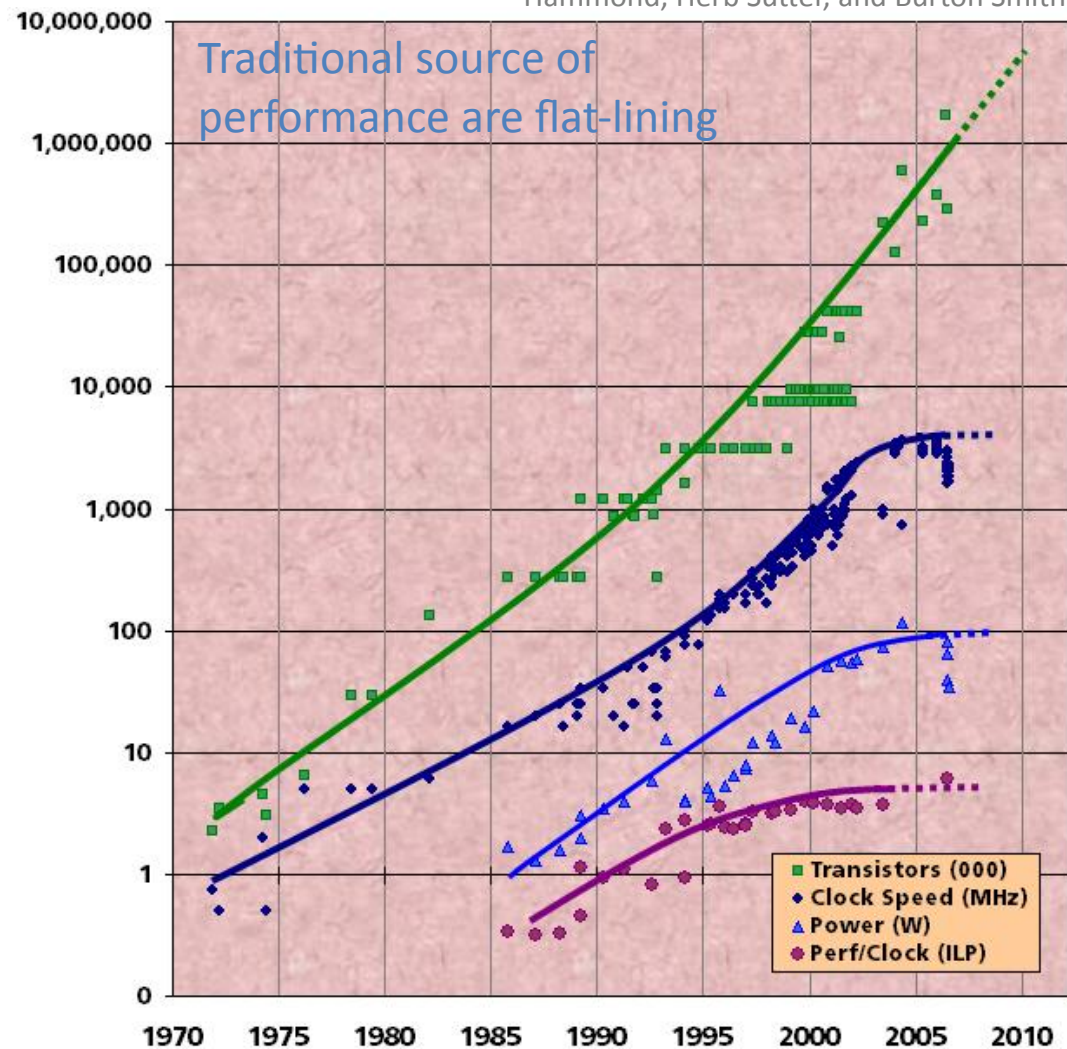
LSST	0.5 PB/month
JGI	5 TB/yr *
LOFAR	500 GB/s
SKA	100 x LOFAR

Energy Efficiency

Exascale will need
1000x Performance
enhancement with 10x
energy consumption
Flops/watt

* Jeff Broughton (NERSC) and JGI

Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith



Developments

Industry

Emergence of more cores on single chips

Number of cores per chip double every two years

Systems with millions of concurrent threads

Systems with inter and intra-chip parallelism

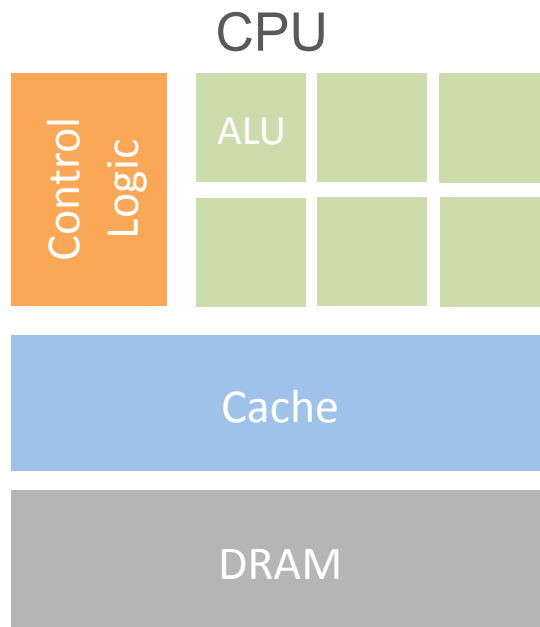
Architectural designs driven by reduction in Energy Consumption

New Parallel Programming models, languages, frameworks, ...

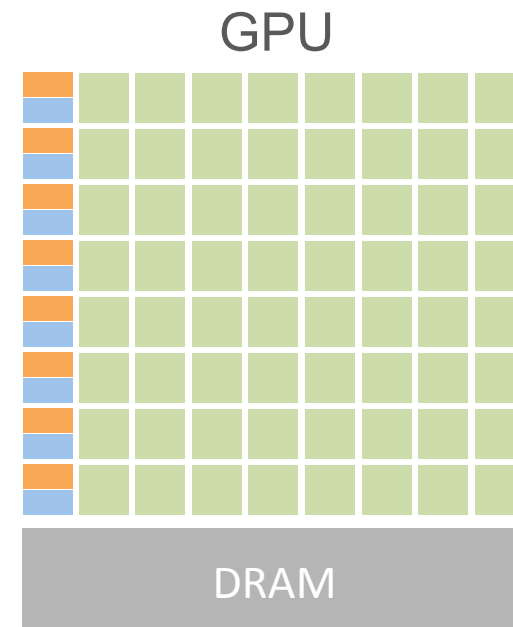
Academia

Graphical Processing Units (GPUs) are adopted as co-processors for high performance computing

Architectural Differences



Less than 20 cores (Heavyweight)
1-2 threads per core
Latency is hidden by large cache



512 cores (Lightweight)
10s to 100s of threads per core
Latency is hidden by fast context switching

GPUs don't run without CPUs

CPUs vs. GPUs

Silly debate... It's all about Cores

Next phase of HPC has been touted as “Disruptive”

Future HPC is massively parallel and likely on hybrid architectures

Programming models may not resemble the current state

Embrace change and brace for impact

Write modular, adaptable and easily mutative applications

Build auto-code generators, auto-tuning tools, frameworks, libraries

Use this opportunity to learn how to efficiently program massively parallel systems

Avoid partaking in turf wars unless are an employee and have to

Applications

X-ray computed tomography



Alain Bonissent et al.

Total volume
560 x 560 x 960 pixels
360 projections
Speed up = 110x

EoR with diesel powered radio interferometry



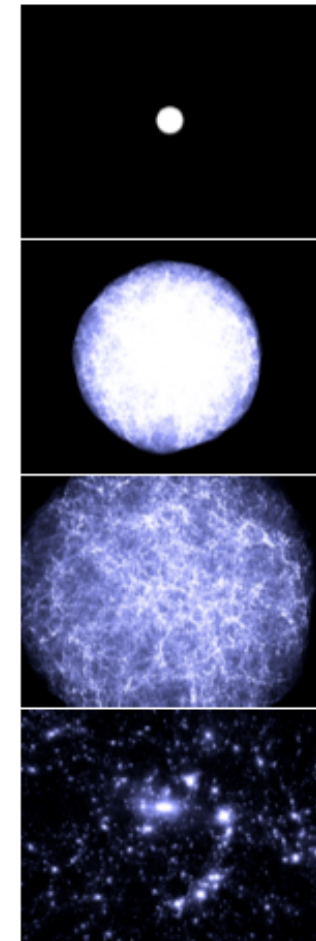
Lincoln Greenhill et al.

512 antennas, correlated visibilities for 130,000 baseline pairs, each with 768 channels and 4 polarizations ~ 20 Tflops. Power budget 20 kW.

INTEL Core2 Quad 2.66GHz = 1121 ms
NVIDIA GPU C1060 = 103.4 ms

4.5 giga-particles, $R = 630$ Mpc
2000x more volume than Kawai et al.

N-body with SCDM



K. Nitadori et al.

GPU

Programming Models

CUDA (Compute Unified Device Architecture)

OpenCL

Microsoft's DirectCompute

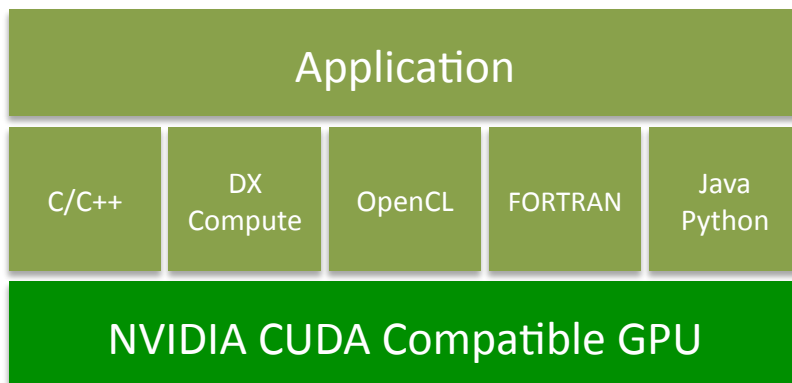
Third party wrappers are also available for Python, Perl, Fortran, Java, Ruby, Lua, MATLAB and IDL, and Mathematica

Compilers from PGI, RCC, HMPP, Copperhead

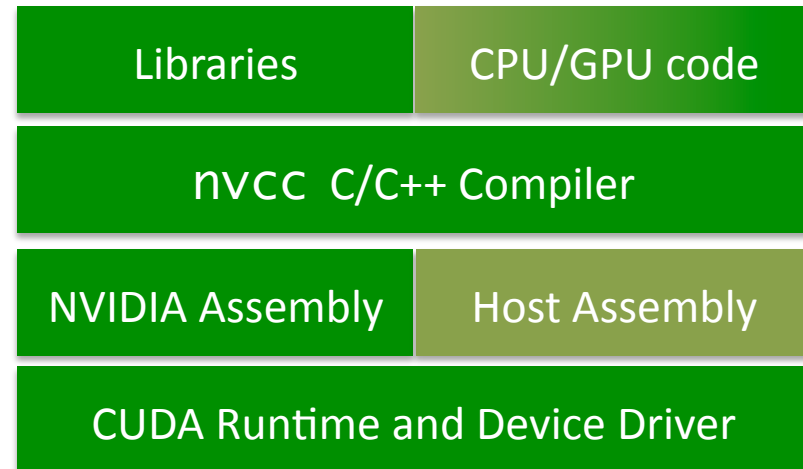
CUDA

Compute Defined Device Architecture

Parallel Computing Architecture

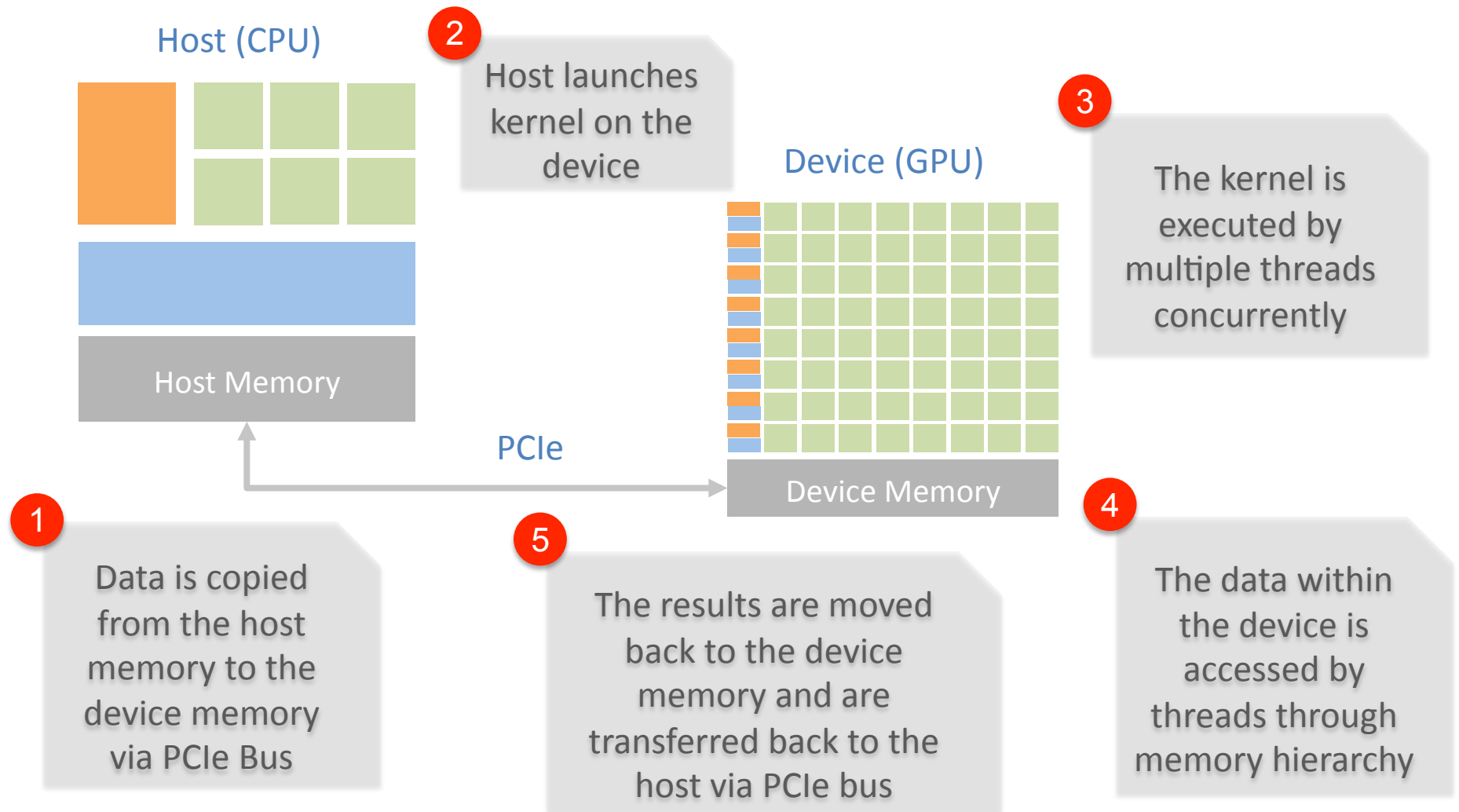


CUDA Device Driver
CUDA Toolkit (compiler, debugger, profiler, lib)
CUDA SDK (examples)
Windows, Mac OS, Linux



Libraries – FFT, Sparse Matrix, BLAS, RNG, CUSP, Thrust...

Dataflow



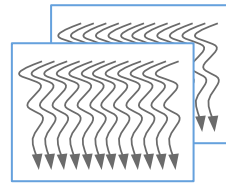
S/W Abstraction

Threads



Kernel is executed by threads processed by CUDA Core

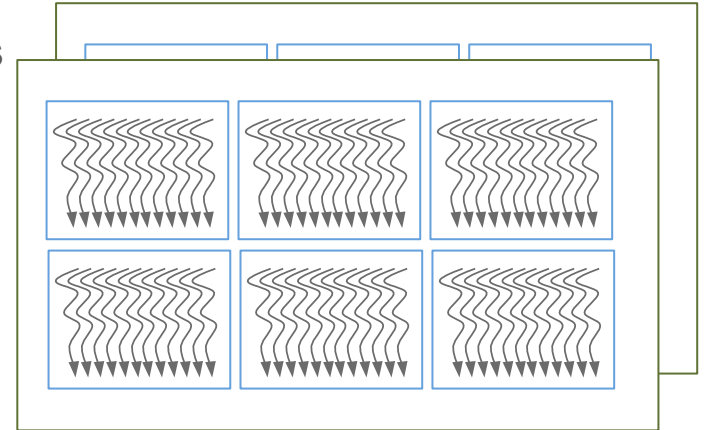
Blocks



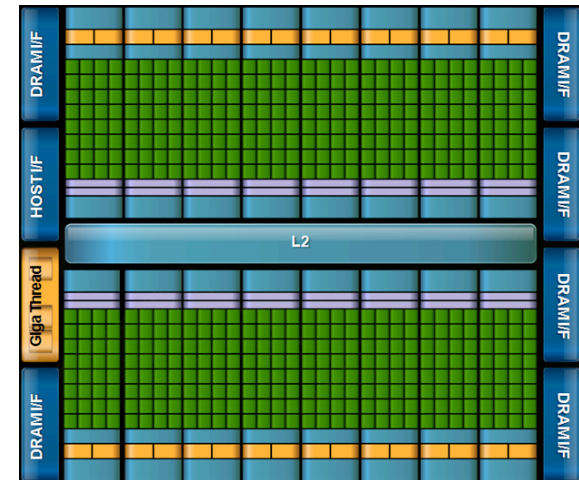
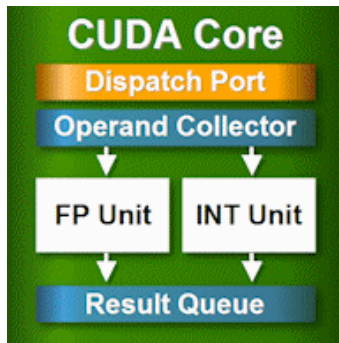
512-1024 threads / block

Maximum 8 blocks per SM
32 parallel threads are executed at the same time in a *WARP*

Grids



One grid per kernel with multiple concurrent kernels



Memory Hierarchy

Private memory

Visible only to the thread

Shared memory

Visible to all the threads in a block

Global memory

Visible to all the threads

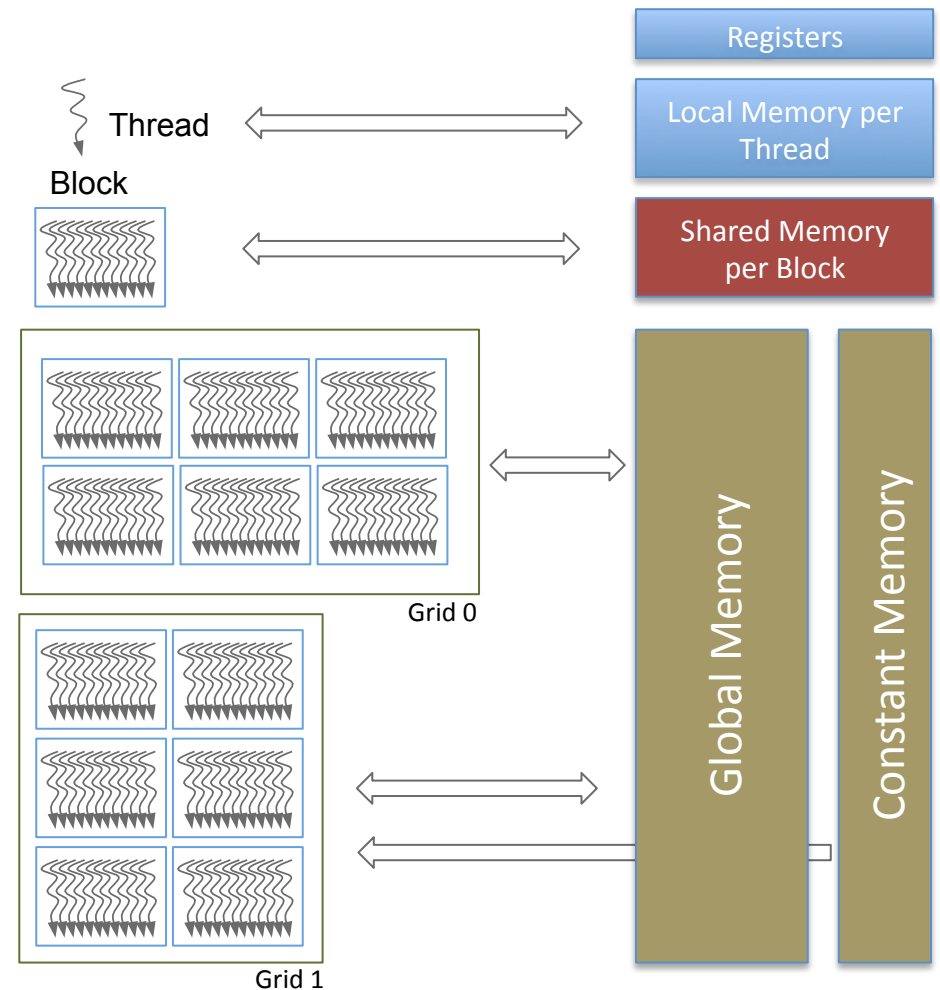
Visible to host

Accessible to multiple kernels

Data is stored in row major order

Constant memory (Read Only)

Visible to all the threads in a block



CUDA API Examples

Which GPU do I have?

```
#include <stdio.h>
int main()
{
    int noOfDevices;
    /* get no. of device */
    cudaGetDeviceCount (&noOfDevice);

    cudaDeviceProp prop;
    for (int i = 0; i < noOfDevices; i++)
    {
        /*get device properties */
        cudaGetDeviceProperties (&prop, i );

        printf ("Device Name:\t %s\n", prop.name);
        printf ("Total global memory:\t %ld\n",
                prop.totalGlobalMem);
        printf ("No. of SMs:\t %d\n",
                prop.multiProcessorCount);
        printf ("Shared memory / SM:\t %ld\n",
                prop.sharedMemPerBlock);
        printf("Registers / SM:\t %d\n",
                prop.regsPerBlock);

    }
    return 1;
}
```

Use
`cudaGetDeviceCount`
`cudaGetDeviceProperties`

Compilation

```
> nvcc whatDevice.cu -o whatDevice
```

Output

Device Name:	Tesla C2050
Total global memory:	2817720320
No. of SMs:	14
Shared memory / SM:	49152
Registers / SM:	32768

For more properties see
`struct cudaDeviceProp`

For details see CUDA Reference Manual

Timing with CUDA Event API

```
int main ()
{
    cudaEvent_t start, stop;
    float time;

    cudaEventCreate (&start);
    cudaEventCreate (&stop);

    cudaEventRecord (start, 0);

    someKernel <<<grids, blocks, 0, 0>>> (...);

    cudaEventRecord (stop, 0);
    cudaEventSynchronize (stop);
    cudaEventElapsedTime (&time, start, stop);

    cudaEventDestroy (start);
    cudaEventDestroy (stop);

    printf ("Elapsed time %f sec\n", time*.001);

    return 1;
}
```

CUDA Event API Timer are,

- OS independent
- High resolution
- Useful for timing asynchronous calls

← Ensures kernel execution has completed

Standard CPU timers will not measure the timing information of the device.

Memory Allocations / Copies

```
int main ()
{
    ...

    float host_signal[N]; host_result[N];
    float *device_signal, *device_result;

    //allocate memory on the device (GPU)
    cudaMalloc ((void**) &device_signal, N * sizeof(float));
    cudaMalloc ((void**) &device_result, N * sizeof(float));

    ... Get data for the host_signal array

    // copy host_signal array to the device
    cudaMemcpy (device_signal, host_signal , N * sizeof(float),
                cudaMemcpyHostToDevice);

    someKernel <<<< >>> (...);

    //copy the result back from device to the host
    cudaMemcpy (host_result, device_result, N * sizeof(float),
                cudaMemcpyDeviceToHost);

    //display the results
    ...
    cudaFree (device_signal); cudaFree (device_result) ;
}
```

Host and device have separate physical memory

Cannot dereference host pointers on device and vice versa

Basic Memory Methods

```
cudaError_t cudaMalloc (void ** devPtr, size_t size)
```

Allocates `size` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. In case of failure `cudaMalloc()` returns `cudaErrorMemoryAllocation`.

Blocking call

```
cudaError_t cudaMemcpy (void * dst, const void * src, size_t count, enum  
                        cudaMemcpyKind kind)
```

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`. The argument `kind` is one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`, and specifies the direction of the copy.

Non-Blocking call

```
cudaError_t cudaMemcpyAsync (void * dst, const void * src, size_t count,  
                             enum cudaMemcpyKind kind, cudaStream_t stream)
```

`cudaMemcpyAsync()` is asynchronous with respect to the host. The call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

See also, `cudaMemset`, `cudaFree`, ...

Kernel

The **CUDA kernel** is,

Run on device

Defined by `__global__` qualifier and does not return anything

```
__global__ void someKernel ();
```

Executed asynchronously by the host with `<<< >>>` qualifier, for example,

```
someKernel <<<nGrid, nBlocks, sharedMemory, streams>>> (...)
```

```
someKernel <<<nGrid, nBlocks>>> (...)
```

The kernel launches a 1- or 2-D **grid** of 1-, 2- or 3-D **blocks** of **threads**

Each thread executes the same kernel in parallel (SIMT)

Threads within blocks can communicate via shared memory

Threads within blocks can be synchronized

Grids and blocks are of type `struct dim3`

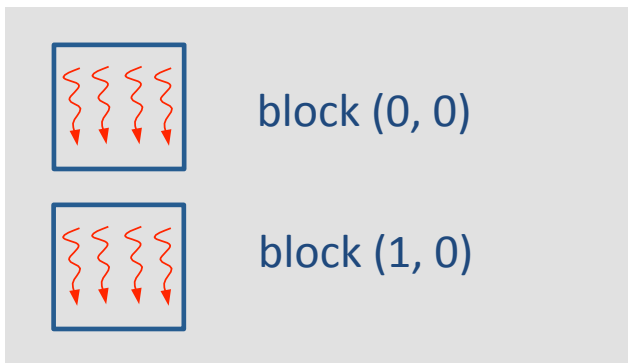
Built-in variables `gridDim`, `blockDim`, `threadIdx`, `blockIdx` are used to traverse across the device memory space with multi-dimensional indexing

Grids, Blocks and Threads

Grid



```
someKernel<<< 1, 1 >>> ();  
gridDim.x    = 1  
blockDim.x   = 1  
blockIdx.x   = 0  
threadIdx.x  = 0
```



```
dim3 blocks (2,1,1);  
someKernel<<< (blocks, 4) >>> ();  
gridDim.x    = 2;  
blockDim.x   = 4;  
blockIdx.x   = 0,1;  
threadIdx.x  = 0,1,2,3,0,1,2,3
```

<<< number of blocks in a grid, number of threads per block >>>

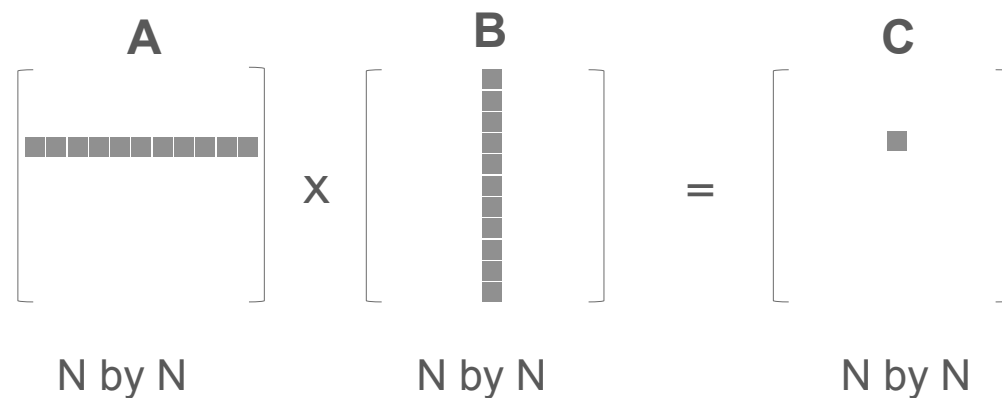
Useful for multidimensional indexing and creating unique thread IDs

```
int index = threadIdx.x + blockDim.x * blockIdx.x;
```

Example - Inner Product

Matrix-multiplication

Each element of product matrix **C** is generated by row column multiplication and reduction of matrices **A** and **B**. This operation is similar to inner product of the vector multiplication kind also known as vector dot product.



For size $N \times N$ matrices the matrix-multiplication $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ will be equivalent to N^2 independent (hence parallel) inner products.

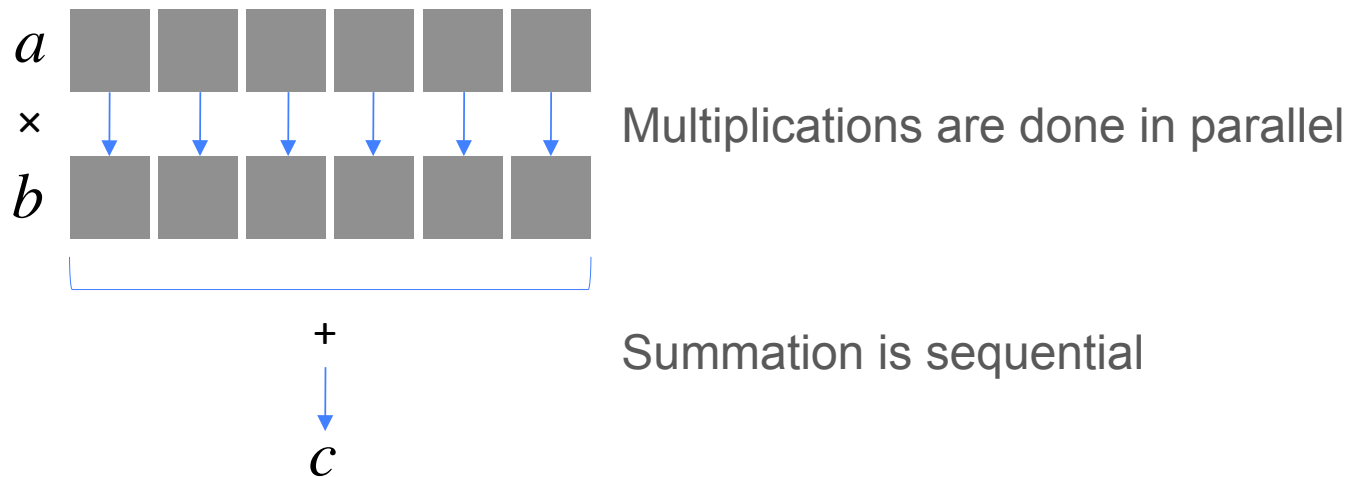
Example

$$c = \sum_i a_i b_i$$

Serial representation

```
double c = 0.0;
for (int i = 0; i < SIZE; i++)
    c += a[i] * b[i];
```

Simple parallelization strategy



Example

CUDA Kernel

```
__global__ void innerProduct (int *a, int *b, int *c)
{
    int product[SIZE];

    int i = threadIdx.x;

    if (i < SIZE)
        product[i] = a[i] * b[i];
}
```

```
__global__ void innerProduct (...)
```

```
{
```

```
    ...
```

```
}
```

```
int main ()
```

```
{
```

```
    ...
```

```
    innerProduct<<<grid, block>>> (...);
```

```
    ...
```

```
}
```

Called in the host code

Example

```
__global__ void innerProduct (int *a, int *b, int *c)
{
    int product[SIZE];

    int i = threadIdx.x;

    if (i < SIZE)
        product[i] = a[i] * b[i];

}
```

Qualifier `__global__` encapsulates device specific code that runs on the device and is called by the host

Other qualifiers are, `__device__`, `__host__`, `host__and__device`

`threadIdx` is a built in iterator for threads. It has 3 dimensions x, y and z.

Each thread with a unique `threadIdx.x` runs the kernel code in parallel.

Example

```
__global__ void innerProduct (int *a, int *b, int *c)
{
    int product[SIZE];

    int i = threadIdx.x;

    if (i < SIZE)
        product[i] = a[i] * b[i];

    int sum = 0;
    for (int k = 0; k < N; k++)
        sum += product[k];
    *c = sum;
}
```

Now we can sum the all the products to get the scalar c

Unfortunately this won't work for following reasons,

- `product[i]` is local to each thread
- Threads are not visible to each other

Example

```
__global__ void innerProduct (int *a, int *b, int *c)
{
    __shared__ int product[SIZE];

    int i = threadIdx.x;

    if (i < SIZE)
        product[i] = a[i] * b[i];

    __syncthreads();

    if (threadIdx.x == 0)
    {
        int sum = 0;
        for (int k = 0; k < SIZE; k++)
            sum += product[k];
        *c = sum;
    }
}
```

First we make the product[i] visible to all the threads by copying it to shared memory

Next we make sure that all the threads are synchronized. In other words each thread has finished its workload before we move ahead. We do this by calling __syncthreads()

Finally we assign summation to one thread (extremely inefficient reduction)

Aside: cudaThreadSynchronize() is used on the host side to synchronize host and device

Example

```
__global__ void innerProduct (int *a, int *b, int *c)
{
    __shared__ int product[SIZE];

    int i = threadIdx.x;

    if (i < SIZE)
        product[i] = a[i] * b[i];

    __syncthreads();

    // Efficient reduction call

    *c = someEfficientLibrary_reduce (product);
}
```

Exercise: Review CUDA SDK example on reduction.

Performance Considerations

Memory Bandwidth

Memory bandwidth – rate at which the data is transferred – is a valuable metric to gauge the performance of an application

Theoretical Bandwidth

Memory bandwidth (GB/s) = Memory clock rate (Hz) × interface width (bytes) / 10^9

Real Bandwidth (Effective Bandwidth)

Bandwidth (GB/s) = [(bytes read + bytes written) / 10^9] / execution time

Bandwidth is affected by the memory access patterns

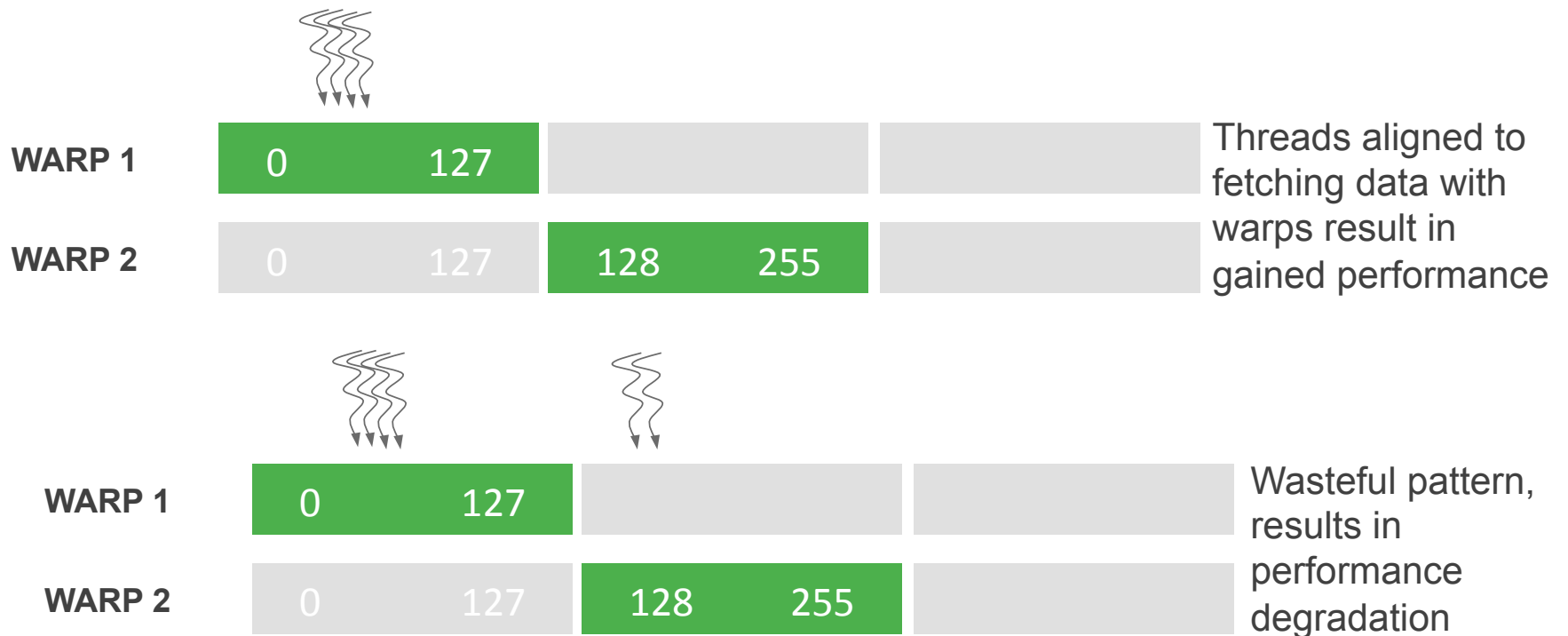
If real bandwidth is much lower than the theoretical then code may need review

Optimize on Real Bandwidth

May also use profilers to estimate bandwidth and bottlenecks

Memory Access Patterns

Global memory access made in a warp is **coalesced** (contiguous addresses) and by default cached in 128-byte words by L1 Cache



Arithmetic Intensity

Memory access bandwidth of GPUs is limited compared to the peak compute throughput

High arithmetic intensity (arithmetic operations per memory access) algorithms perform well on such architectures

Example

Fermi peak throughput for SP is 1 TFLOP/s and DP is 0.5 TFLOP/s
Global memory (off-chip) bandwidth is 144 GB/s

For every 4 byte single precision floating point operand bandwidth is 36 GB/s and 18 GB/s for double precision

To obtain peak throughput will require $1000/36 \sim 28$ SP (14 DP) arithmetic operations

Example revisited

```
__global__ void innerProduct (int *a, int *b, int *c)
{
    __shared__ int product[SIZE];

    int i = threadIdx.x;

    if (i < SIZE)
        product[i] = a[i] * b[i];

    __syncthreads();

    if (threadIdx.x == 0)
    {
        int sum = 0;
        for (int k = 0; k < SIZE; k++)
            sum += product[k];
        *c = sum;
    }
}
```

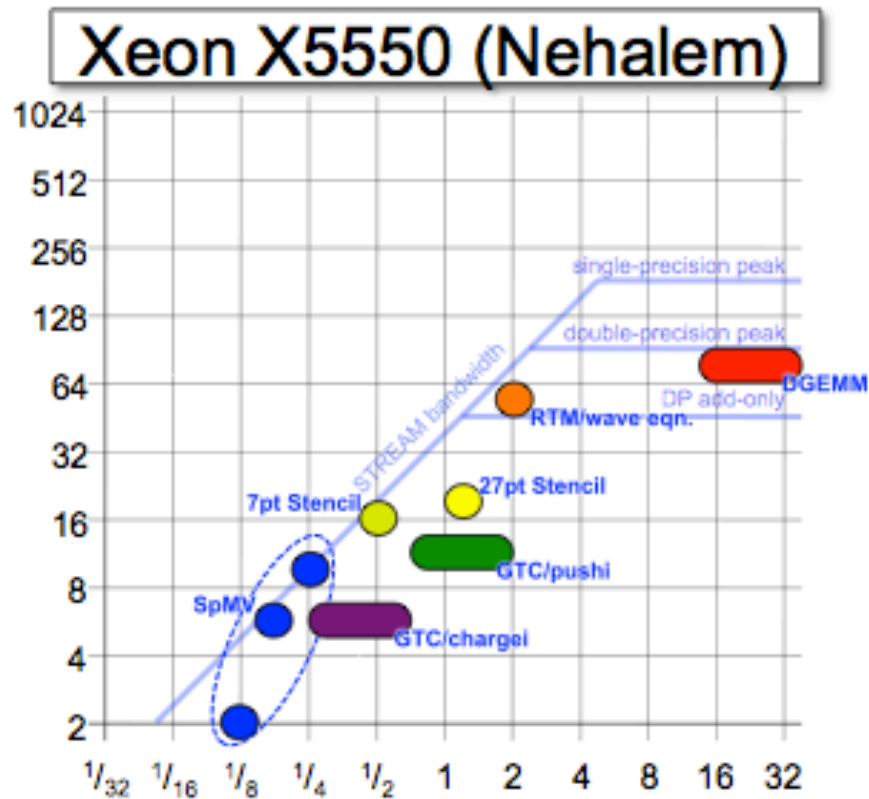
Contrast this with inner product example where for every 2 memory (data a_i and b_i) accesses only two operations (multiplication and add) are performed. That is ratio of 1 as opposed to 28 that is required for peak throughput.

Room for algorithm improvement!

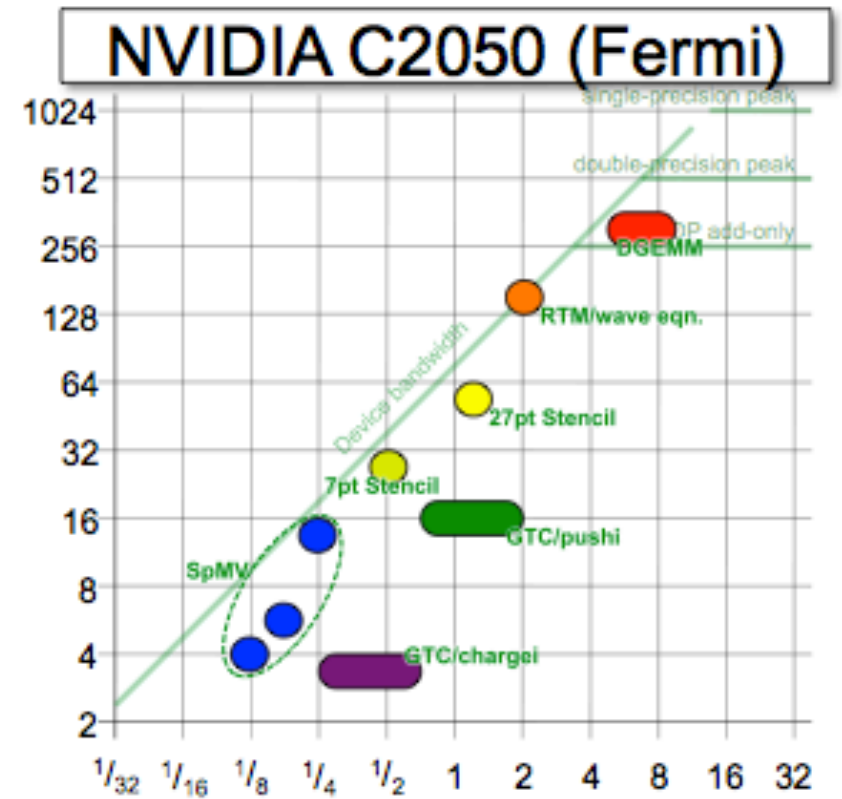
Aside: Not all performance will be peak performance

Benchmarks

Relative Performance of Algorithms



Gflop/s



Arithmetic Intensity

Courtesy - Sam Williams

Optimization Strategies

Coalesced memory data accesses (use faster memories like shared memory)

Minimize data transfer over PCIe (~ 5 GB/s)

Overlap data transfers and computations with asynchronous calls

Use fast page-locked memory (pinned memory – host memory guaranteed to device)

Judiciously

Threads in a block should be multiples of 32 (warp size). Experiment with your device

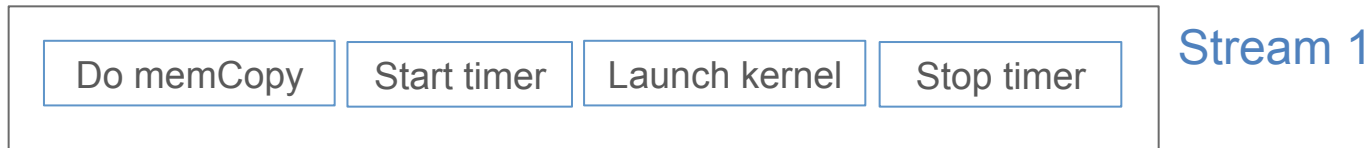
Smaller thread-blocks better than large many threads blocks when resource limited

Fast libraries (cuBLAS, Thrust, CUSP, cuFFT,...)

Built-in arithmetic instructions

CUDA Streams

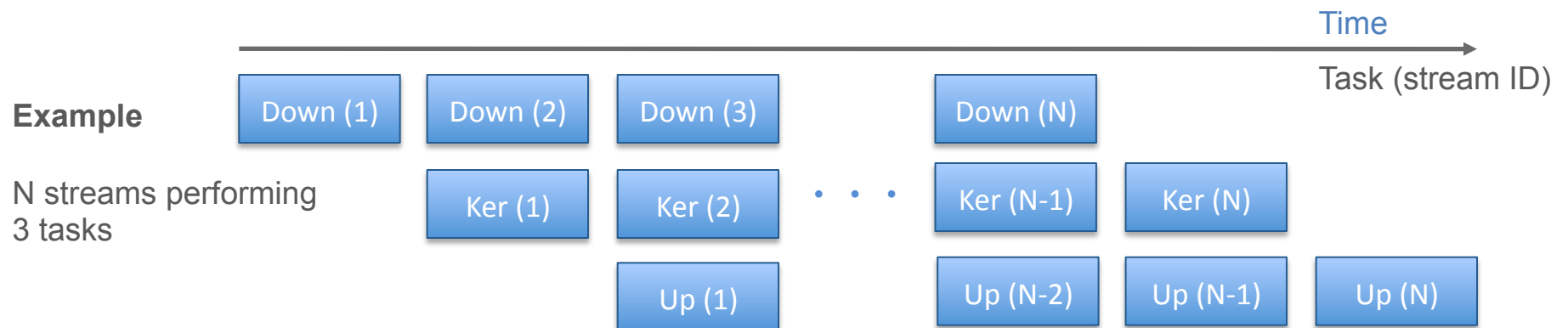
Stream is defined as sequence of device operations executed in order



```
cudaStream_t stream0, stream1;
```

```
cudaStreamCreate (&stream0);
```

```
cudaMemcpyAsync (... , stream0); someKernel<<<... , stream0>>>();  
cudaMemcpyAsync (... , stream1); someKernel<<<... , stream1>>>();  
cudaStreamSynchronize (stream0);
```



Use case

Simulations

GAMER

Hsi-Yu Schive, T. Chiueh, and Y. C. Tsai

Astrophysics adaptive mesh refinement (AMR) code with solvers for hydrodynamics and gravity

Parallelization achieved by OpenMP, MPI on multi-node multicores and CUDA for accelerators (GPU)

Decoupling of AMR (CPU) and solvers (GPU) lends to increased performance, ease of code development

Speed-ups of the order of 10-12x attained on single and multi-GPU heterogeneous systems

GAMER Framework

Hemant Shukla, Hsi-Yu Schive, Tak-Pong Woo, and T. Chiueh

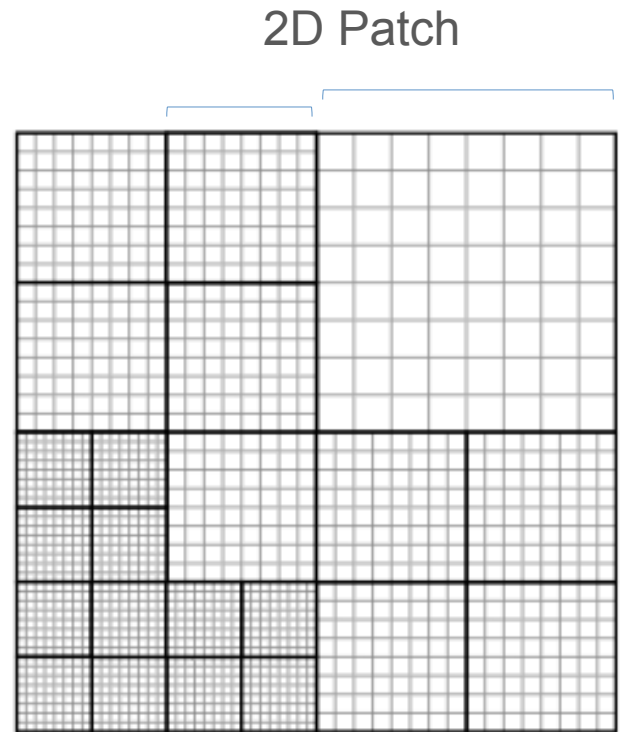
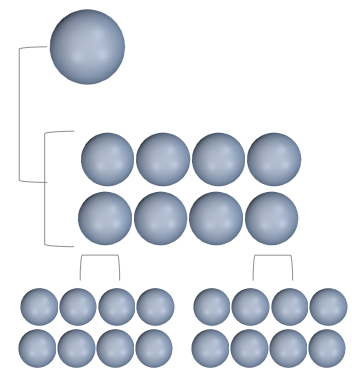
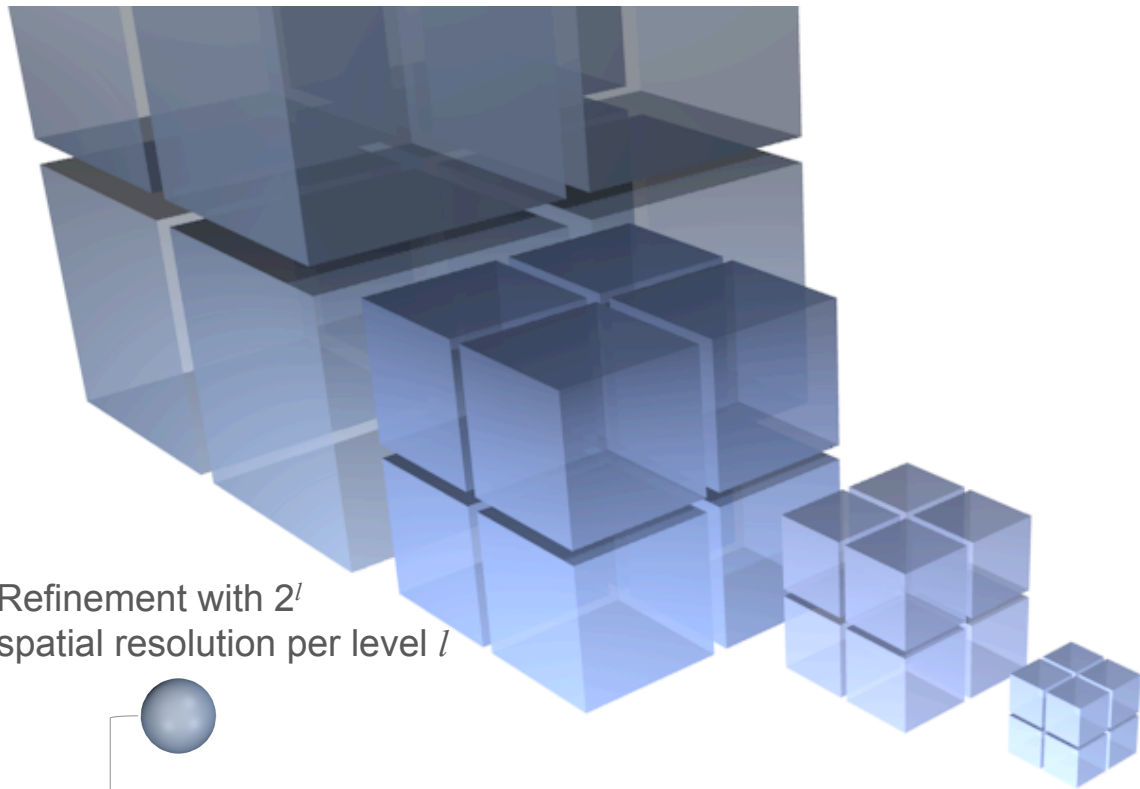
Generalized GAMER codebase to multi-science framework

Use GAMER to deeply benchmark heterogeneous hardware, optimizations and algorithms in applications

Collect performance, memory access, power consumption and various other metrics for broader user base

Develop codebases as ensembles of highly optimized existing and customizable components for HPC

Adaptive Mesh Refinement



8^3 cells per patch

Identical spatial geometry (same kernel)
Uniform and individual time-steps

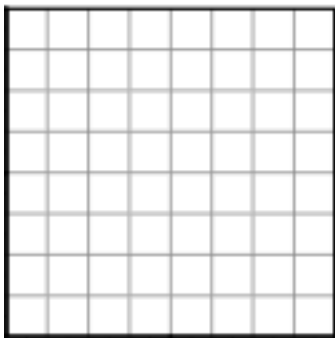
Figure - Hsi-Yu Schive et al., 2010

Construct and Dataflow

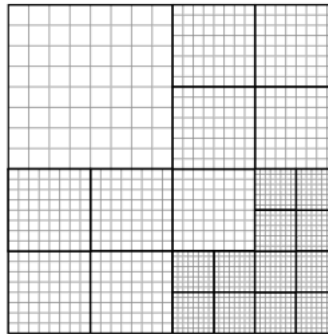
GAMER Codebase C++/CUDA, MPI, OpenMP

AMR, Framework, Libraries

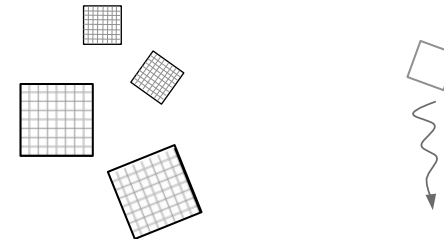
Solvers Poisson, Hydro, Custom, ...



Problem domain covered with coarse patch on CPUs



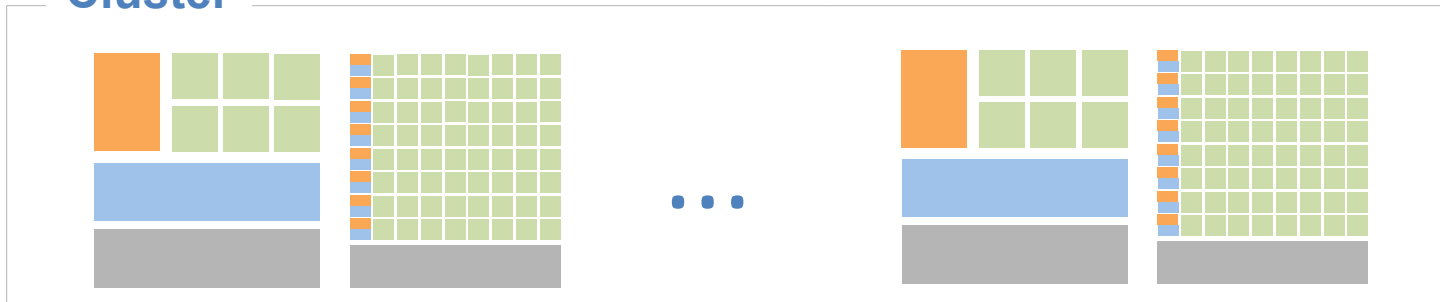
User defined refinement, spatial averaging, flux correction on CPUs



Concurrently patches are transferred to GPUs, processed by solvers, one cell per thread, and returned



Cluster



Solvers

Hydrodynamics PDE Solver

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho v_j)}{\partial x_j} = 0$$

$$\frac{\partial(\rho v_i)}{\partial t} + \frac{\partial(\rho v_i v_j + P \delta_{ij})}{\partial x_j} = -\rho \frac{\partial \phi}{\partial x_i}$$

$$\frac{\partial e}{\partial t} + \frac{\partial[(e + P)v_j]}{\partial x_j} = -\rho v_j \frac{\partial \phi}{\partial x_j}$$

3D Euler equations solved in 5 separate schemes

Second-order relaxing Total Variation Diminishing
Weighted average flux
MUSCL-Hancock (MHM)
MUSCL-Hancock (VL)
Corner transport upwind (CTU)

Flux conservation is done using Riemann Solver
(4 types - exact solver, HLLE, HLLC, and Roe)

Poisson-Gravity Solver

$$\nabla^2 \phi(\vec{x}) = 4\pi G \rho(\vec{x})$$

Laplacian operator ∇^2 is replaced by seven-point finite difference operator

For root level patches Green's functions is used using FFTW

For refined levels SOR is used

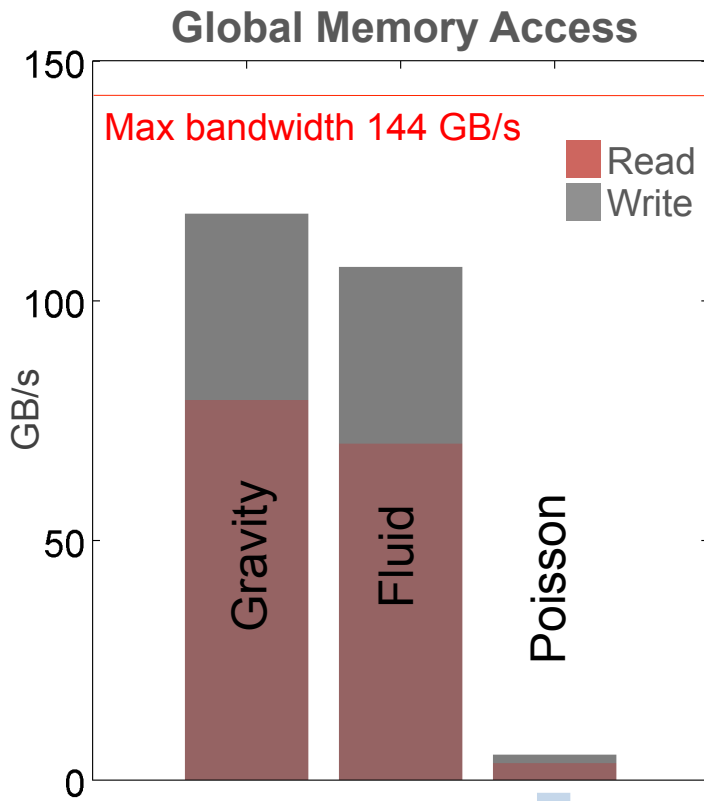
Recently implemented

Multigrid Poisson Solver
Hilbert space-filling curve (load balancing)

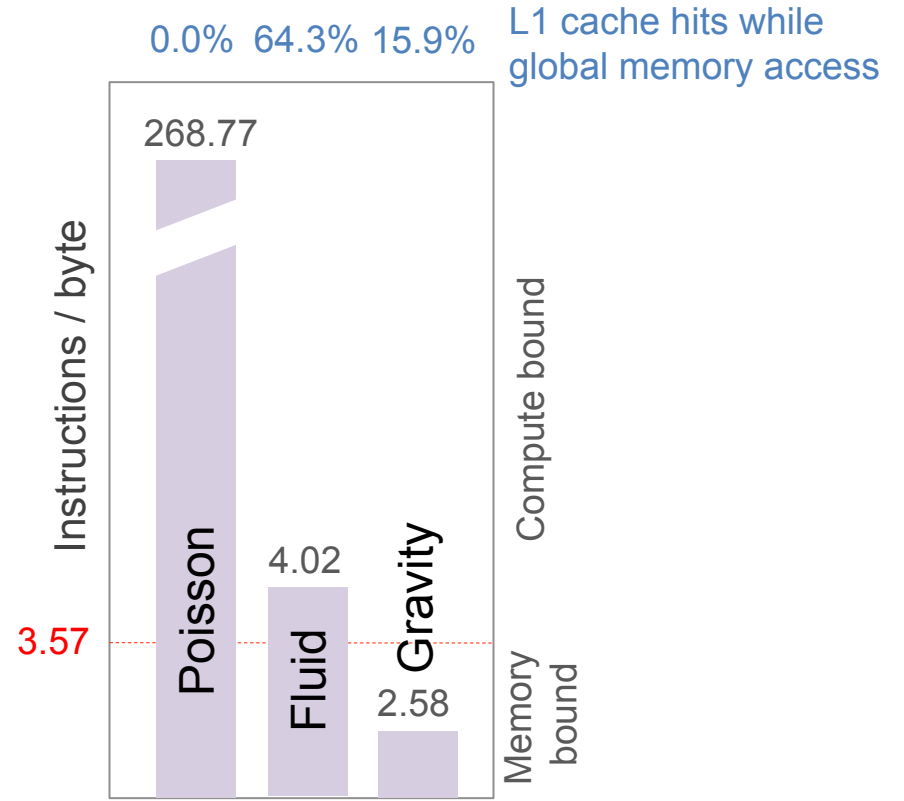
Currently implementing

Fast Poisson Solver with Dirichlet's boundary conditions

Kernel Analysis



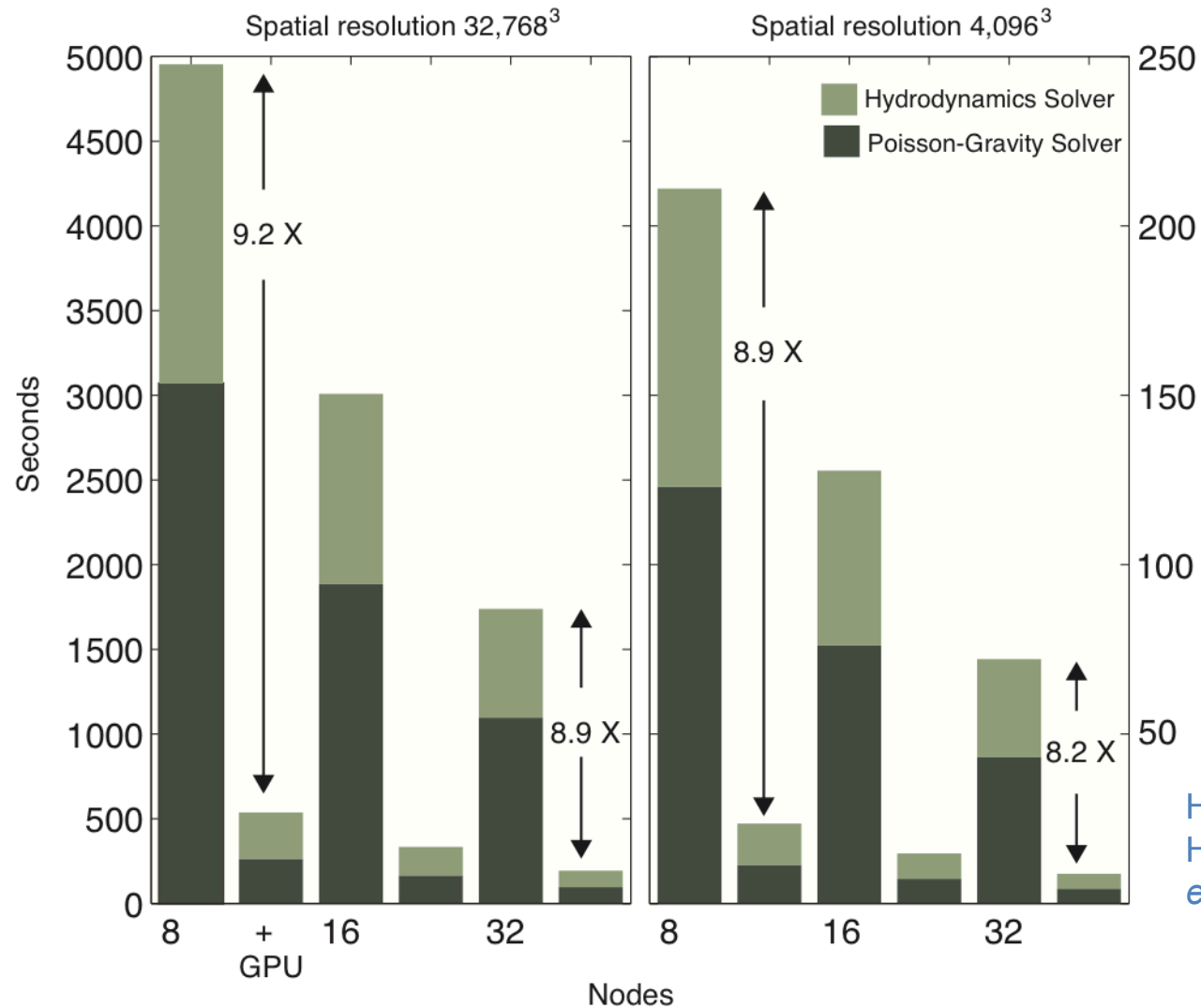
Intensive use of shared memory



SOR takes 20-30 iterations to converge

Results

Large scale Cosmological Simulations with GAMER

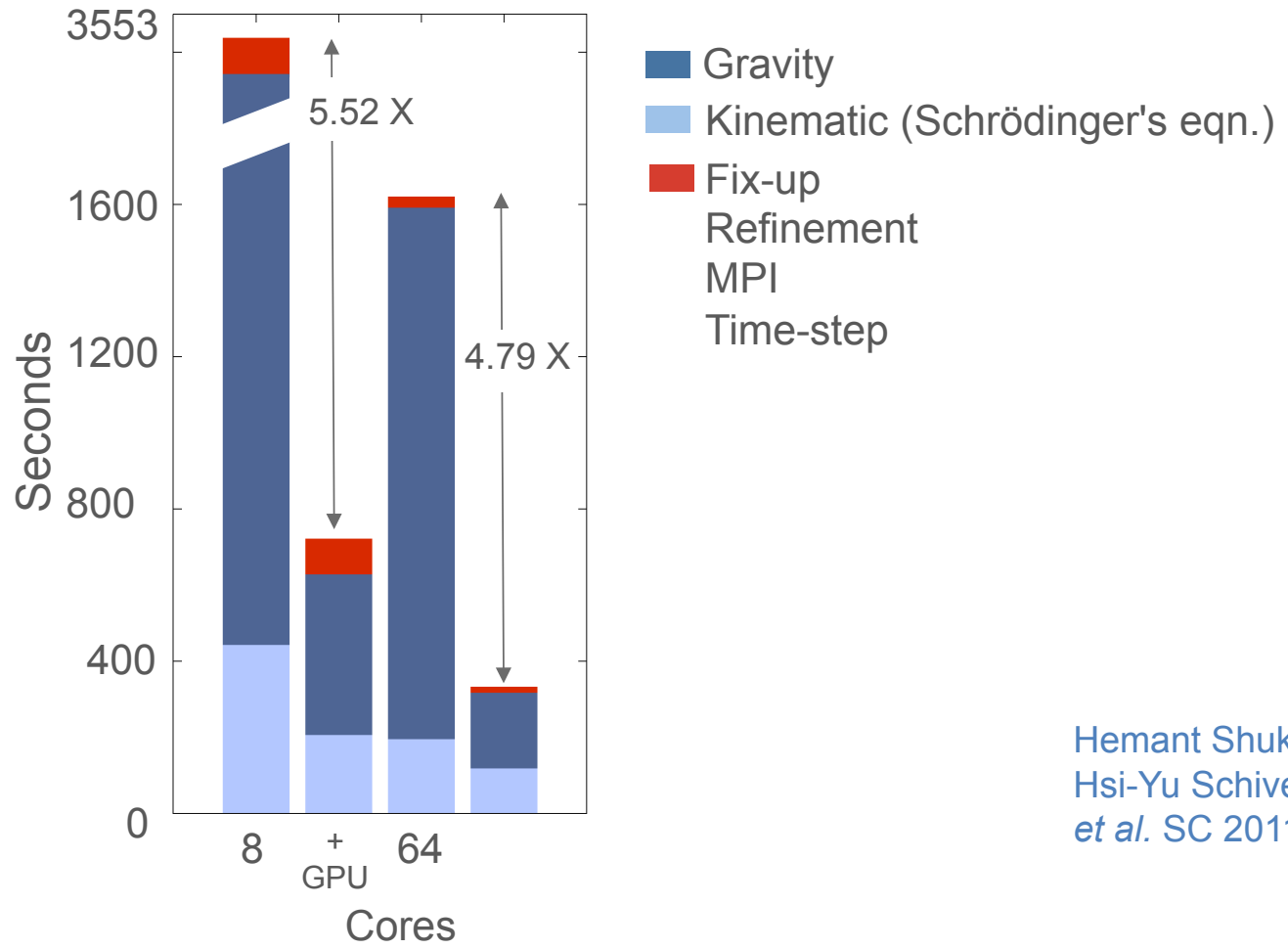


Hemant Shukla,
Hsi-Yu Schive
et al. SC 2011

Results

Bosonic Dark Matter Simulation

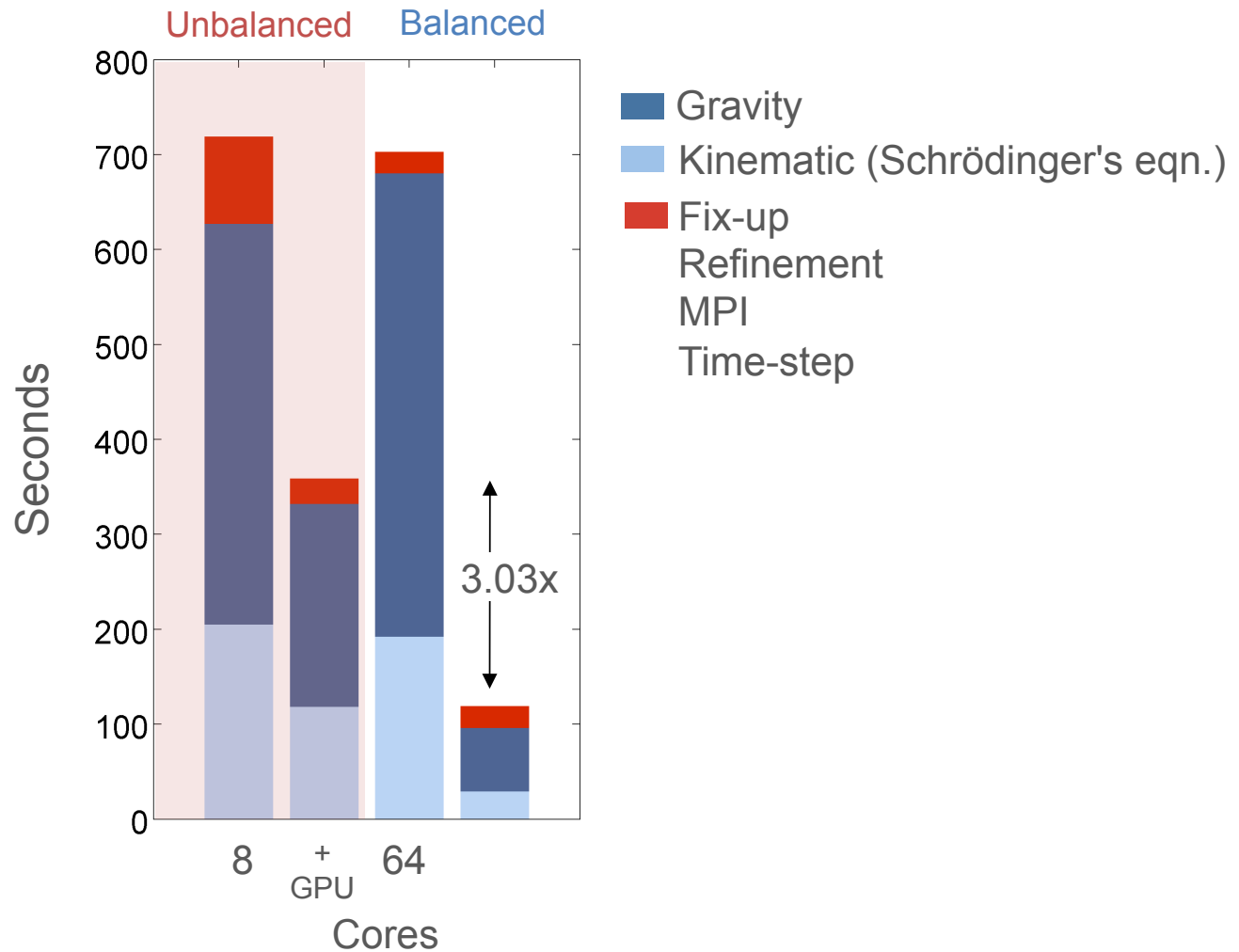
Base level resolution 256^3 to level 7 $32,768^3$



Hemant Shukla,
Hsi-Yu Schive
et al. SC 2011

New Results

Load Balance with Hilbert space filling curve



References

CUDA

<http://developer.nvidia.com/category/zone/cuda-zone>

OpenCL

<http://www.khronos.org/opencv/>

GPGPU

<http://www.gpucomputing.net/>

Advanced topics from Jan 2011 ICCS Summer School

<http://iccs.lbl.gov/workshops/tutorials.html>

Conclusion

If you have parallel code you may benefit from GPUs

In some cases algorithms written on sequential machines may not migrate efficiently and require reexamination and rewrite

If you have short-term goal(s) it may be worthwhile looking into CUDA etc

CUDA provides better performance over OpenCL (Depends)

Most efficient codes optimally use the entire system and not just parts

Heterogeneous computing and parallel programming are here to stay

Number one 2-PetaFlop/s HPC machine in the world (Tianhe-1 in China) is a heterogeneous cluster with 7k+ NVIDIA GPUs and 14k Intel CPUs