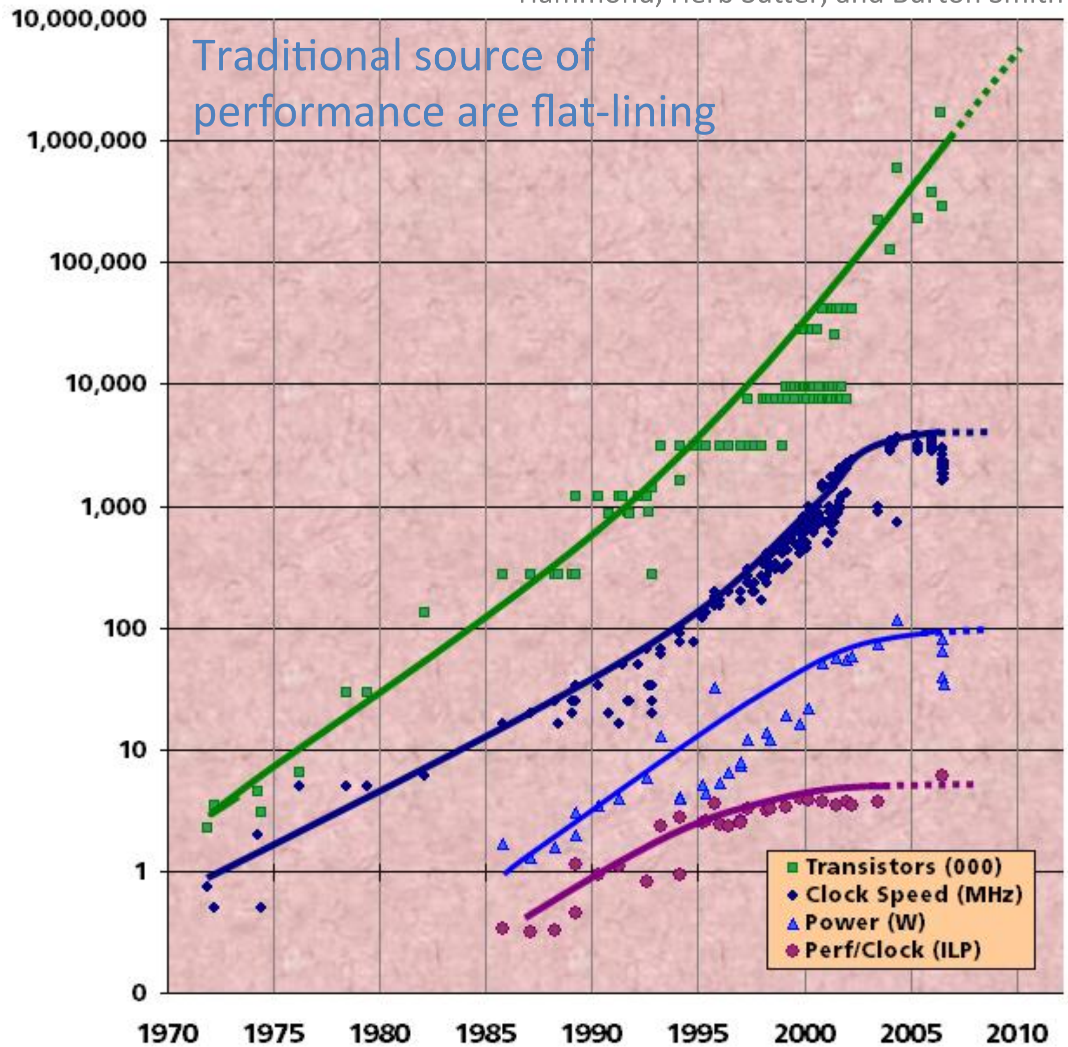
Introduction to CUDA Programming

Hemant Shukla

hshukla@lbl.gov

Trends

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



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

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

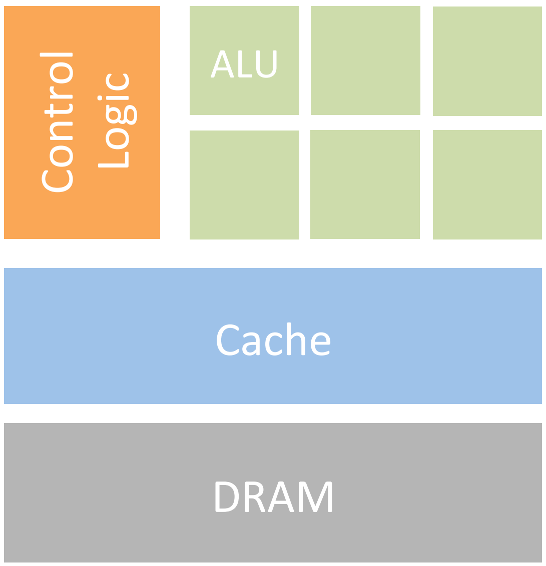
New Parallel Programming models, languages, frameworks, …

Academia

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

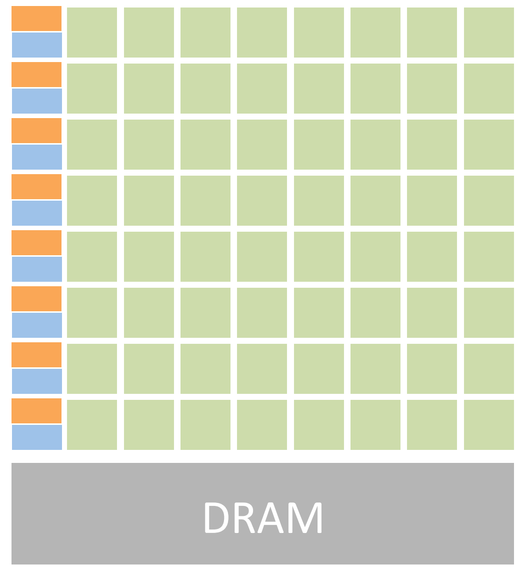
Architectural Differences

CPU



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

GPU



512 cores
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

Applications

X-ray computed tomography

Alain Bonissent et al.

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

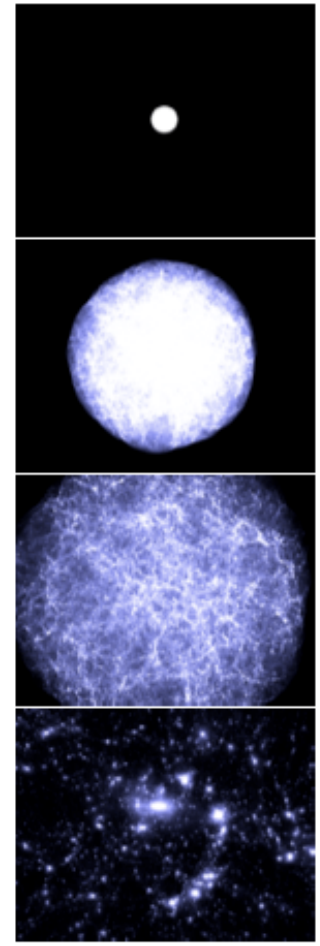
EoR with diesel powered radio interferometry

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

N-body with SCDM



K. Nitadori et al.

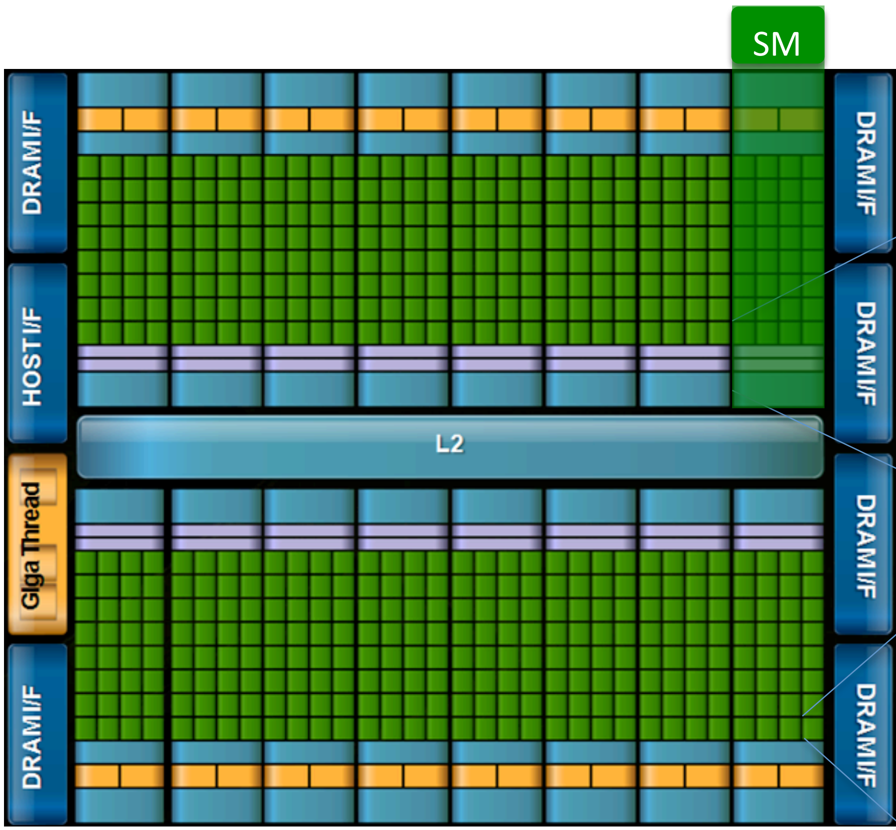
4.5 giga-particles, R = 630 Mpc
2000x more volume than Kawai et al.

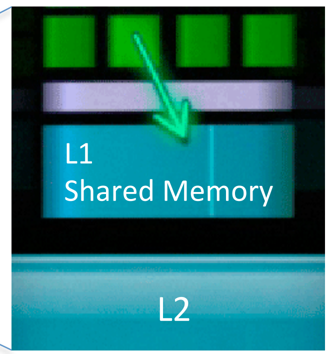
GPU

GPU H/W Example

NVIDIA FERMI

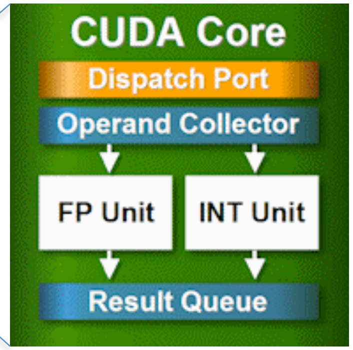
16 Stream Multiprocessors (SM)
512 CUDA cores (32/SM)
IEEE 754-2008 floating point (DP and SP)
6 GB GDDR5 DRAM (Global Memory)
ECC Memory support
Two DMA interface





Reconfigurable L1 Cache and Shared Memory
48 KB / 16 KB

L2 Cache 768 KB



Load/Store address width 64 bits. Can calculate addresses of 16 threads per clock.

Programming Models

CUDA (Compute Unified Device Architecture)

OpenACC

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

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

Parallel Computing Architecture

Application

C/C++ | DX Compute | OpenCL | FORTRAN | Java Python

NVIDIA CUDA Compatible GPU

Libraries | CPU/GPU code

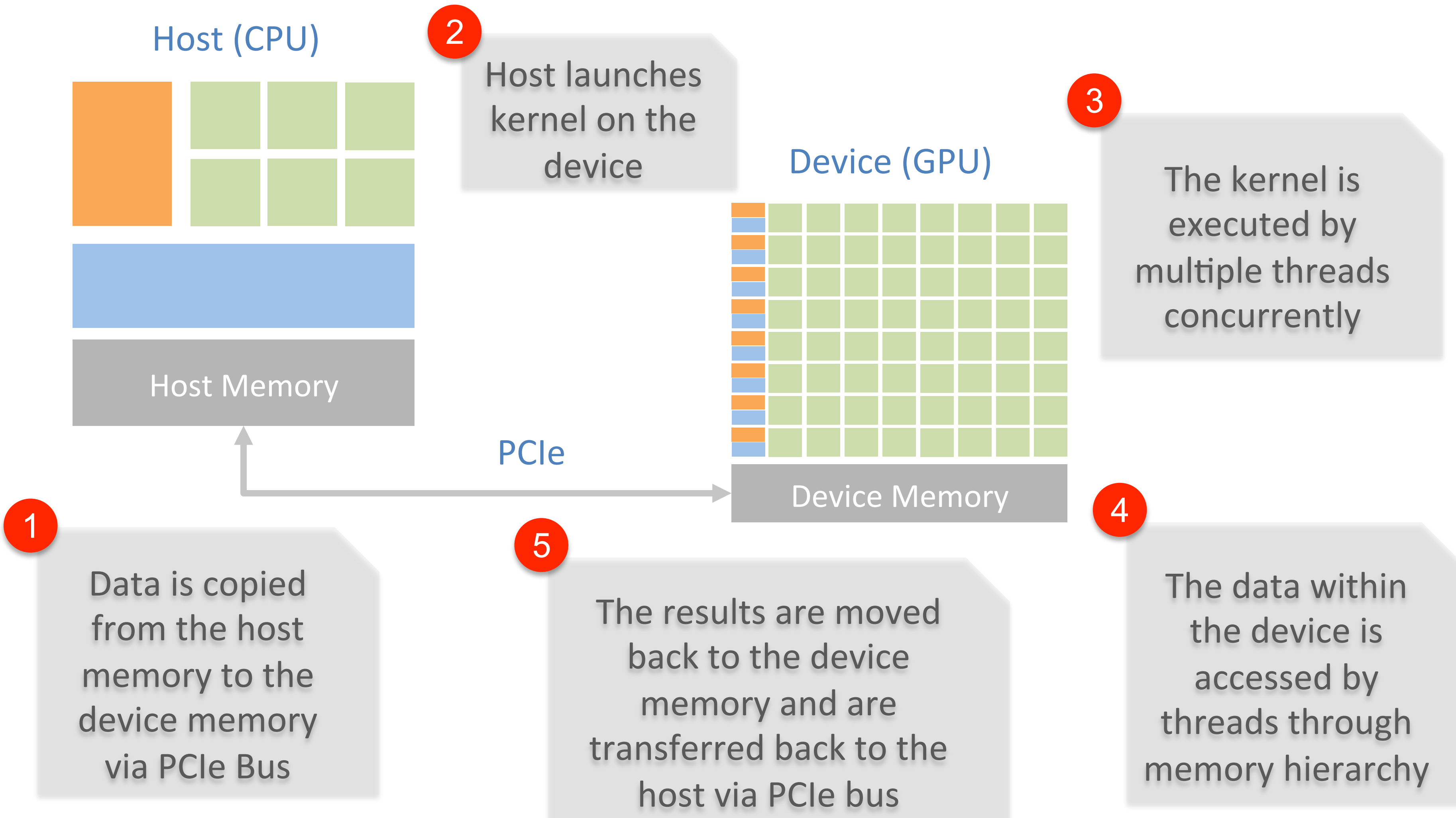
nvcc C/C++ Compiler

NVIDIA Assembly | Host Assembly

CUDA Runtime and Device Driver

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

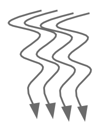
Dataflow



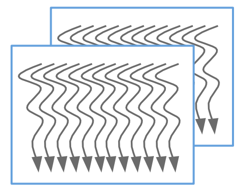
Host (CPU)
Host Memory
PCIe
Device (GPU)
Device Memory
1
Data is copied from the host memory to the device memory via PCIe Bus
2
Host launches kernel on the device
3
The kernel is executed by multiple threads concurrently
4
The data within the device is accessed by threads through memory hierarchy
5
The results are moved back to the device memory and are transferred back to the host via PCIe bus

S/W Abstraction

Threads

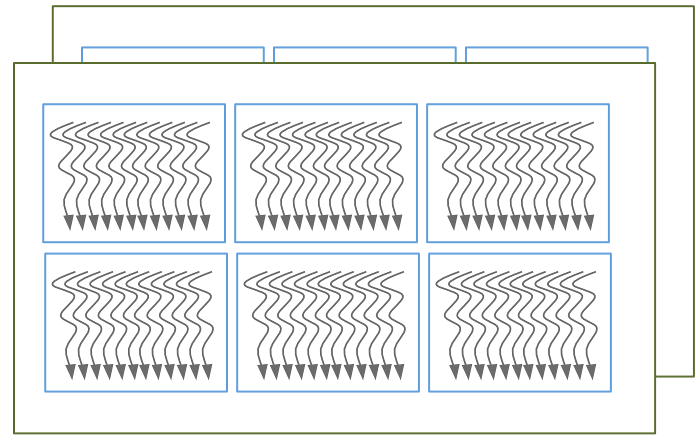


Blocks



512-1024 threads / block

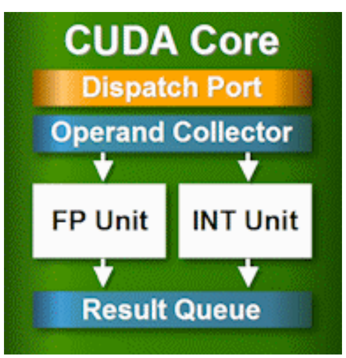
Grids

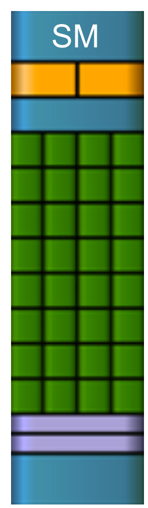


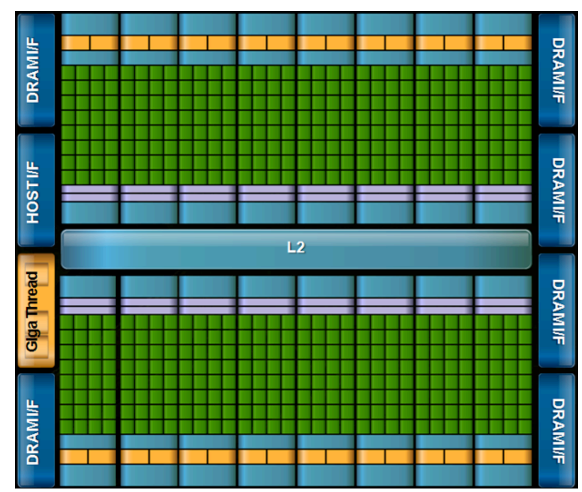
Kernel is executed by threads processed by CUDA Core

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

One grid per kernel with multiple concurrent kernels







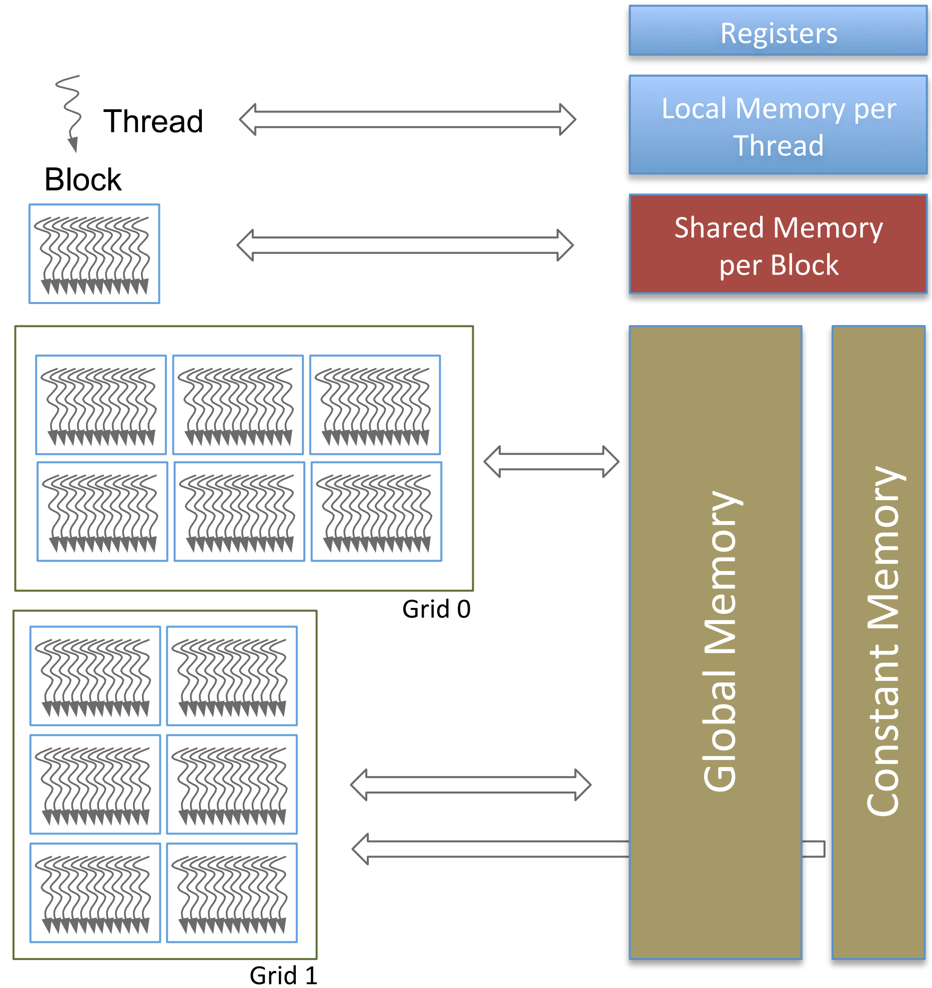
Memory Hierarchy

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 (&noOfDevices);

     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,

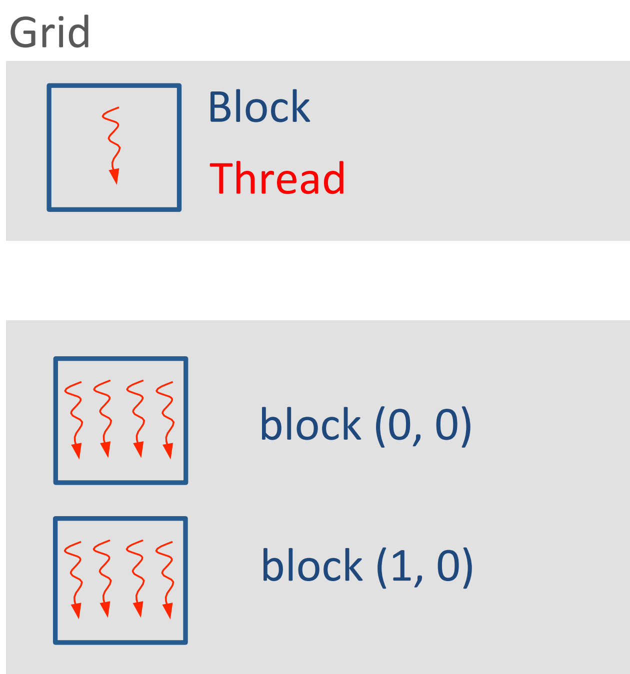
```
someKernel <<<nGrid, nBlocks, sharedMemory, streams>>> (...)
someKernel <<<nGrid, nBlocks>>> (...)
```

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




```
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;
```

Thread Indices

Array traversal

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

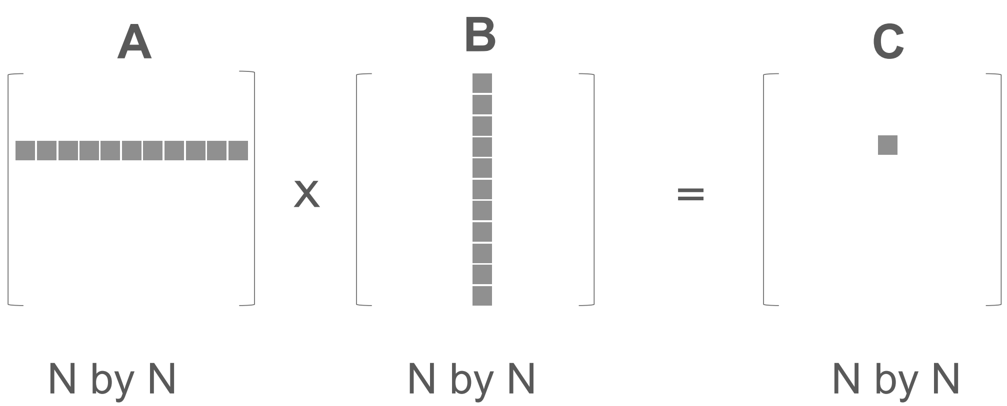
```
blockDim.x  = 4
blockIdx.x  = 0
threadIdx.x = 0, 1, 2, 3
Index       = 0, 1, 2, 3
```

```
blockDim.x  = 4
blockIdx.x  = 1
threadIdx.x = 0, 1, 2, 3
Index       = 4, 5, 6, 7
```

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 × N matrices the matrix-multiplication **C** = **A** • **B** will be equivalent to N^2 independent (hence parallel) inner products.

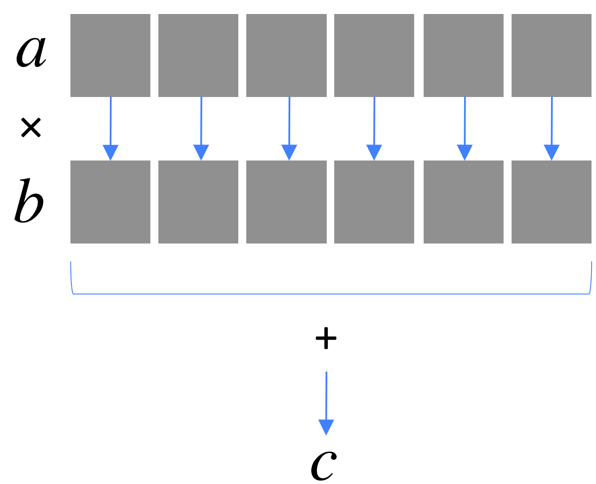
Example

Serial representation

$$c = \sum_i a_i b_i$$

```
double c = 0.0;

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

Simple parallelization strategy



Multiplications are done in parallel

Summation is sequential

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);
}
```

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

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

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 throughout will require 1000/36 ~ 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

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

Atomic Functions

Used to avoid race conditions resulting from thread synchronization and coordination issues.

Multiple threads accessing same address space for read/write simultaneously. Applicable to both shared memory and global memory.

Atomic methods in CUDA guarantee address update without interrupts. Implemented using locks and serialization.

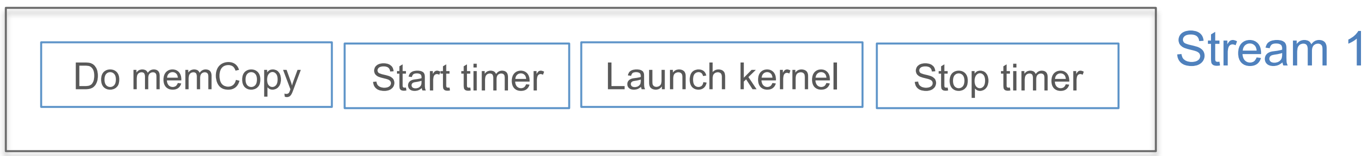
Atomic functions run faster on shared memory than on shared memory.

Atomic functions should also be used judiciously as they serialize the code. Overuse results in performance degradation.

Examples: `atomicAdd, atomicMax, atomicXor...`

CUDA Streams

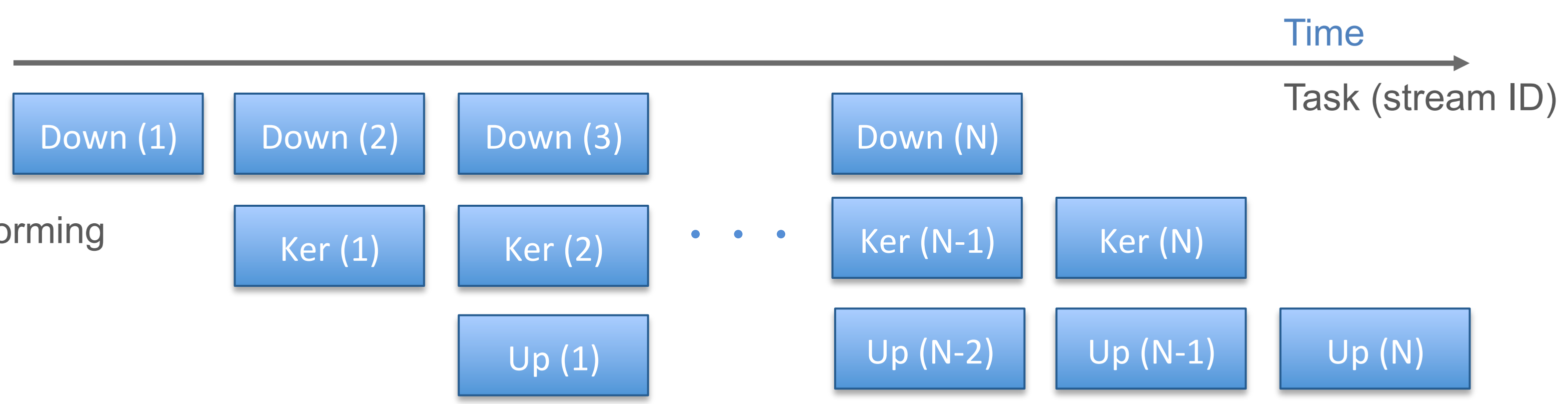
Stream is defined as sequence of device operations executed in order




```
cudaStream_t stream0, stream1;

cudaStreamCreate (&stream0);

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

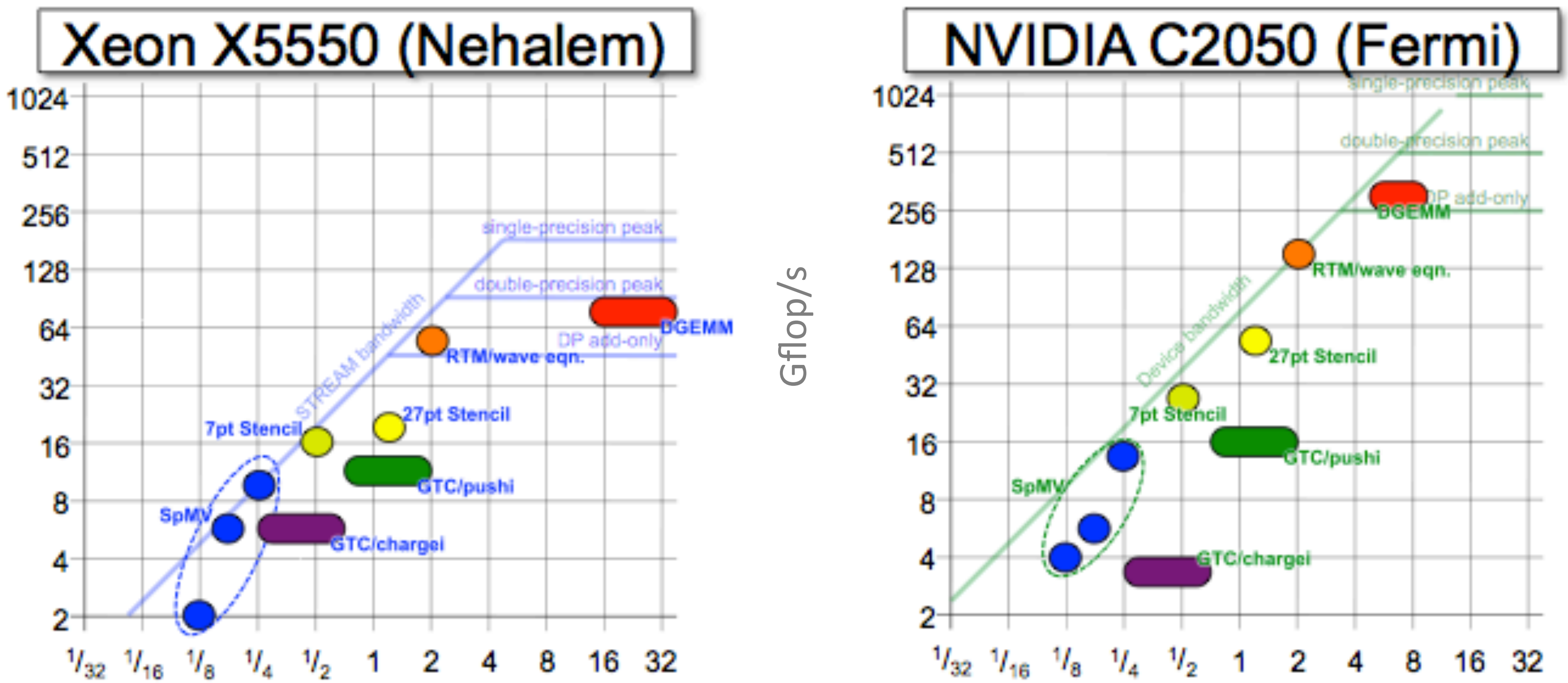
Example

N streams performing
3 tasks



Benchmarks

Relative Performance of Algorithms



Arithmetic Intensity

Courtesy - Sam Williams

References

CUDA
http://developer.nvidia.com/category/zone/cuda-zone

OpenCL
http://www.khronos.org/opencl/

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 two2-PetaFlop/s HPC machine in the world (Tianhe-1 in China) is a heterogeneous cluster with 7k+ NVIDIA GPUs and 14k Intel CPUs

Algorithms

Lessons from ICCS Tutorials by Wen-Mei Hwu

Think Parallel

Promote fine grain parallelism

Consider minimal data movement

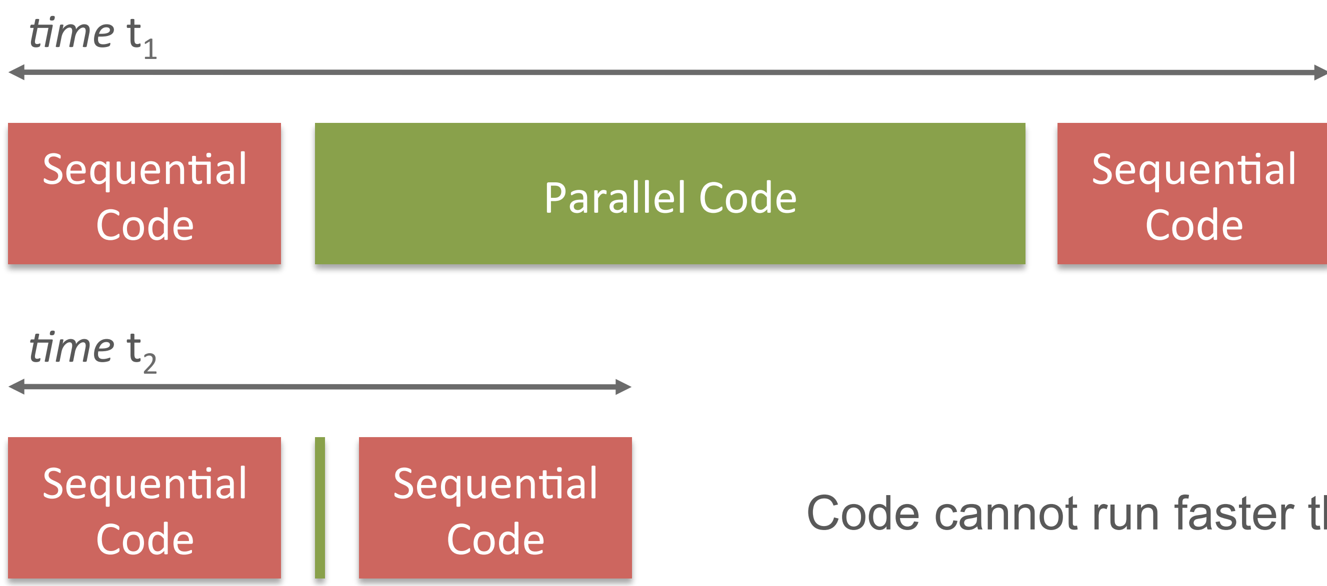
Exploit parallel memory access patterns

Data layout

Data Blocking/Tiling

Load Balance

Amdhal's Argument



Code cannot run faster than time t_2

If X is the serialized part of the code then speedup cannot be greater than 1/1-X no matter how many cores are added.

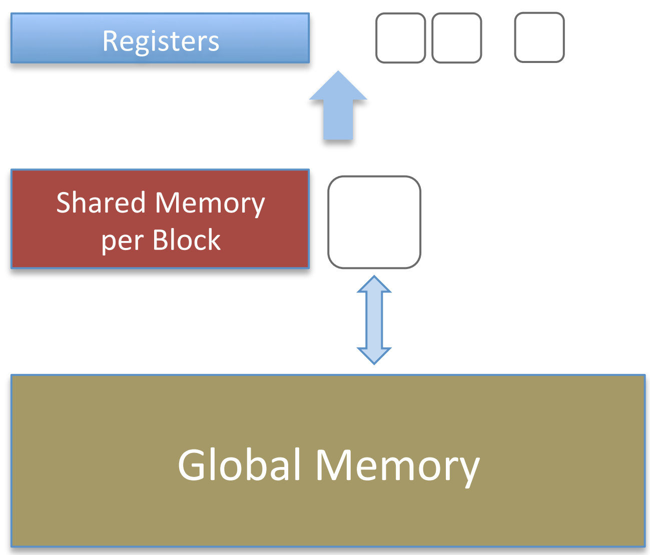
Blocking

Also known as Tiling.

Basic idea is to move blocks/tiles of commonly useable data from global memory into shared memory or registers memory.

Register Tiling

Shared Memory Tiling



Reuse computed results

Get data blocks for thread to share

Blocking / Tiling Technique

Focused Access pattern

Identify block/tile of global memory data to be accessed by threads.

Load the data into the fast memory (Shared, register)

Get the multithreads to use the data

Assure barrier synchronization

Repeat (move to next block, next iterations etc.)

Make the most of one load of data into fast memory

Variables on Memory

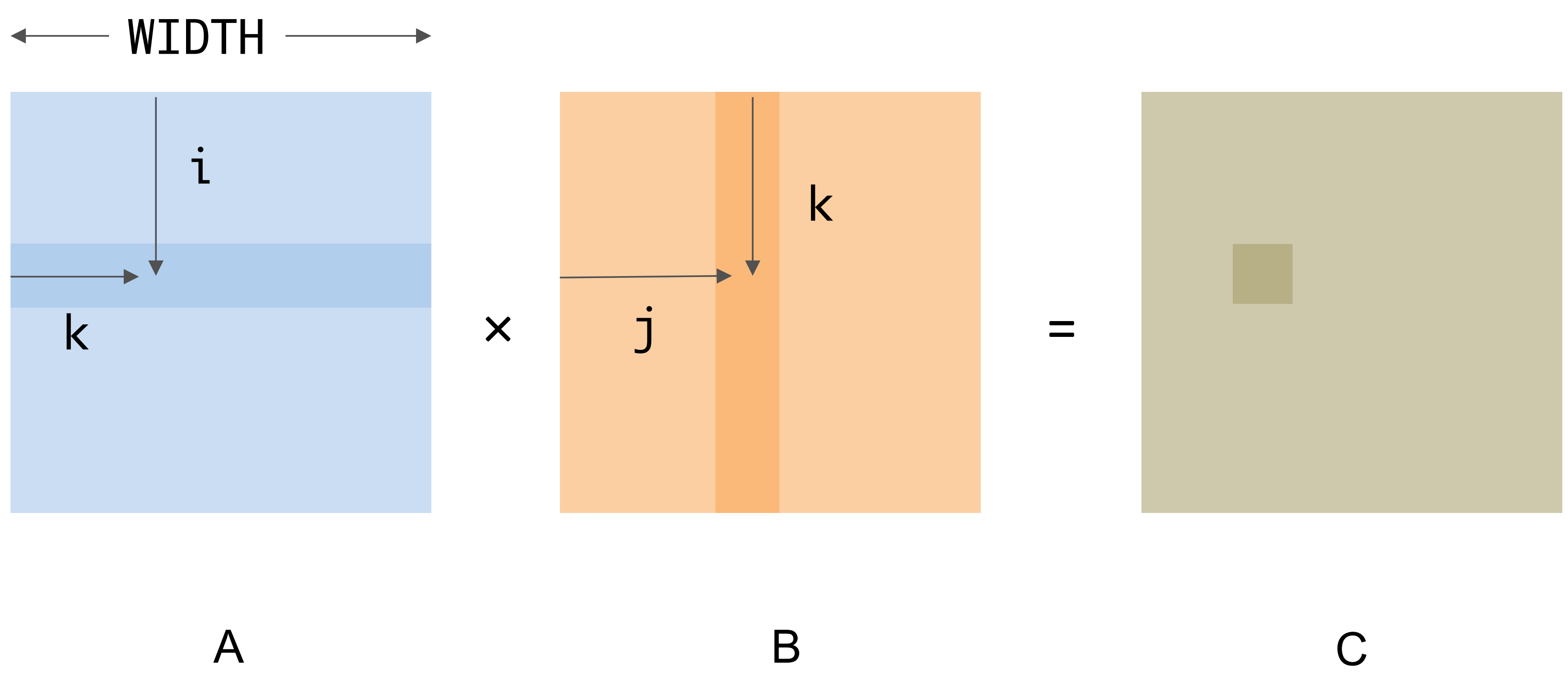
CUDA Variable Type Qualifiers

```
__device__ __shared__   int SharedVar;
__device__              int GlobalVar;
__device__ __constant__ int ConstantVar;
```

Kernel variables without any qualifiers reside in a registe with an exception for arrays that reside in local memory

Matrix Multiplication

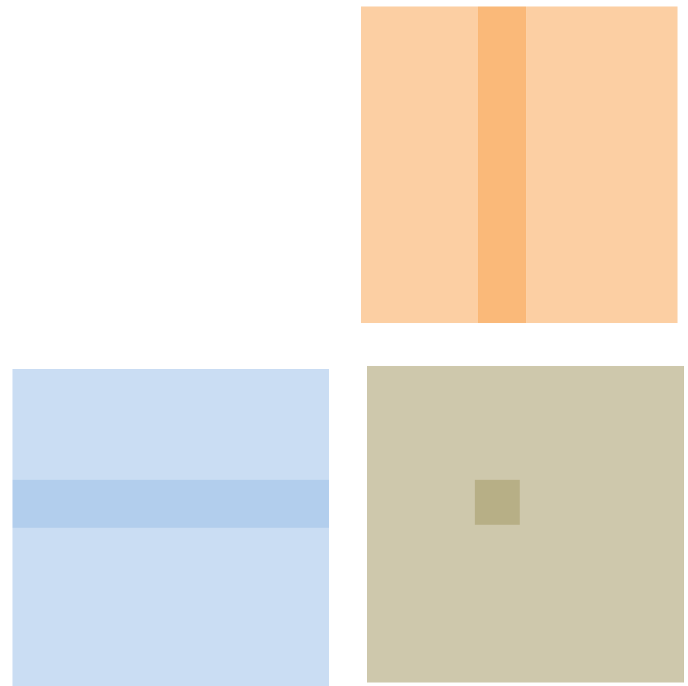
Example



Matrix Multiplication…

CPU Version

```
void matrixMultiplication ( float* A, float* B, float* C, int WIDTH)
{

    for (i → 0 : WIDHT)

        for (j → 0 : WIDTH)

            for (k → 0 : WIDTH)

                a = A_i;
                b = B_j;
                sum += a * b;

        C_ij = sum;
}
```



Matrix Multiplication...

GPU Version (Memory locations)

Constant memory

```
__global__ void matrixMultiplication (float* A, float* B, float* C, int WIDTH)
{
```

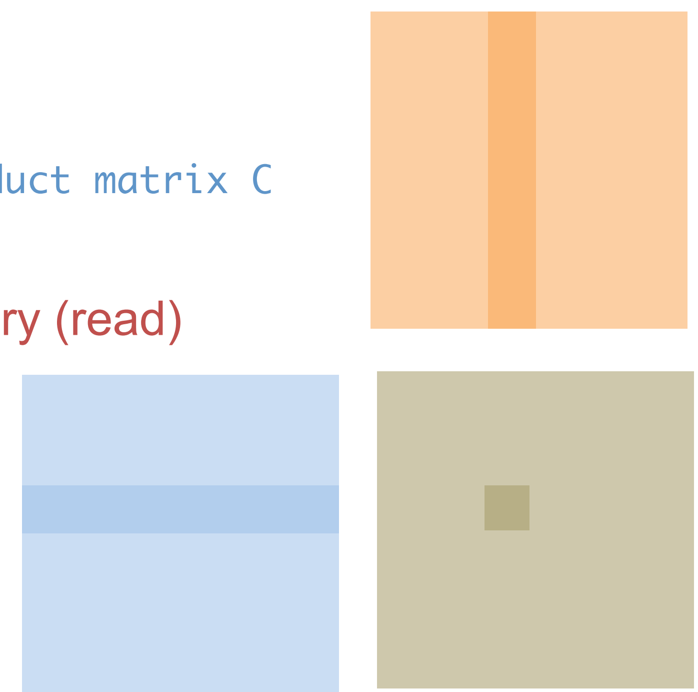
Shared memory

```
    int i = blockIdx.y * WIDTH + threadIdx.y;
    int j = blockIdx.x * WIDTH + threadIdx.x;

    // each thread computes one element of product matrix C
    for (k → 0 : k)

        sum += A[i][k] * B[k][j];
```

Global memory (read)

```
    C[i][j] = sum;
}
```

Global memory (write)



Matrix Multiplication…

Kernel analysis

2 floating point read accesses, 2 x 4 bytes = 8 bytes per one multiply and add that is 2 floating point operations per second (add and multiply). Hence the throughput is 8 bytes / 2 = 4B / FLOPs.

Theoretical peak of Fermi is 530 FLOPs

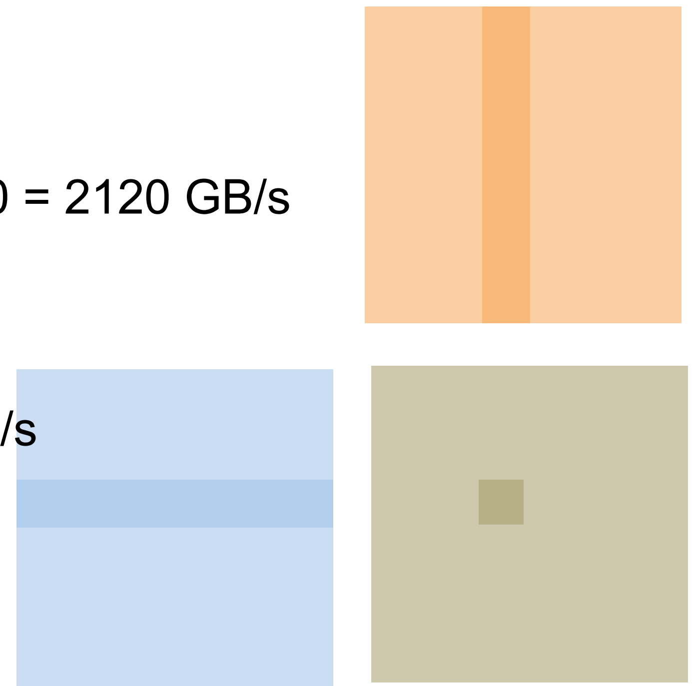
To achieve peak will require bandwidth of 4 x 530 = 2120 GB/s

The actual bandwidth is 177GB/s

With this bandwidth it yields 177/4 = 44.25 FLOP/s

About 12 times below peak performance.

In practice it will be slower.



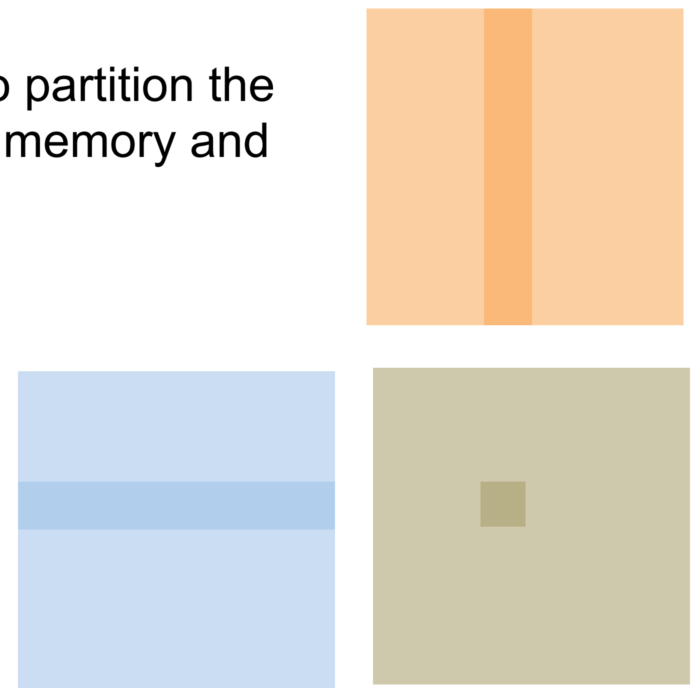
Matrix Multiplication…

How to speed up?

BLOCKING

Load data into shared memory and reuse

Since the Shared memory size is small it helps to partition the data in equal sized blocks that fit into the shared memory and reuse.



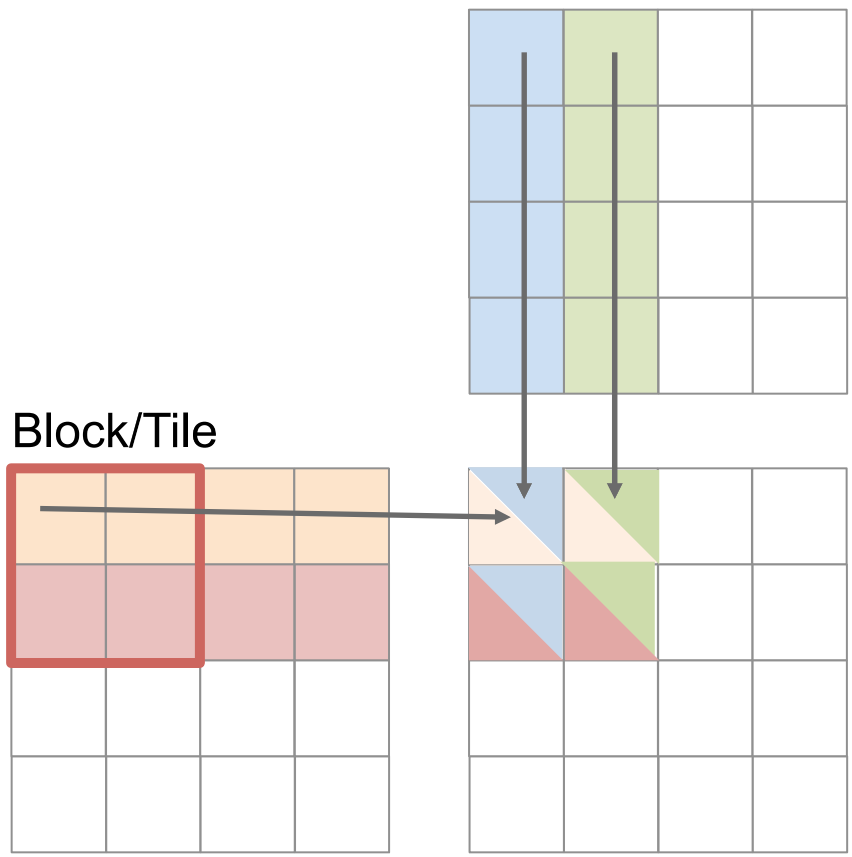
Matrix Multiplication...

Partial rows and columns are loaded in shared memory

One row is reused to calculate two elements.

Multiple blocks are executed in parallel.

For a 16 x 16 tile width the global memory loads are reduced by 16.



Matrix Multiplication…

	Tile 1			Tile 2		
$T_{0,0}$	$A_{0,0}$ ↓ $A\_S_{0,0}$	$B_{0,0}$ $B\_S_{0,0}$	$C_{0,0} = A\_S_{0,0} * B\_S_{0,0} + A\_S_{1,0} * B\_S_{0,1}$	$A_{2,0}$ $A\_S_{0,0}$	$B_{0,2}$ $B\_S_{0,0}$	$C_{0,0} = A\_S_{0,0} * B\_S_{0,0} + A\_S_{1,0} * B\_S_{0,1}$
$T_{1,0}$	$A_{0,0}$ $A\_S_{1,0}$	$B_{0,0}$ $B\_S_{1,0}$	$C_{1,0} = A\_S_{0,0} * B\_S_{1,0} + A\_S_{1,0} * B\_S_{1,1}$	$A_{3,0}$ $A\_S_{1,0}$	$B_{1,2}$ $B\_S_{1,0}$	$C_{1,0} = A\_S_{0,0} * B\_S_{1,0} + A\_S_{1,0} * B\_S_{1,1}$
$T_{0,1}$	$A_{0,1}$ $A\_S_{0,1}$	$B_{0,1}$ $B\_S_{0,1}$	$C_{0,1} = A\_S_{0,1} * B\_S_{0,0} + A\_S_{1,1} * B\_S_{0,1}$	$A_{2,1}$ $A\_S_{0,1}$	$B_{0,3}$ $B\_S_{0,1}$	$C_{0,1} = A\_S_{0,1} * B\_S_{0,0} + A\_S_{1,1} * B\_S_{0,1}$
$T_{1,1}$	$A_{1,1}$ $A\_S_{1,1}$	$B_{1,1}$ $B\_S_{1,1}$	$C_{1,1} = A\_S_{0,1} * B\_S_{1,0} + A\_S_{1,1} * B\_S_{1,1}$	$A_{3,1}$ $A\_S_{1,1}$	$B_{1,3}$ $B\_S_{1,1}$	$C_{1,1} = A\_S_{0,1} * B\_S_{1,0} + A\_S_{1,1} * B\_S_{1,1}$

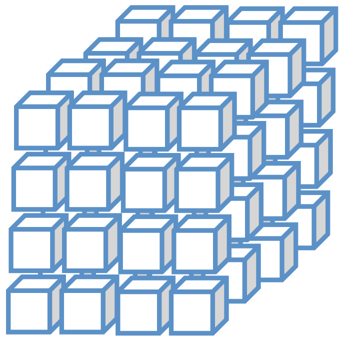
Threads

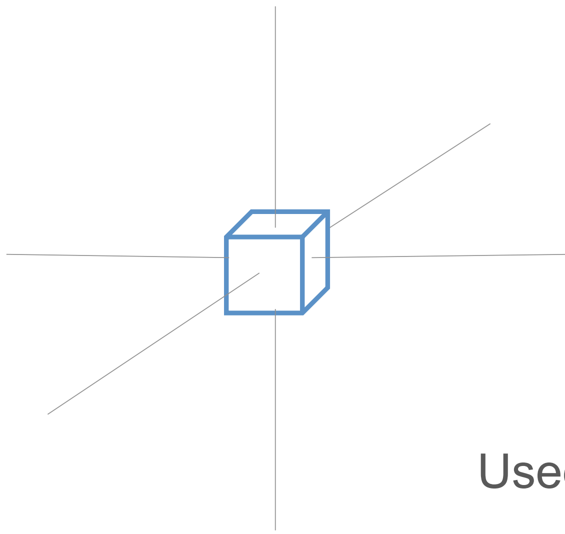
Time

Matrix Multiplication…

```
__global__ void matrixMultiplication(float* A, float* B, float* C, int WIDTH,
                                     int TILE_WIDTH)
{
     __shared__float A_S[TILE_WIDTH][TILE_WIDTH];
     __shared__float B_S[TILE_WIDTH][TILE_WIDTH];

     int bx = blockIdx.x;  int by = blockIdx.y;
     int tx = threadIdx.x; int ty = threadIdx.y;

// row and column of the C element to calculate
     int Row = by * TILE_WIDTH + ty;
     int Col = bx * TILE_WIDTH + tx;
     float sum = 0;
// Loop over the A and B tiles required to compute the C element
     for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collectively Load A and B tiles from the global memory into shared memory
          A_S[tx][ty] = A[(m*TILE_WIDTH + tx)*Width+Row];
          B_S[tx][ty] = B[Col*Width+(m*TILE_WIDTH + ty)];
          __syncthreads();

          for (int k = 0; k < TILE_WIDTH; ++k)
               sum += A_S[tx][k] * B_C[k][ty];
          __synchthreads();
     }
     C [Row*Width+Col] = sum;
}
```

7-Point Stencil





Used for PDEs, Convolution etc.

7-Point Stencil …

Conceptually all points can be upgraded in parallel.

Each computations performs global sweep of entire data.

Memory bound.

Challenge is to parallelize without overusing memory bandwidth.

7-Point Stencil …

Calculate values along one axis.

Traversing the axis 3 values are needed along the axis

Keep the three values in the register for next iteration

This is called Register Tiling

For 7-point there are 2 in the register so only 5 access will be needed.

A combination of register and block tiling should give 7x speed up.

In reality 4-5x because halos have to be considered.

Questions?

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

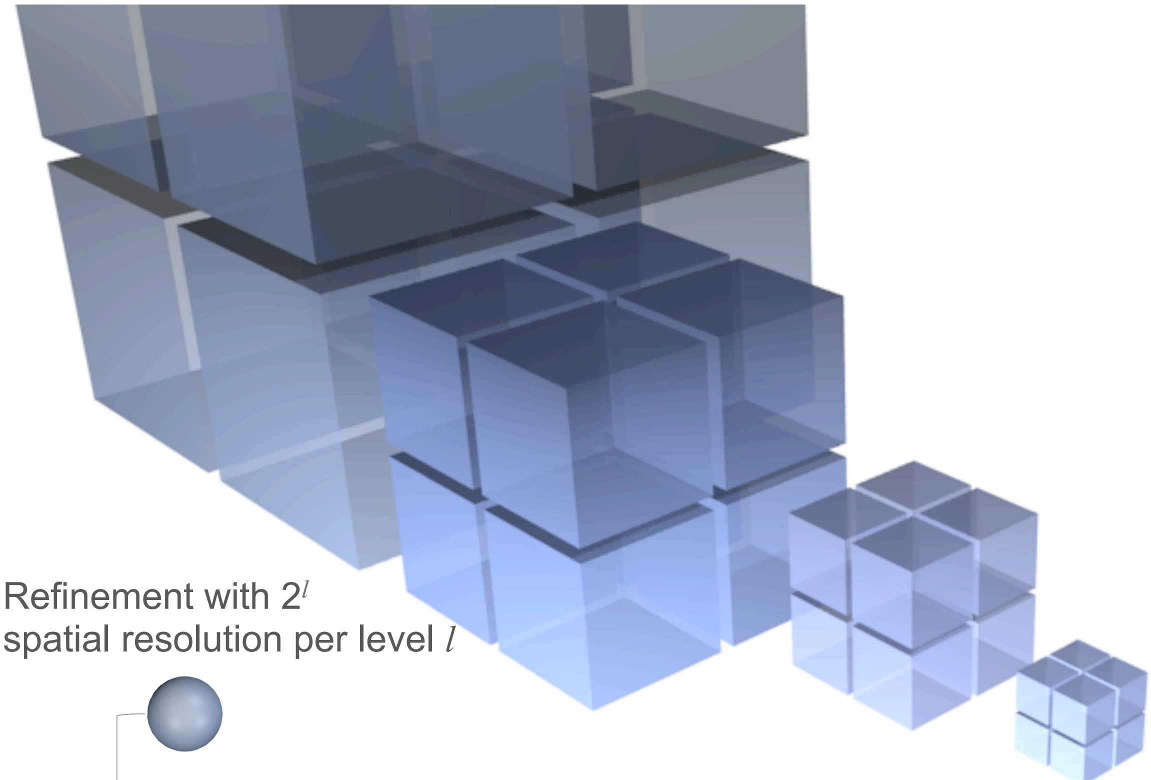
Generalized GAMER codebase to multi-science framework

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

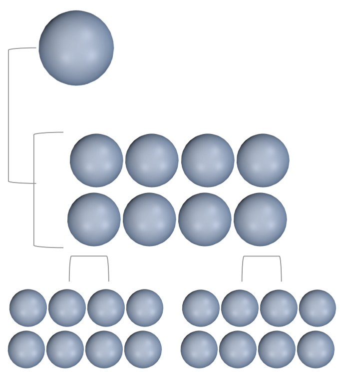
Collect performance, memory access, power consumption and various other metrics for broader user base

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

Adaptive Mesh Refinement

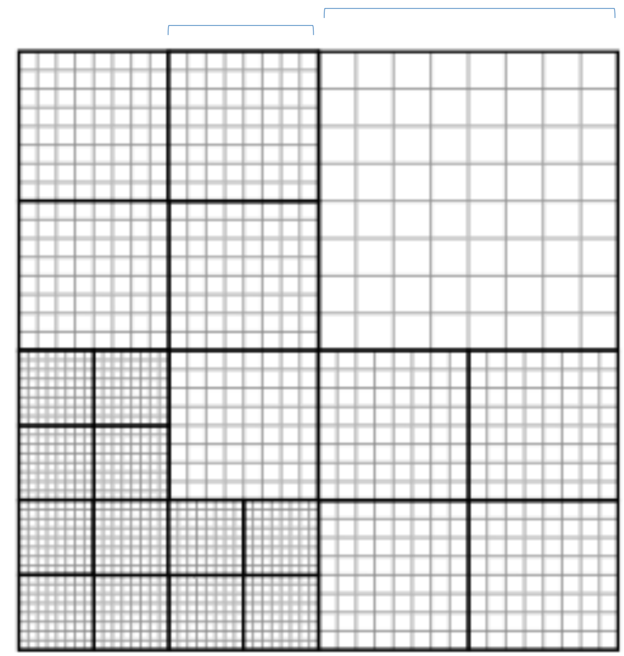


Refinement with 2^l spatial resolution per level l



Data stored in Octree data structure

2D Patch



8^3 cells per patch

Identical spatial geometry (same kernel)

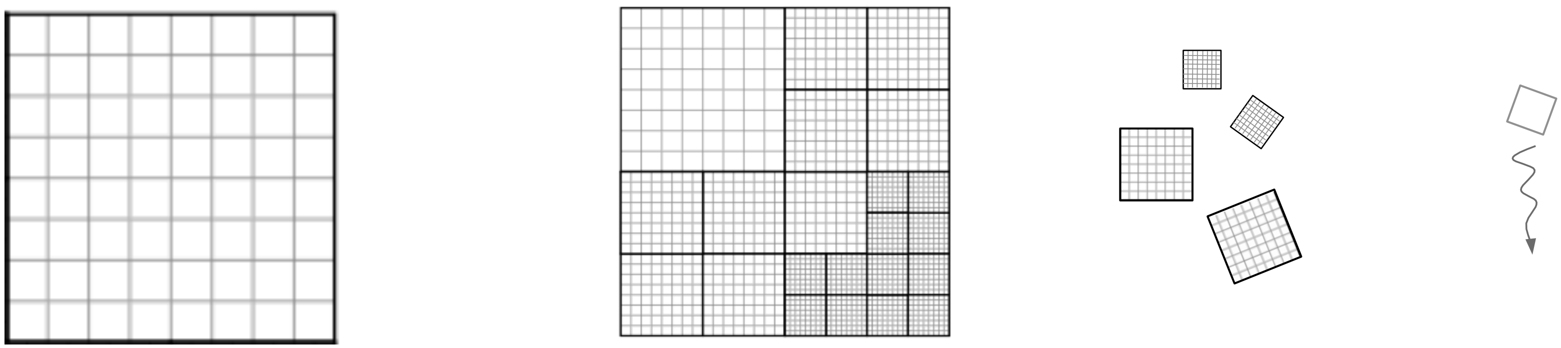
Uniform and individual time-steps

Construct and Dataflow

GAMER Codebase C++/CUDA, MPI, OpenMP

AMR, Framework, Libraries

Solvers Poisson, Hydro, Custom, …



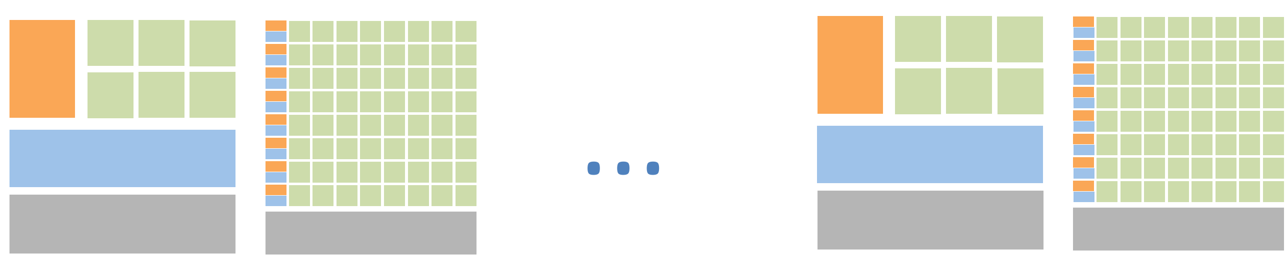
Problem domain covered with coarse patch on CPUs

User defined refinement, spatial averaging, flux correction on CPUs

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

Time Steps

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

GAMER Framework

Allows for adding custom/new solvers to the codebase

New Solver inherits

Async memcpy, concurrent execution, MPI and OpenMP optimization

New Solver implements

The size of computational stencil

An optimized CPU version of the implementation

An optimized GPU version of the implementation

CUDA thread blocks and stream objects

Multi-Science

Cosmological Large-scale Structure

Gravitational potential

$$\nabla^2 \phi(\vec{x}) = 4\pi G a[\rho(\vec{x}) - \rho_b(\vec{x})]$$

Bosonic Dark Matter

Schrodinger-Poisson equation

$$i\hbar \frac{\partial \psi}{\partial t} = -\frac{\hbar^2}{2a^2 m}\nabla^2 \psi + mV\psi$$

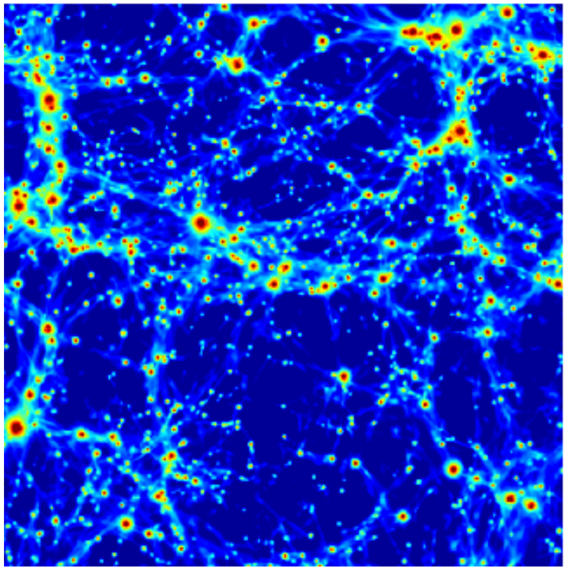
Gravitational Lensing Potential

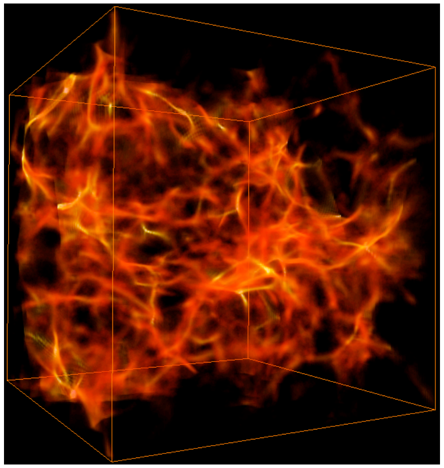
Lens equation and mass relationship

$$\vec{u} = \vec{x} - \nabla \phi(\vec{x})$$

$$\nabla^2 \phi(\vec{x}) = \Sigma(\vec{x}) / \Sigma_{cr}$$

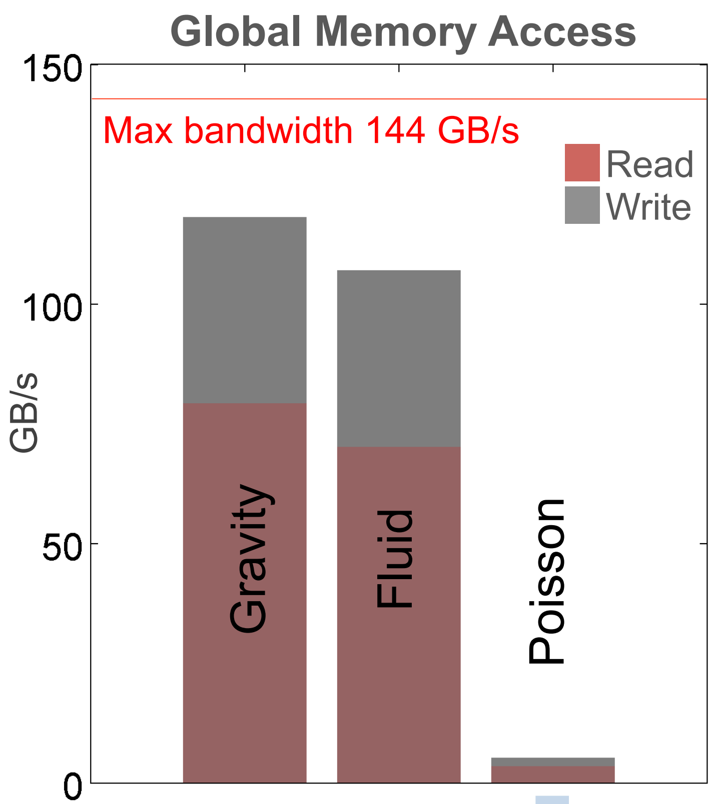
Effective resolution 8192^3



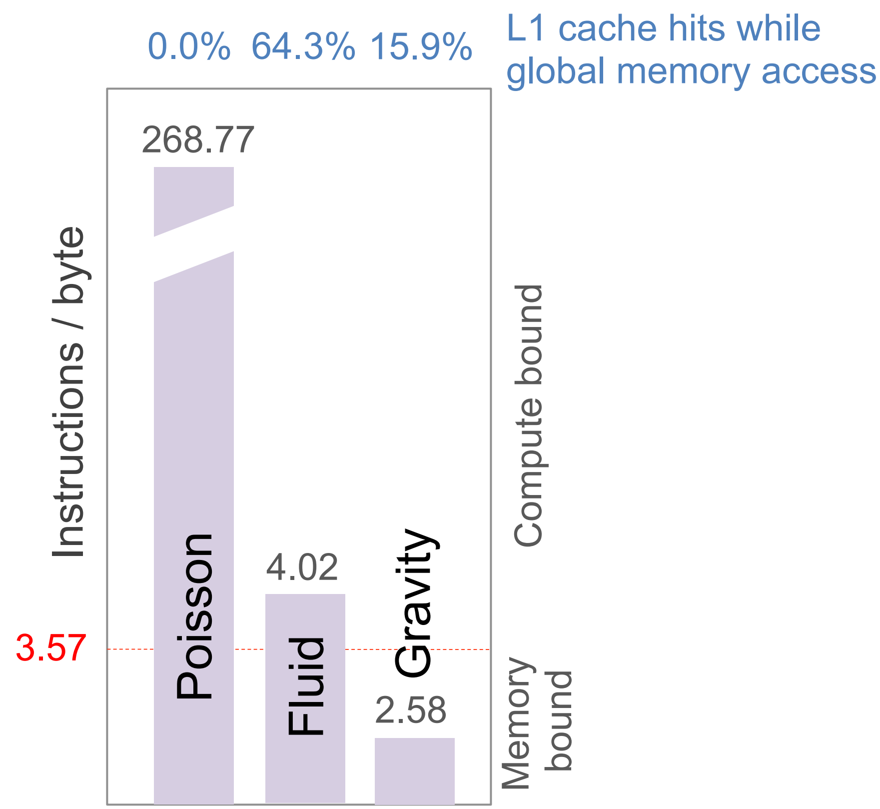


Structure due to dark matter model in early universe

Kernel Analysis



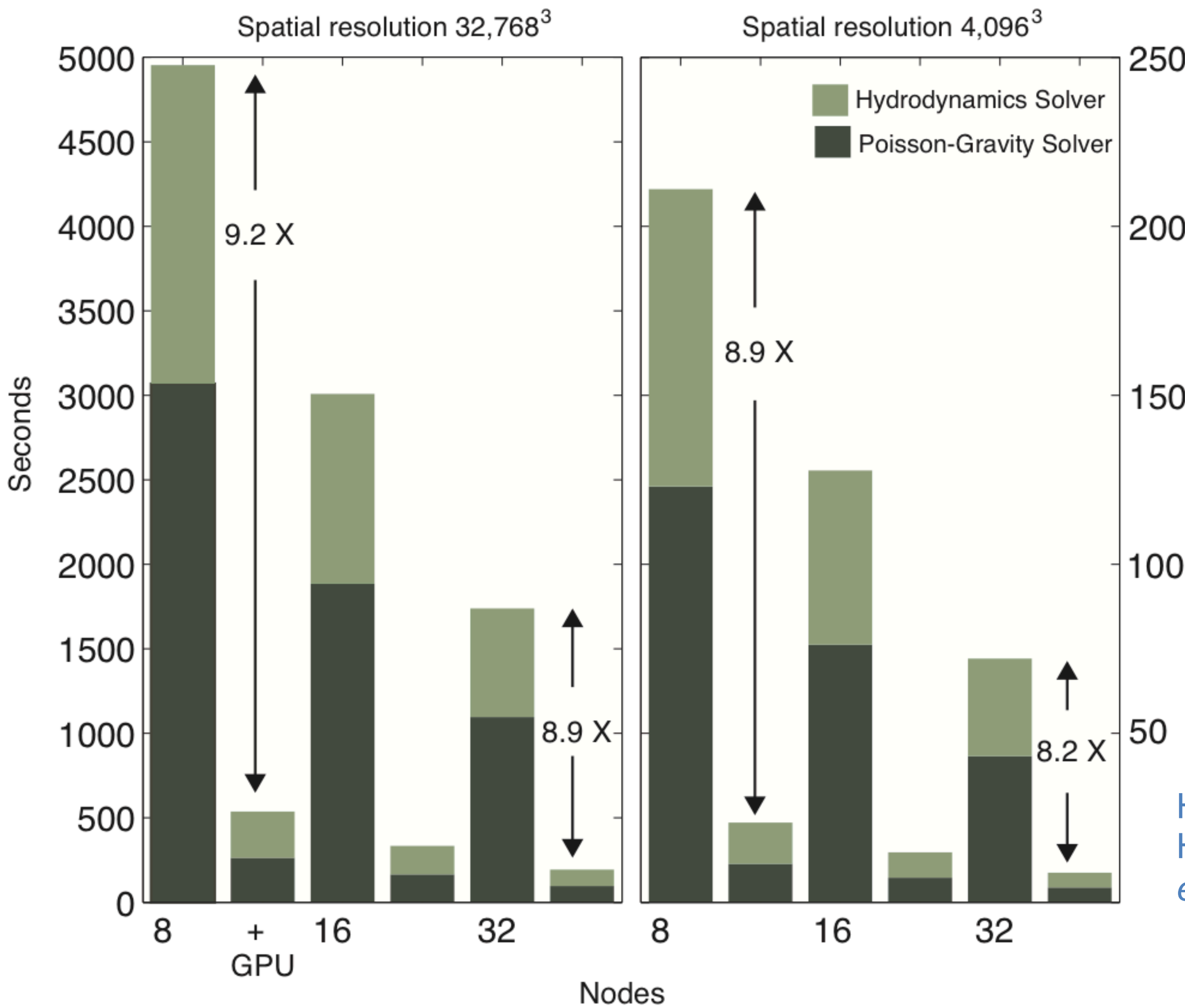
Intensive use of shared memory



SOR takes 20-30 iterations to converge

Results

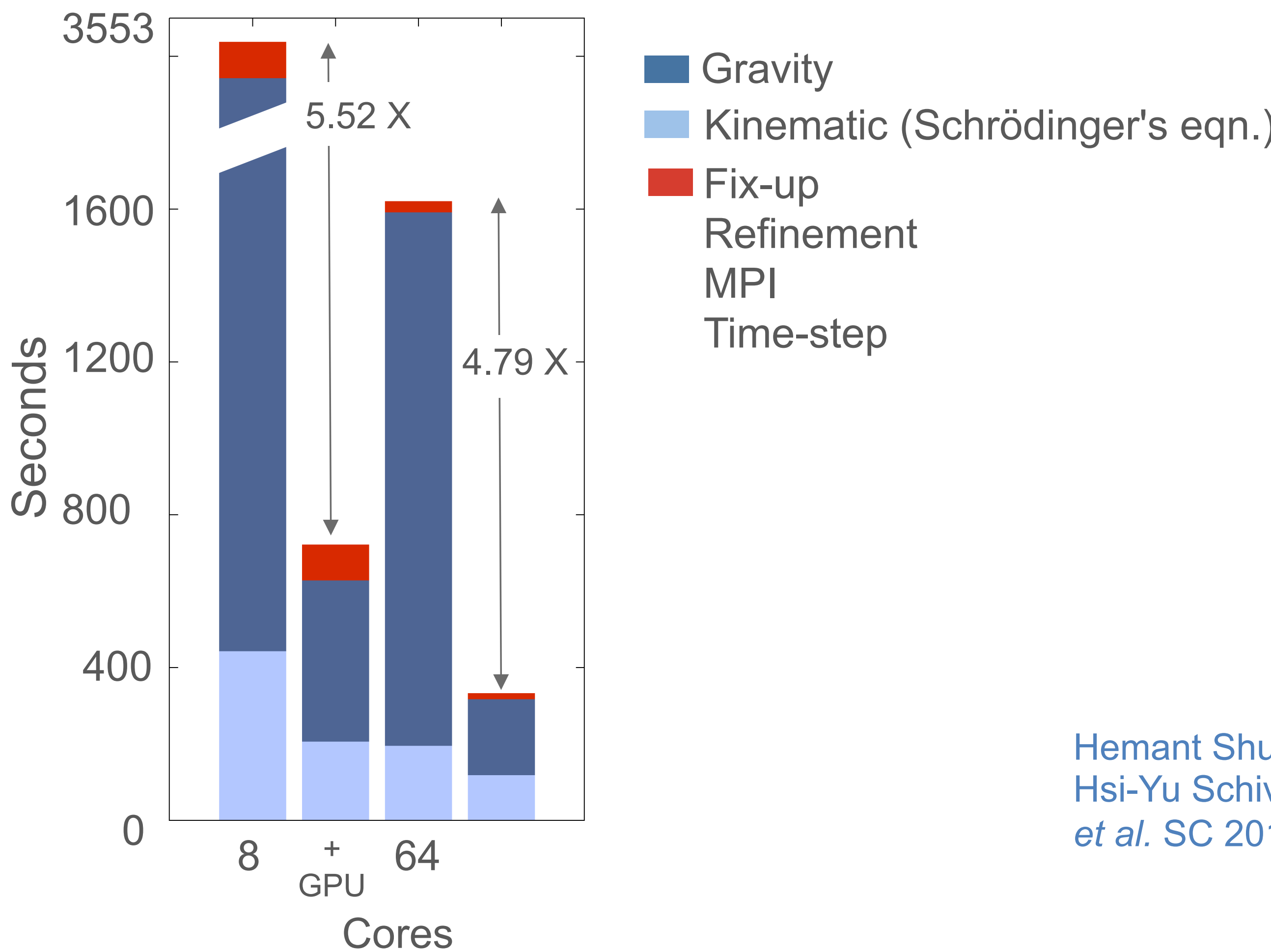
Large scale Cosmological Simulations with GAMER



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

Results

Bosonic Dark Mater 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