Graphs on Manycores

Lecture 23

CSCI 4974/6971

1 Dec 2016

Today's Biz

1. **Reminders**
2. Review
3. Graphs on Manycores

Reminders

- Assignment 6: due date Dec 8th
- Final Project Presentation: December 8th
- Project Report: December 11th
  - Intro, Background and Prior Work, Methodology, Experiments, Results
  - Submit as PDF
- Office hours: Tuesday & Wednesday 14:00-16:00 Lally 317
  - Or email me for other availability

Today's Biz

1. Reminders
2. **Review**
3. Graphs on Manycores

Quick Review

Graphs as Matrices:

- Graph edges $\rightarrow$ adjacency matrix
- Graph algorithms $\rightarrow$ linear algebraic operations
  - BFS $\rightarrow A^T x$ iteratively
- Why?
  - Develop abstraction for domain scientists to work with
  - Matrix and vector operations have been highly studied and optimized for all architectures
  - Allow developed algorithms to be machine-independent ("future-proof")

Today's Biz

1. Reminders
2. Review
3. **Graphs on Manycores**

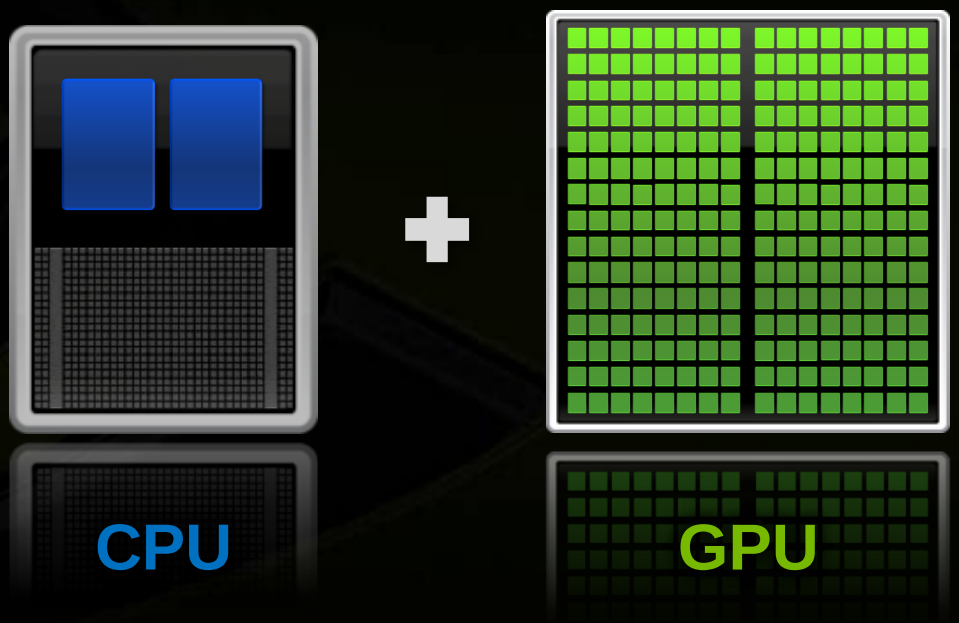
CUDA Overview

Cliff Woolley, NVIDIA

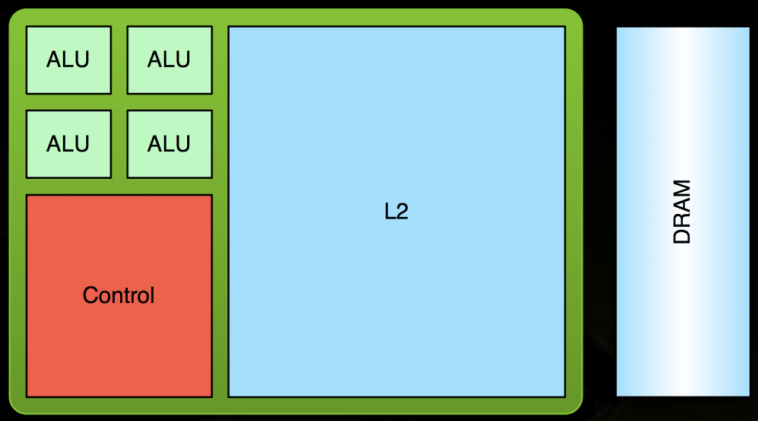

CUDA Overview
Cliff Woolley, NVIDIA
Developer Technology Group
NVIDIA

GPGPU Revolutionizes Computing

Latency Processor + Throughput processor

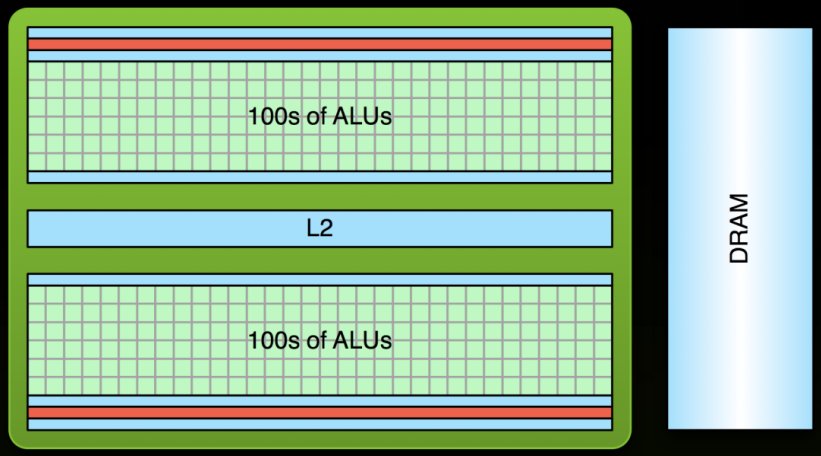


Low Latency or High Throughput?



CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

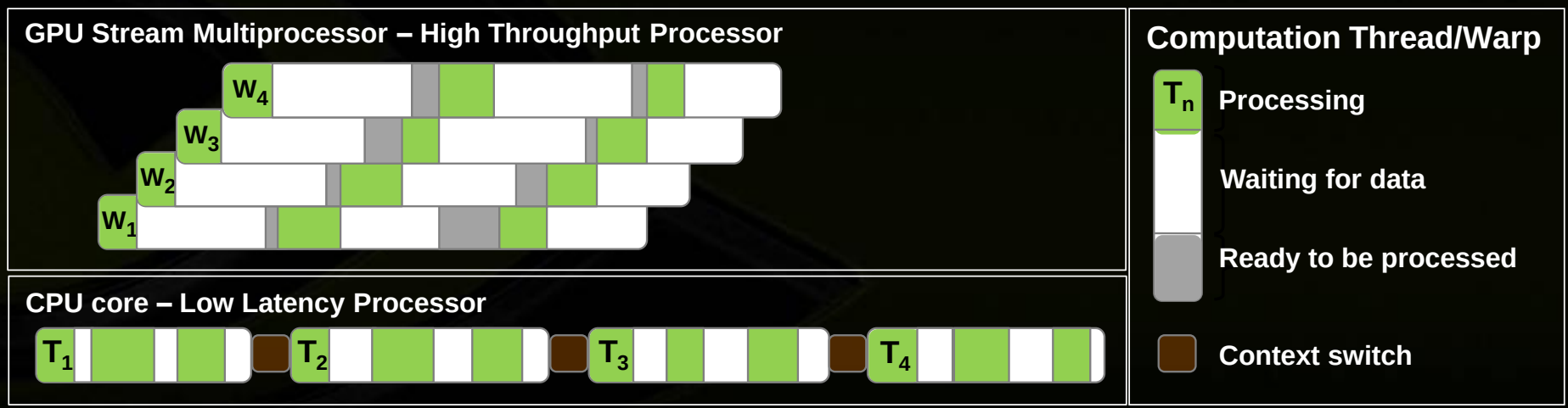


GPU

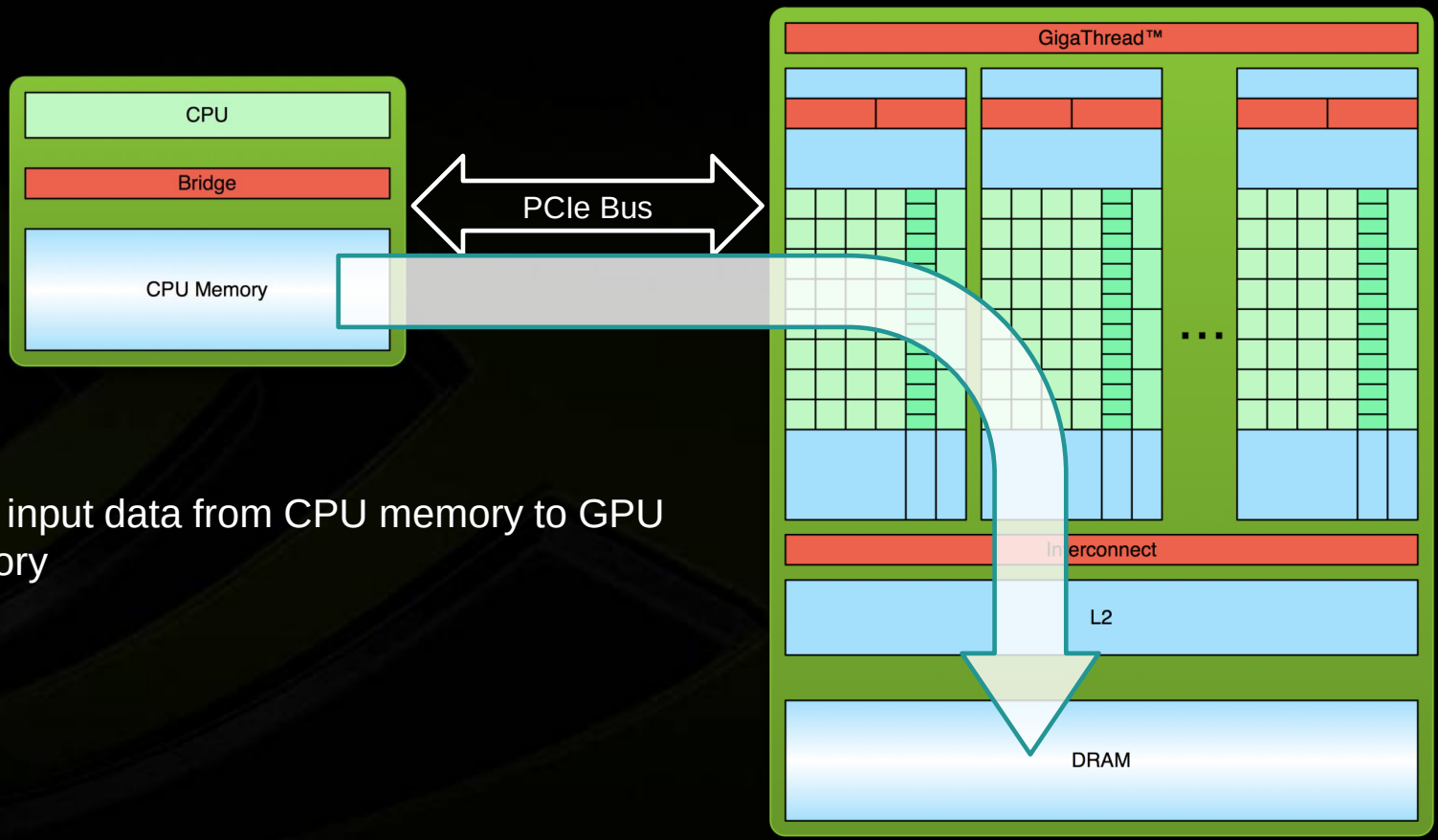
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps

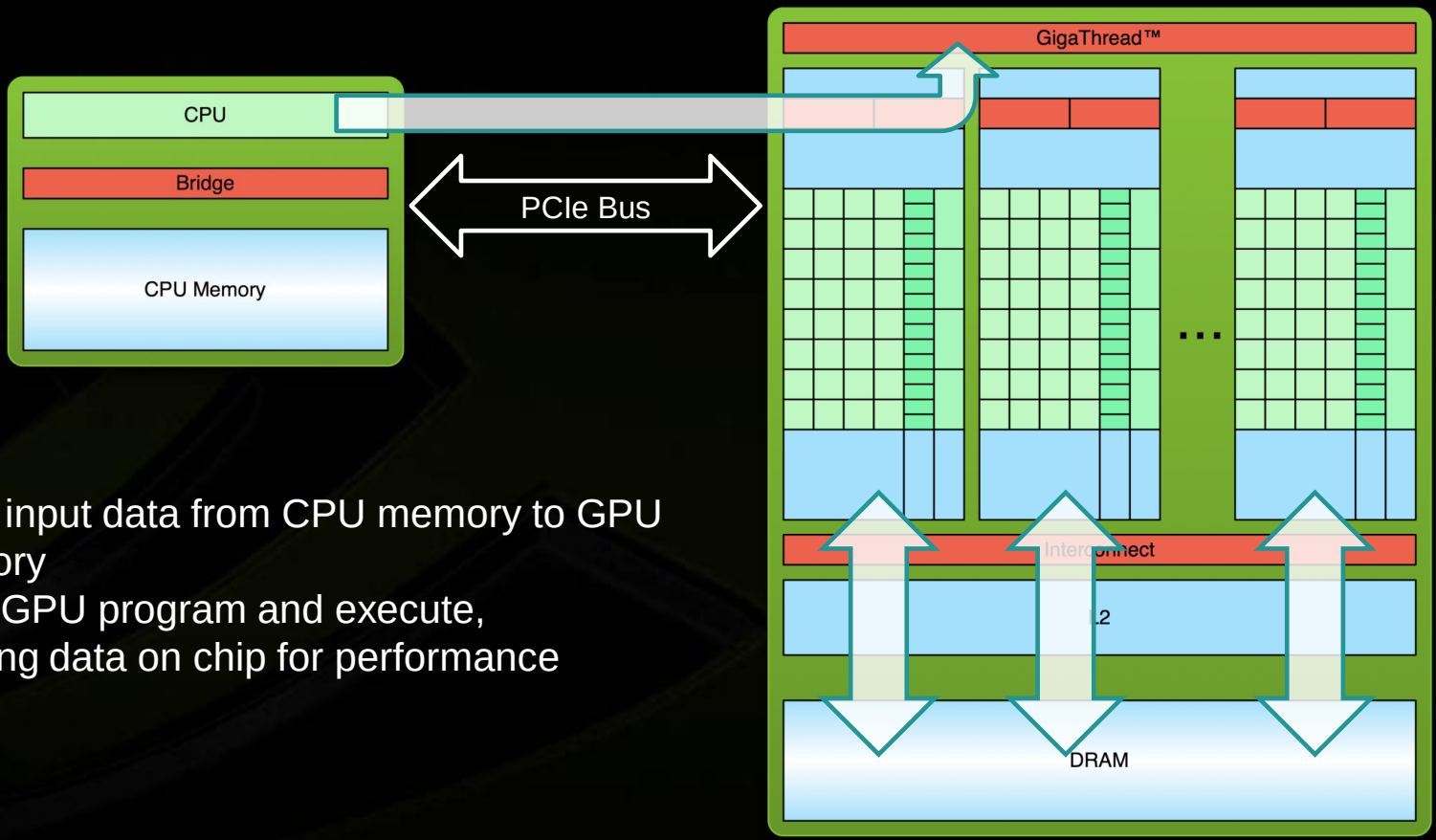


Processing Flow



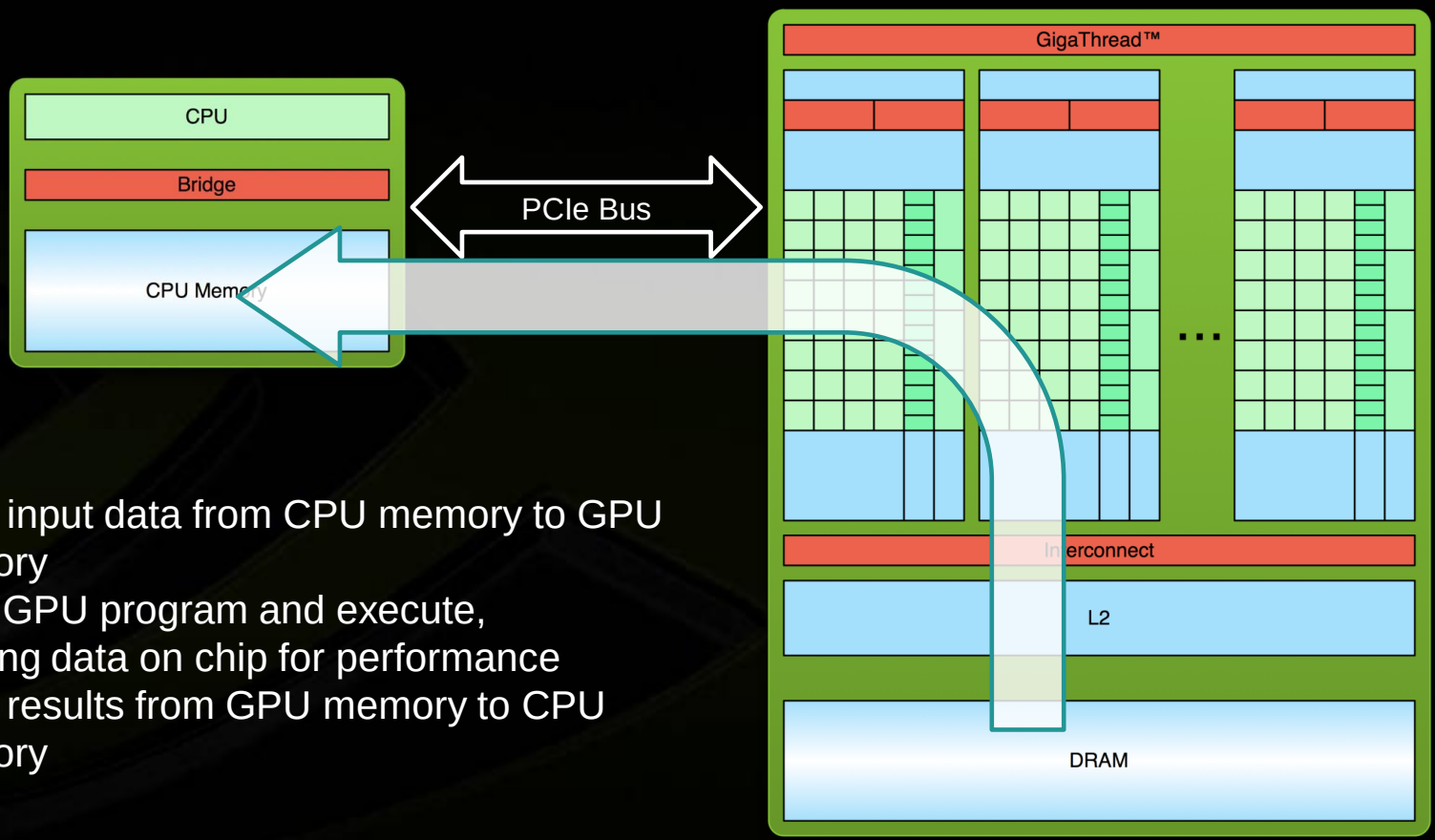
1. Copy input data from CPU memory to GPU memory

Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Processing Flow

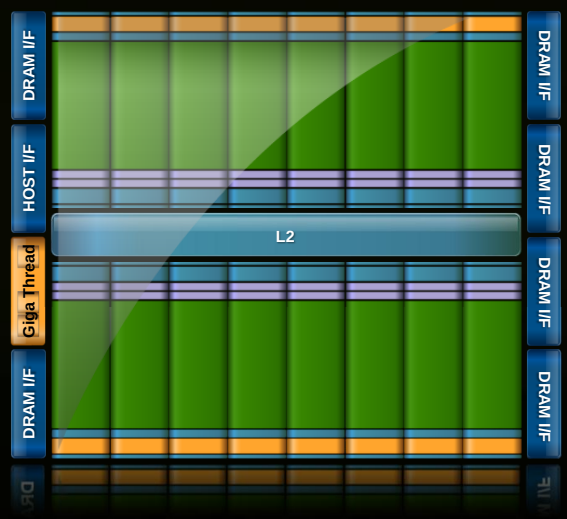


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory


GPU ARCHITECTURE

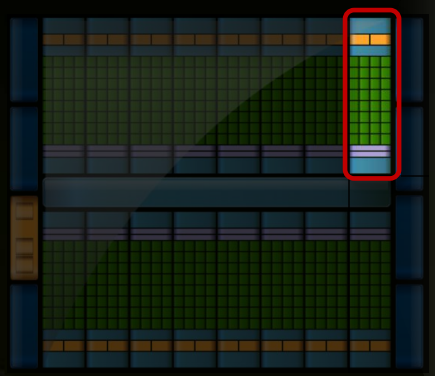
GPU Architecture: Two Main Components

- **Global memory**
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 6 GB
  - Bandwidth currently up to 150 GB/s for Quadro and Tesla products
  - ECC on/off option for Quadro and Tesla products
- **Streaming Multiprocessors (SMs)**
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches



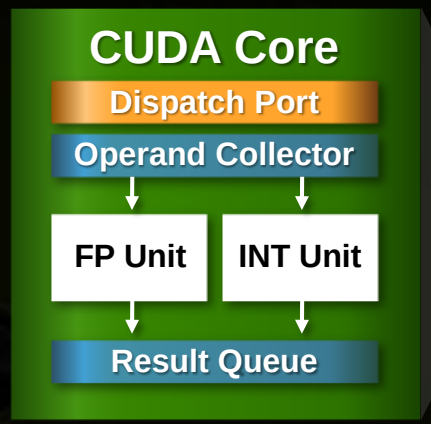
GPU Architecture – Fermi: Streaming Multiprocessor (SM)

- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers



GPU Architecture – Fermi: CUDA Core

- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit



GPU Architecture – Fermi: Memory System

- **L1**
  - 16 or 48KB / SM, can be chosen by the program
  - Hardware-managed
  - Aggregate bandwidth per GPU: 1.03 TB/s
- **Shared memory**
  - User-managed scratch-pad
    - Hardware will not evict until threads overwrite
  - 16 or 48KB / SM (64KB total is split between Shared and L1)
  - Aggregate bandwidth per GPU: 1.03 TB/s

GPU Architecture – Fermi: Memory System

- **ECC protection:**
  - **DRAM**
    - ECC supported for GDDR5 memory
  - **All major internal memories are ECC protected**
    - Register file, L1 cache, L2 cache

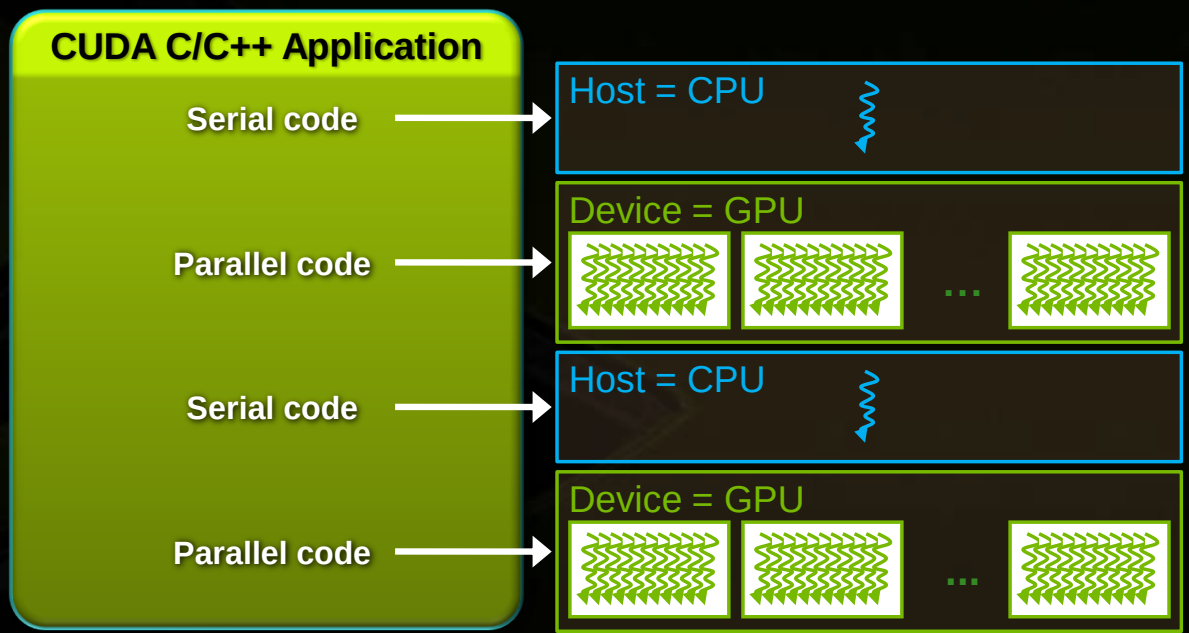
Overview of Tesla C2050/C2070 GPU

C2050 Specifications	
Processor clock	1.15 GHz
# of CUDA cores	448
Peak floating-point perf	1.03 Tflops (SP)
Memory clock	1.5 GHz
Memory bus width	384 bit
Memory size	3 GB / 6 GB

CUDA PROGRAMMING MODEL

Anatomy of a CUDA C/C++ Application

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements



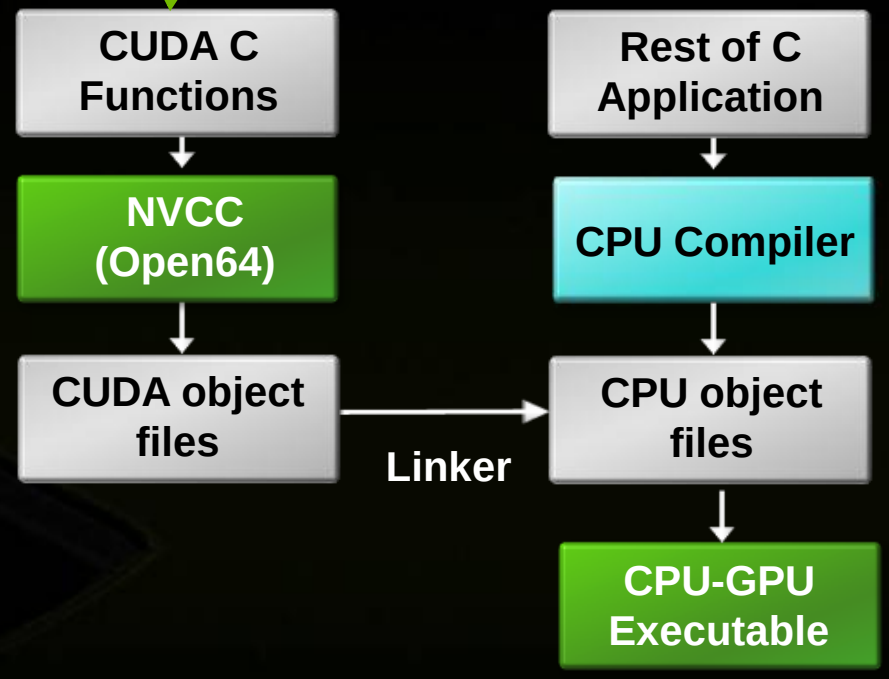
Compiling CUDA C Applications

```
void serial_function(… ) {
  ...
}
void other_function(int ... ) {
  ...
}
```

```
void saxpy_serial(float ... ) {
   for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```

```
void main( ) {
  float x;
  saxpy_serial(..);
  ...
}
```

Modify into Parallel CUDA C code



CUDA C : C with a few keywords

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

CUDA C : C with a few keywords

- **Kernel: function called by the host that executes on the GPU**
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
- **Functions must be declared with a qualifier:**
  - __global__ : GPU kernel function launched by CPU, must return void
  - __device__ : can be called from GPU functions
  - __host__ : can be called from CPU functions (default)
  - __host__ and __device__ qualifiers can be combined

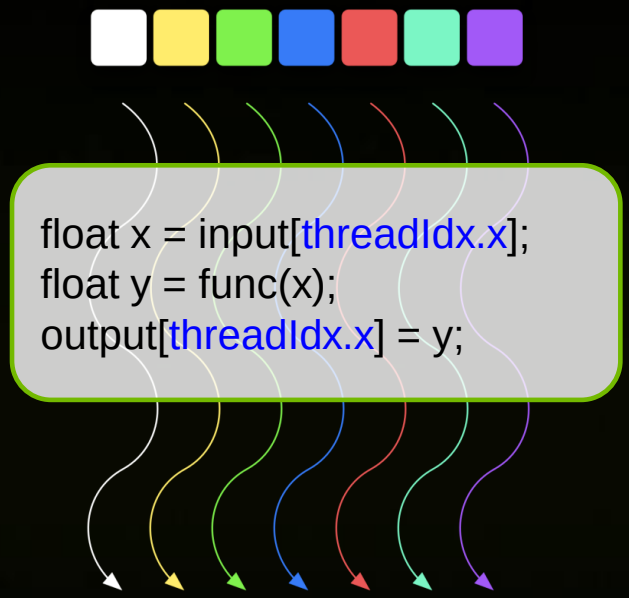
CUDA Kernels

- **Parallel portion of application: execute as a kernel**
  - **Entire GPU executes kernel, many threads**
- **CUDA threads:**
  - **Lightweight**
  - **Fast switching**
  - **1000s execute simultaneously**

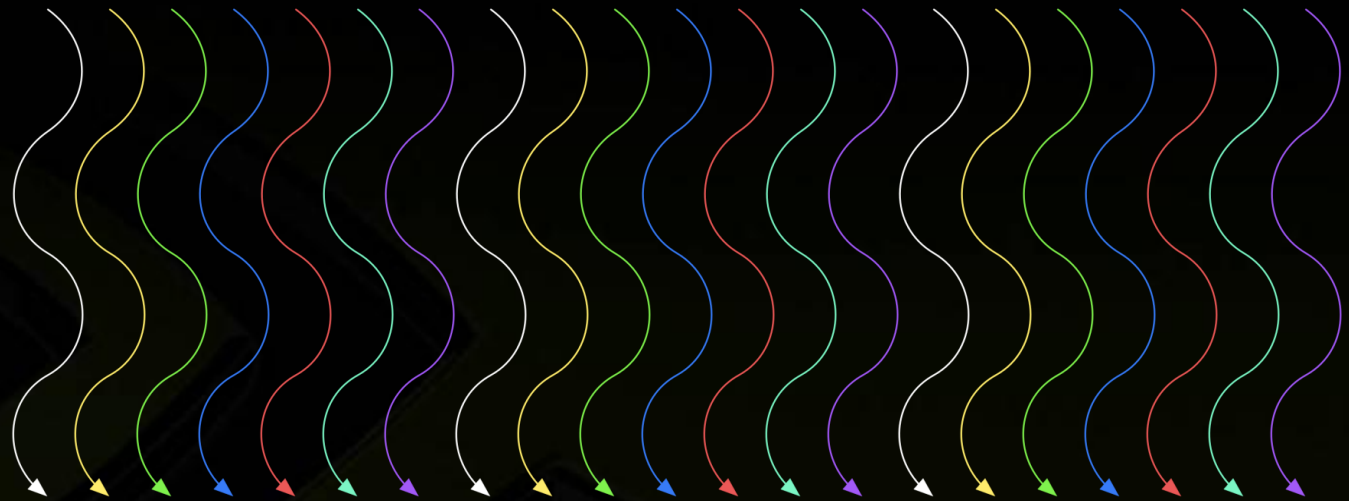
CPU	Host	Executes functions
GPU	Device	Executes kernels

CUDA Kernels: Parallel Threads

- A **kernel** is a function executed on the GPU as an array of threads in parallel
- All threads execute the same code, can take different paths
- Each thread has an ID
  - Select input/output data
  - Control decisions



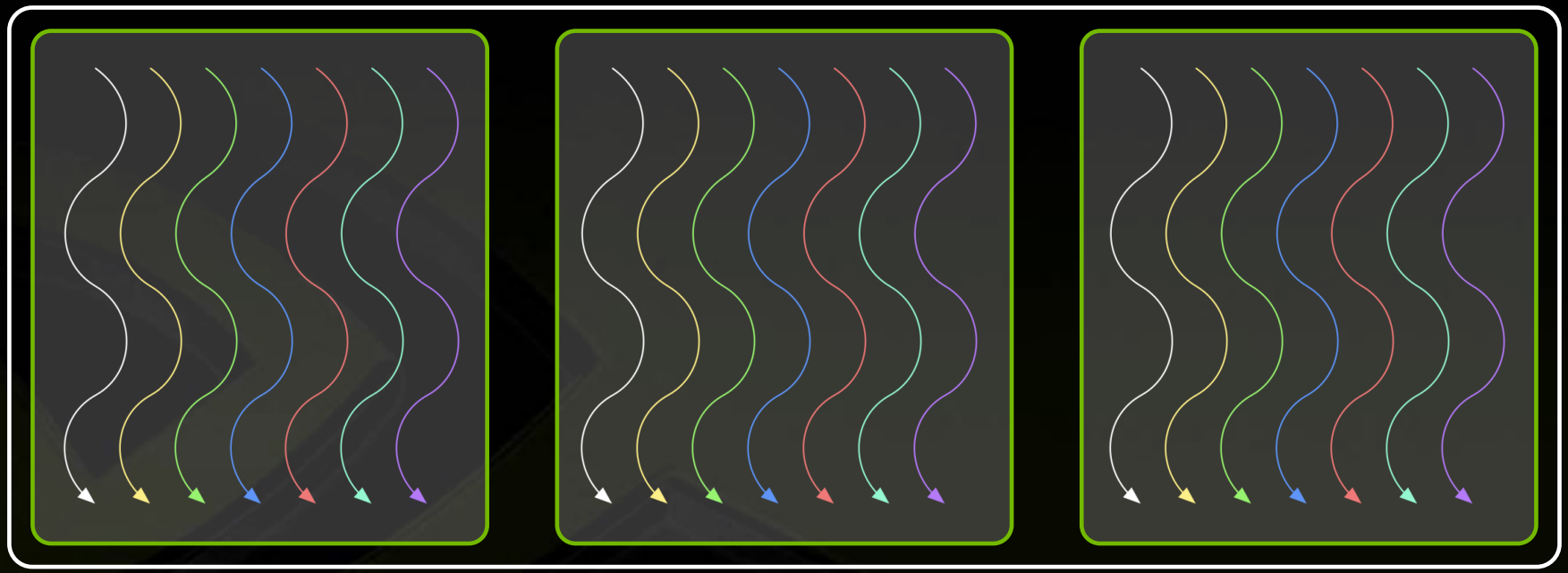
CUDA Kernels: Subdivide into Blocks



CUDA Kernels: Subdivide into Blocks

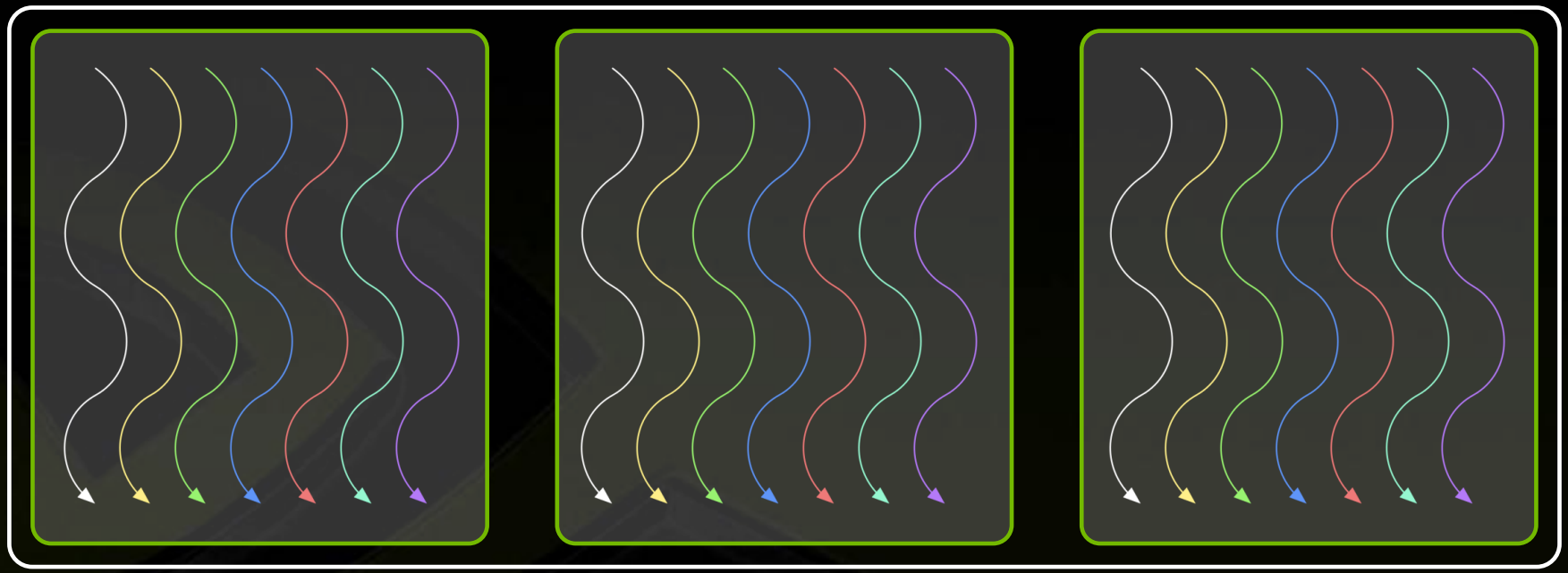
- Threads are grouped into **blocks**

CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid

CUDA Kernels: Subdivide into Blocks

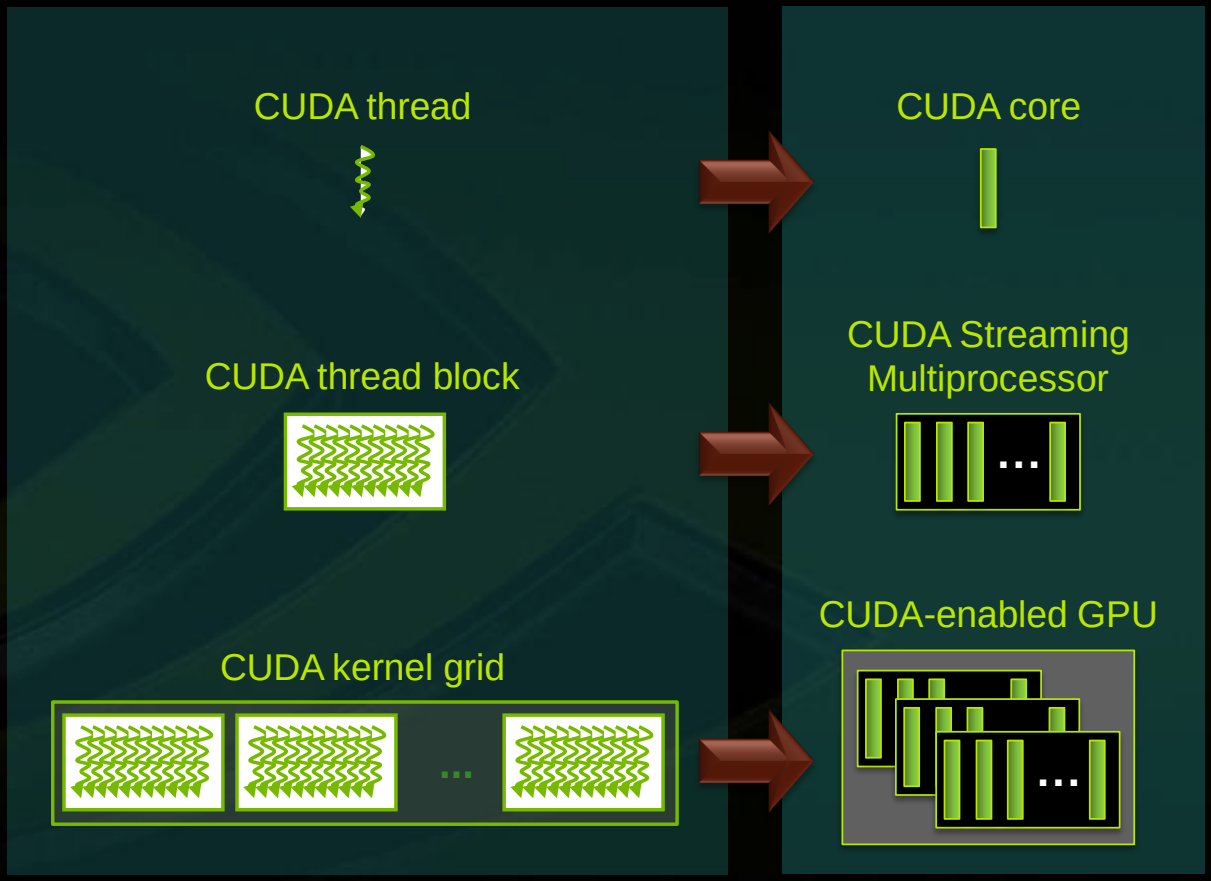


- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Kernel Execution



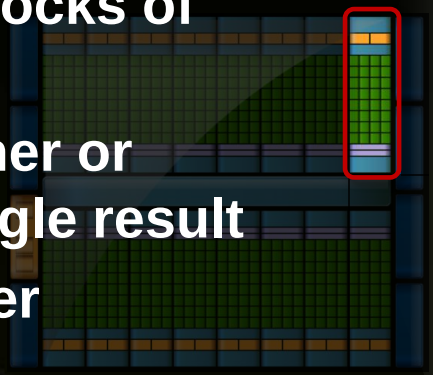
- Each thread is executed by a core

- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

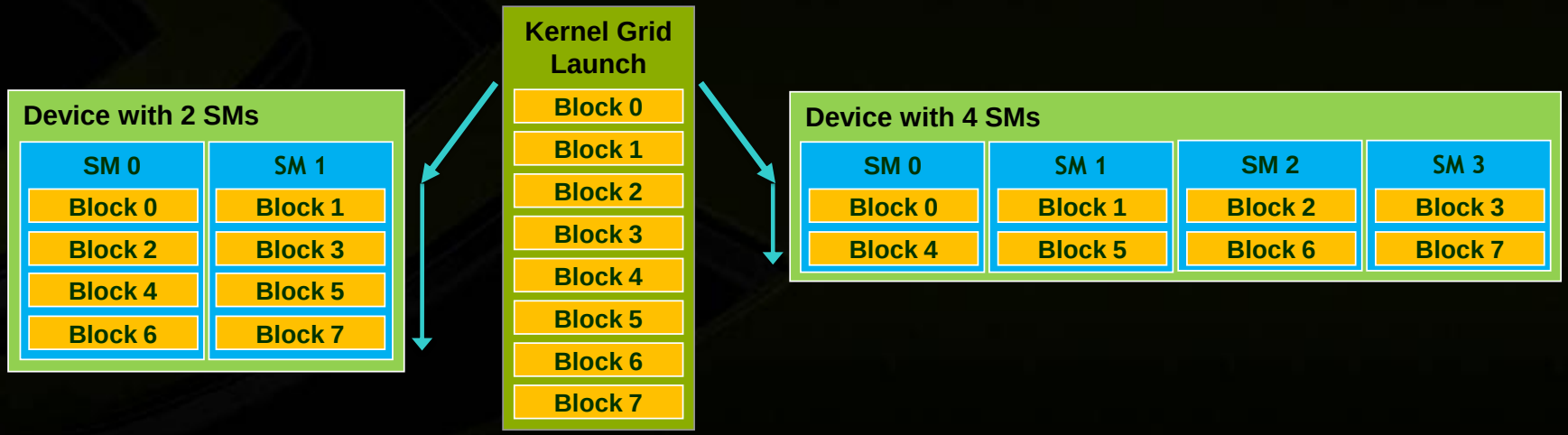
Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory all will use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other



Thread blocks allow scalability

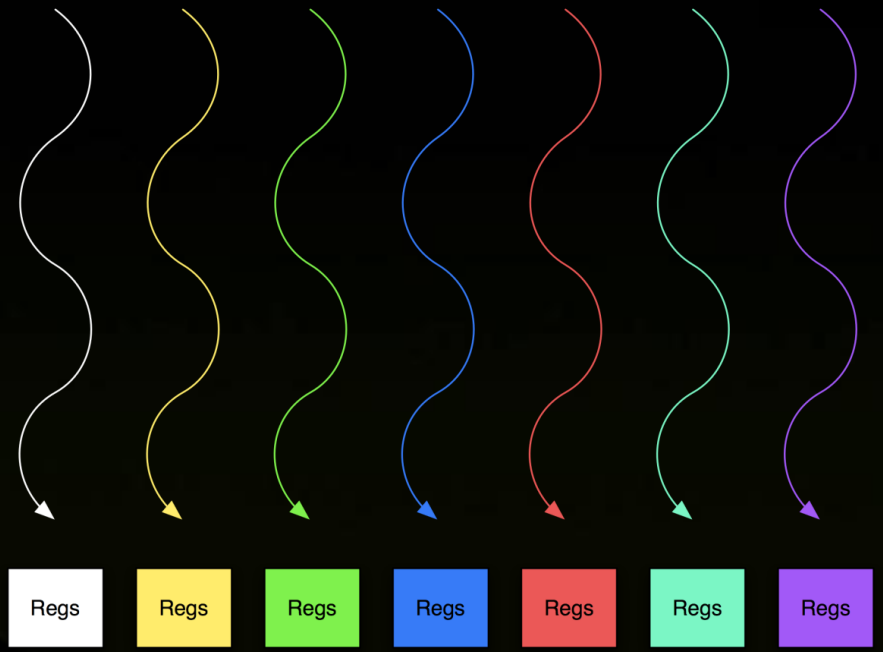
- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs




NVIDIA
CUDA MEMORY SYSTEM

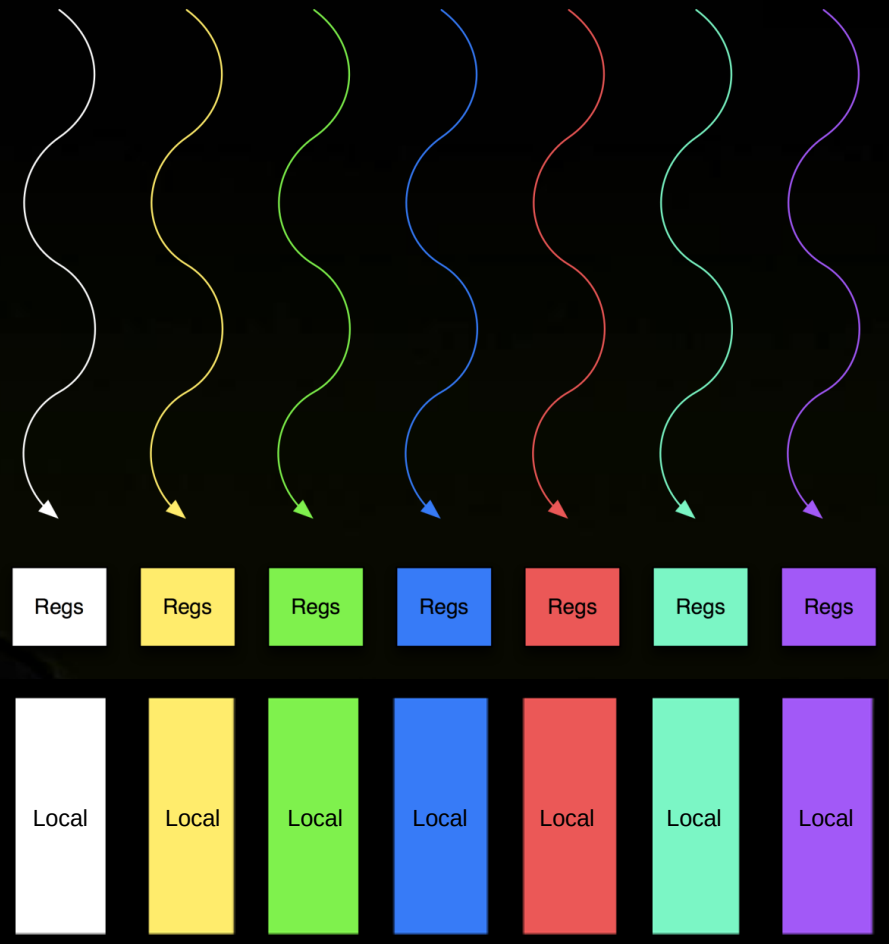
Memory hierarchy

- **Thread:**
  - **Registers**



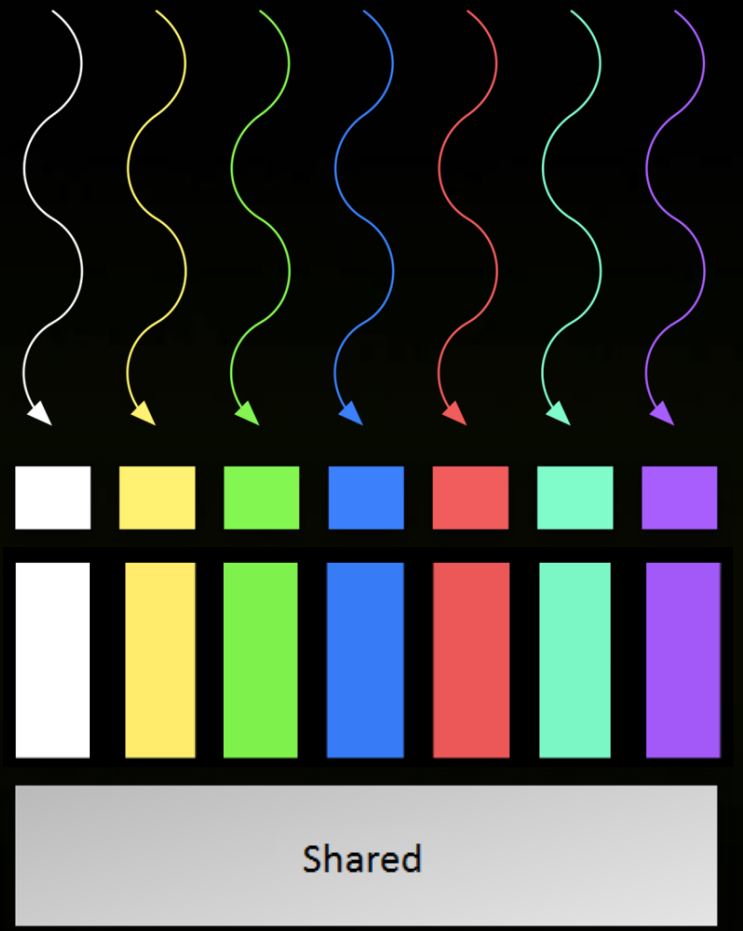
Memory hierarchy

- **Thread:**
  - **Registers**
  - **Local memory**



Memory hierarchy

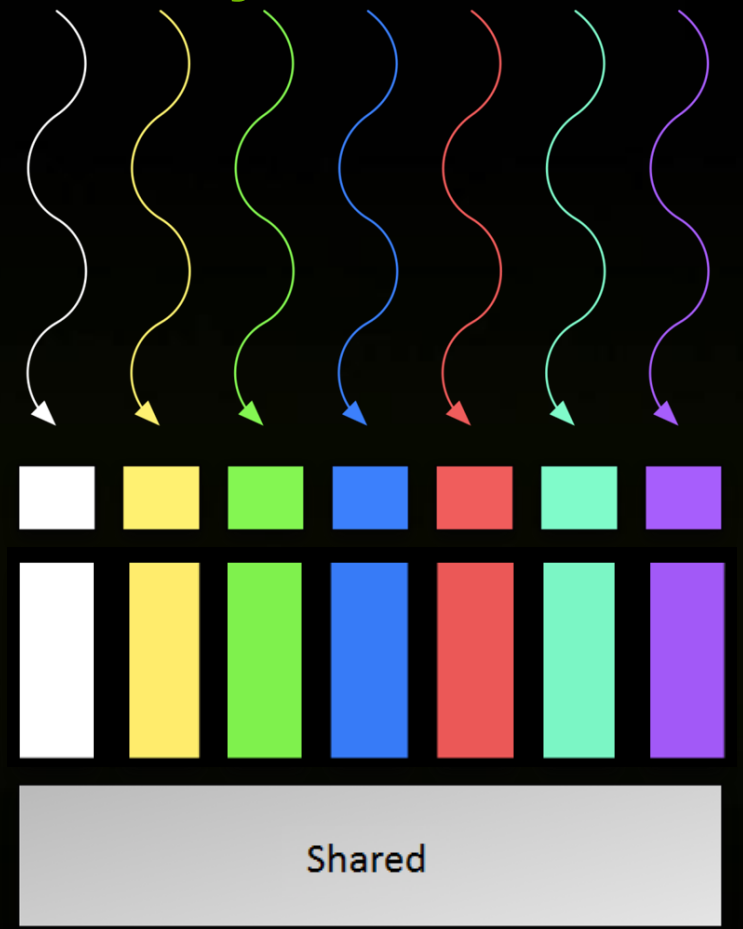
- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory



Memory hierarchy : Shared memory

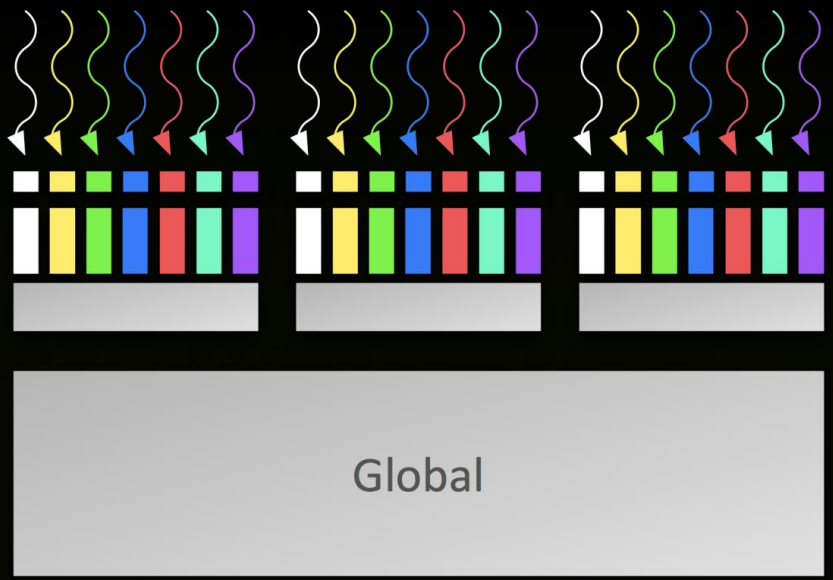
__shared__ int a[SIZE];

- Allocated per thread block, same lifetime as the block
- Accessible by any thread in the block
- Latency: a few cycles
- High aggregate bandwidth:
  - 14 * 32 * 4 B * 1.15 GHz / 2 = 1.03 TB/s
- Several uses:
  - Sharing data among threads in a block
  - User-managed cache (reducing gmem accesses)



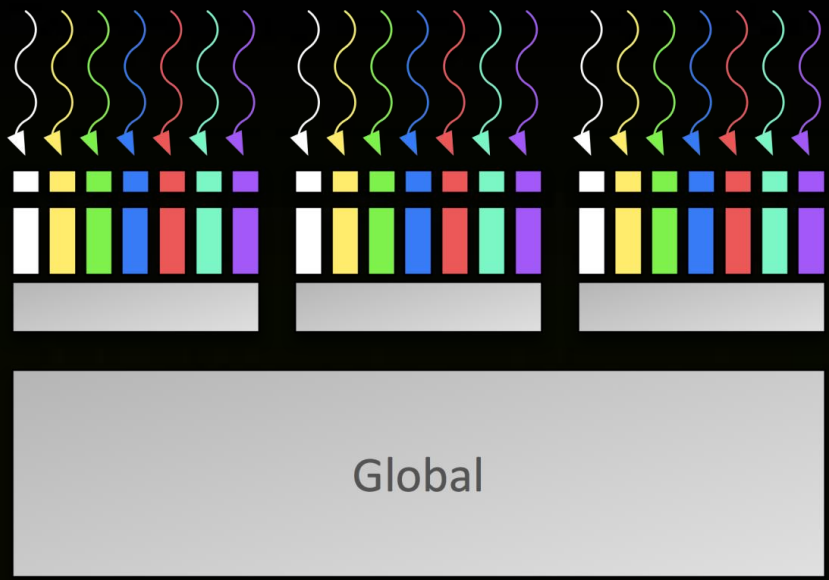
Memory hierarchy

- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory



Memory hierarchy : Global memory

- Accessible by all threads of any kernel
- Data lifetime: from allocation to deallocation by host code
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)
- Latency: 400-800 cycles
- Bandwidth: 156 GB/s
  - Note: requirement on access pattern to reach peak performance



Kight's Landing (KNL) - Xeon Phi

Avinash Sodani, Intel

Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor

Avinash Sodani
KNL Chief Architect
Senior Principal Engineer, Intel Corp.

Legal

Knights Landing: Next Intel® Xeon Phi™ Processor

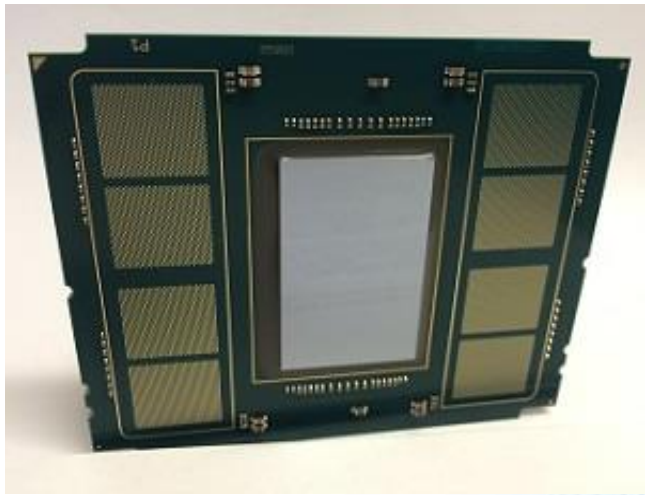
Intel® Many-Core Processor targeted for HPC and Supercomputing

First **self-boot** Intel® Xeon Phi™ processor that is **binary compatible** with main line IA. Boots standard OS.

Significant improvement in scalar and **vector** performance

Integration of **Memory on package**: innovative memory architecture for high bandwidth and high capacity

Integration of **Fabric on package**

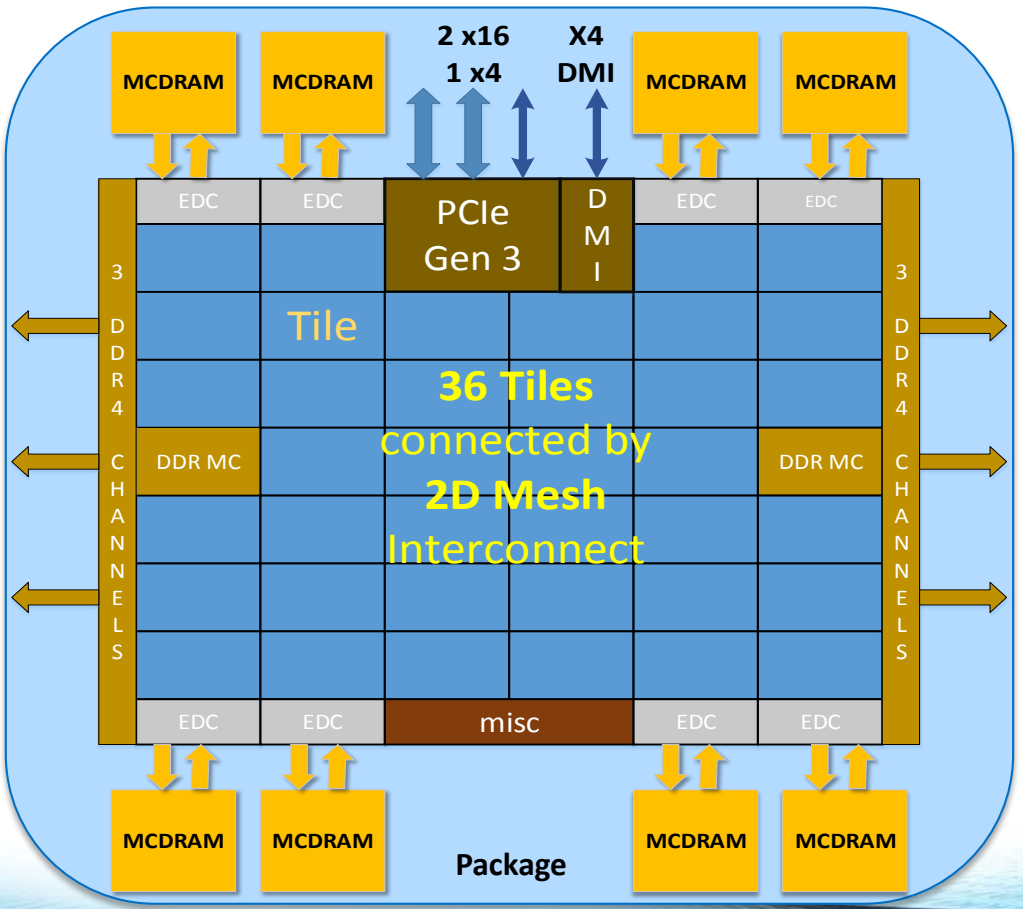


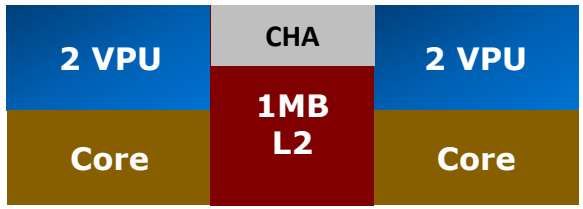
Three products

KNL Self-Boot	KNL Self-Boot w/ Fabric	KNL Card
(Baseline)	(Fabric Integrated)	(PCIe-Card)

Potential future options subject to change without notice.
All timeframes, features, products and dates are preliminary forecasts and subject to change without further notification.

Knights Landing Overview





Chip: 36 Tiles interconnected by **2D Mesh**

Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW

DDR4: 6 channels @ 2400 up to 384GB

IO: 36 lanes PCIe Gen3. 4 lanes of DMI for chipset

Node: 1-Socket only

Fabric: Omni-Path on-package (not shown)

Vector Peak Perf: 3+TF DP and 6+TF SP Flops

Scalar Perf: ~3x over Knights Corner

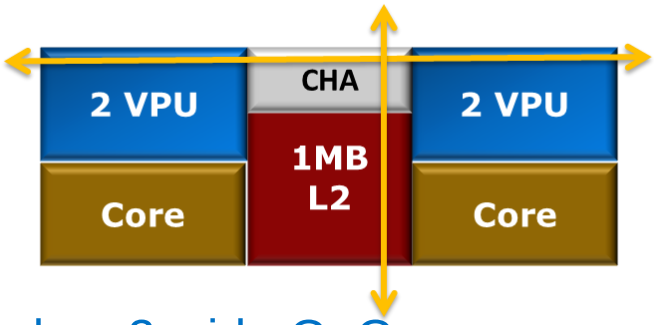
Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). 2Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

KNL Tile:

2 Cores, each with 2 VPU

1M L2 shared between two Cores



Core: Changed from Knights Corner (KNC) to KNL. Based on 2-wide OoO Silvermont™ Microarchitecture, but with *many* changes for HPC.

4 thread/core. Deeper OoO. Better RAS. Higher bandwidth. Larger TLBs.

2 VPU: 2x AVX512 units. 32SP/16DP per unit. X87, SSE, AVX1, AVX2 and EMU

L2: 1MB 16-way. 1 Line Read and ½ Line Write per cycle. Coherent across all Tiles

CHA: **C**aching/**H**ome **A**gent. Distributed Tag Directory to keep L2s coherent. MESIF protocol. 2D-Mesh connections for Tile

Many Trailblazing Improvements in KNL

Improvements	What/Why
Self Boot Processor	No PCIe bottleneck
Binary Compatibility with Xeon	Runs all legacy software. No recompilation.
New Core: Atom™ based	~3x higher ST performance over KNC
Improved Vector density	3+ TFLOPS (DP) peak per chip
New AVX 512 ISA	New 512-bit Vector ISA with Masks
Scatter/Gather Engine	Hardware support for gather and scatter
New memory technology: MCDRAM + DDR	Large High Bandwidth Memory → MCDRAM Huge bulk memory → DDR
New on-die interconnect: Mesh	High BW connection between cores and memory
Integrated Fabric: Omni-Path	Better scalability to large systems. Lower Cost

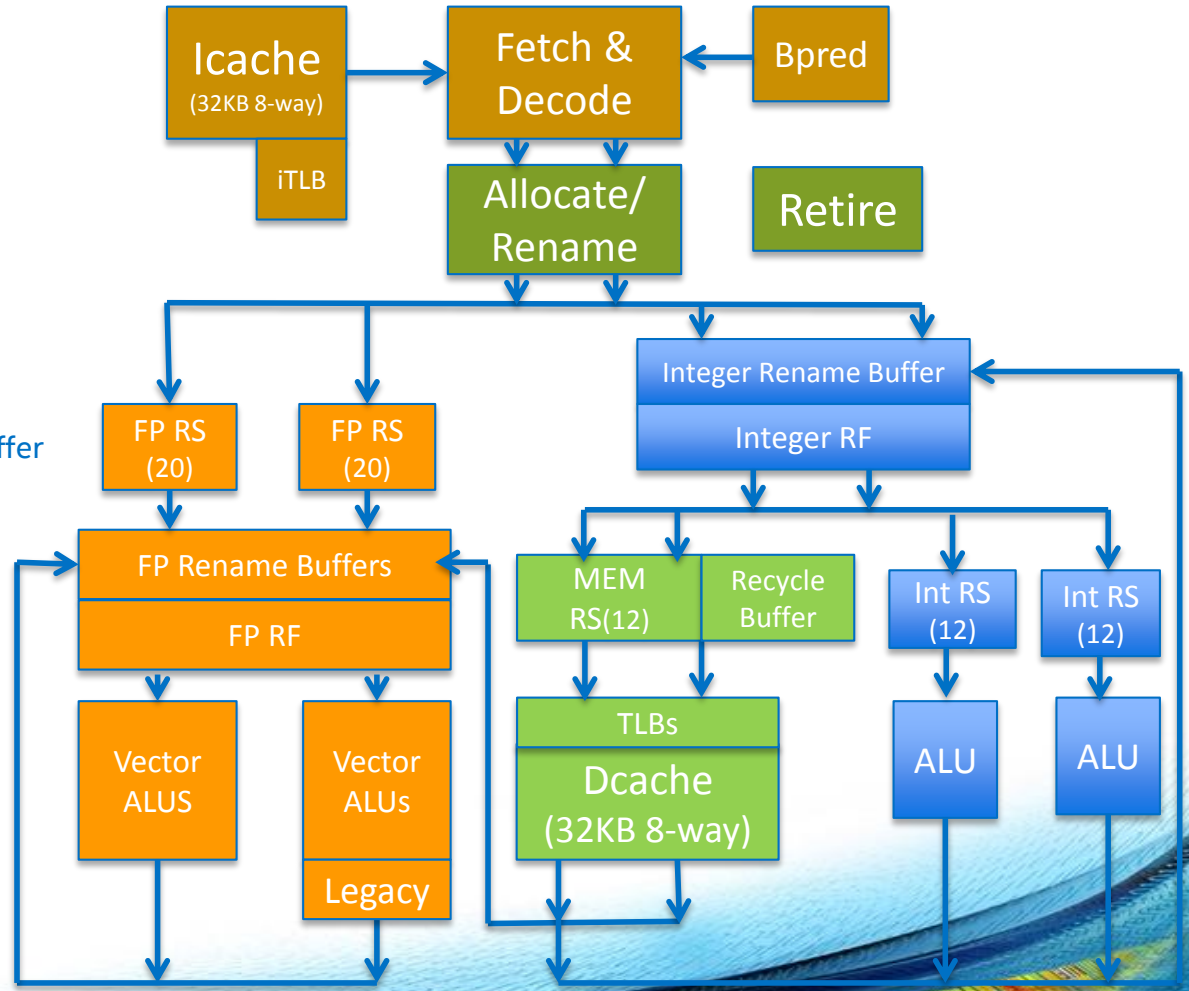
Core & VPU

- Out-of-order core w/ 4 SMT threads
- VPU tightly integrated with core pipeline

- 2-wide Decode/Rename/Retire
- ROB-based renaming. 72-entry ROB & Rename Buffers
- Up to 6-wide at execution
- Int and FP RS OoO.
- MEM RS inorder with OoO completion. Recycle Buffer holds memory ops waiting for completion.
- Int and Mem RS hold source data. FP RS does not.

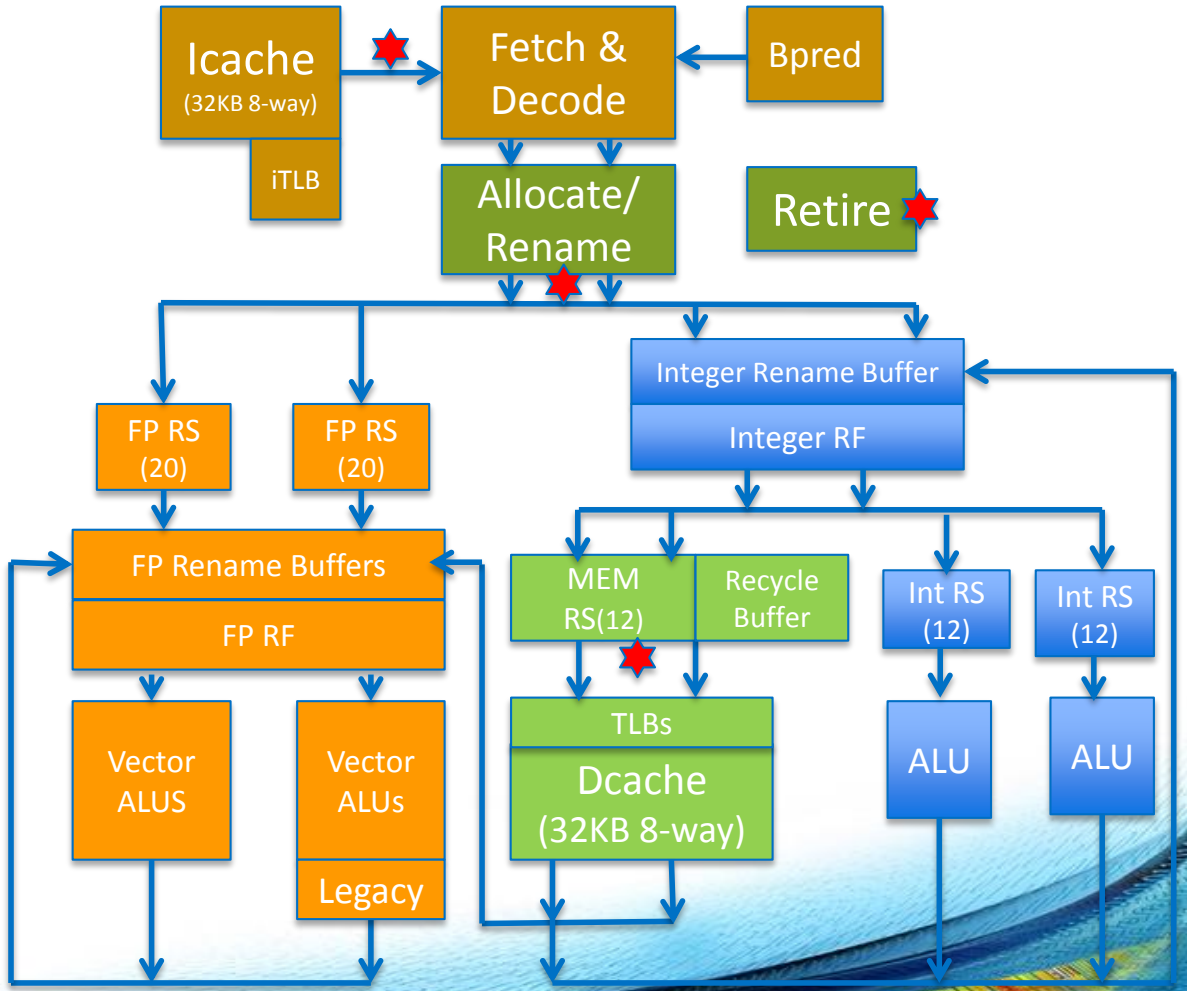
- 2x 64B Load & 1 64B Store ports in Dcache.
- 1st level uTLB: 64 entries
- 2nd level dTLB: 256 4K, 128 2M, 16 1G pages

- L1 Prefetcher (IPP) and L2 Prefetcher.
- 46/48 PA/VA bits
- Fast unaligned and cache-line split support.
- Fast Gather/Scatter support

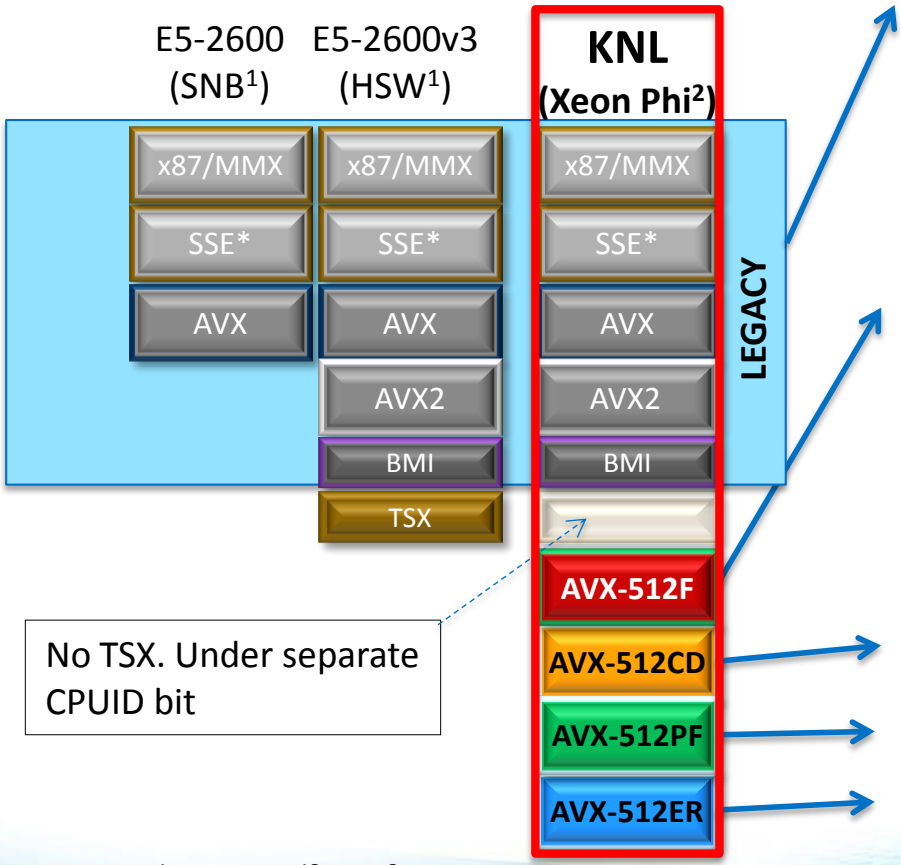


Threading

- 4 Threads per core. Simultaneous Multithreading.
- Core resources **shared** or **dynamically repartitioned** between active threads
  - ROB, Rename Buffers, RS: Dynamically partitioned
  - Caches, TLBs: Shared
  - E.g., 1 thread active → uses full resources of the core
- Several Thread Selection points in the pipeline. (✸)
  - Maximize throughput while being fair.
  - Account for available resources, stalls and forward progress



KNL ISA



No TSX. Under separate CPUID bit

KNL implements all legacy instructions

- Legacy binary runs w/o recompilation
- KNC binary requires recompilation

KNL introduces AVX-512 Extensions

- 512-bit FP/Integer Vectors
- 32 registers, & 8 mask registers
- Gather/Scatter

Conflict Detection: Improves Vectorization

Prefetch: Gather and Scatter Prefetch

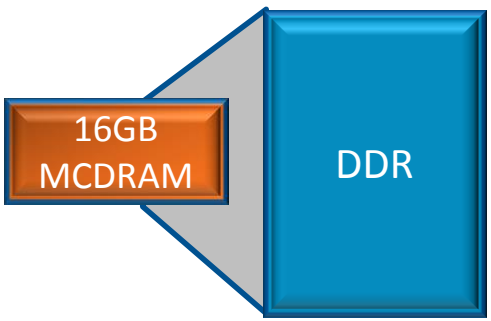
Exponential and Reciprocal Instructions

1. Previous Code name Intel® Xeon® processors
2. Xeon Phi = Intel® Xeon Phi™ processor

Memory Modes

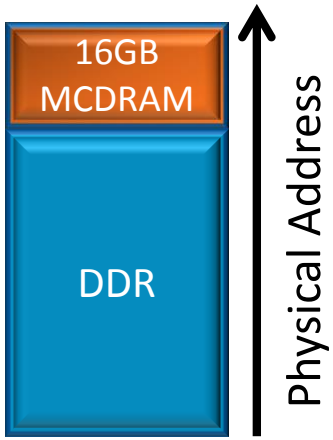
Three Modes. Selected at boot

Cache Mode



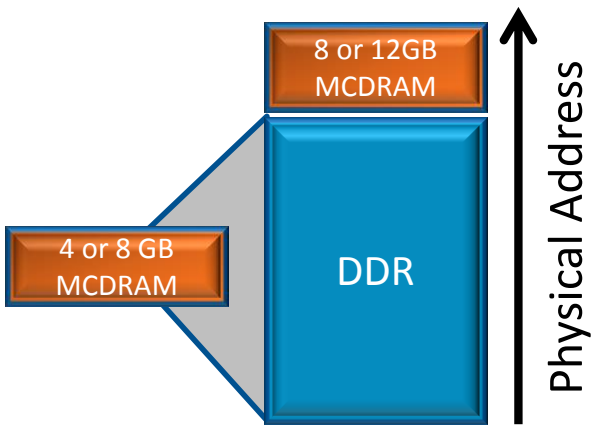
- SW-Transparent, Mem-side cache
- Direct mapped. 64B lines.
- Tags part of line
- Covers whole DDR range

Flat Mode



- MCDRAM as regular memory
- SW-Managed
- Same address space

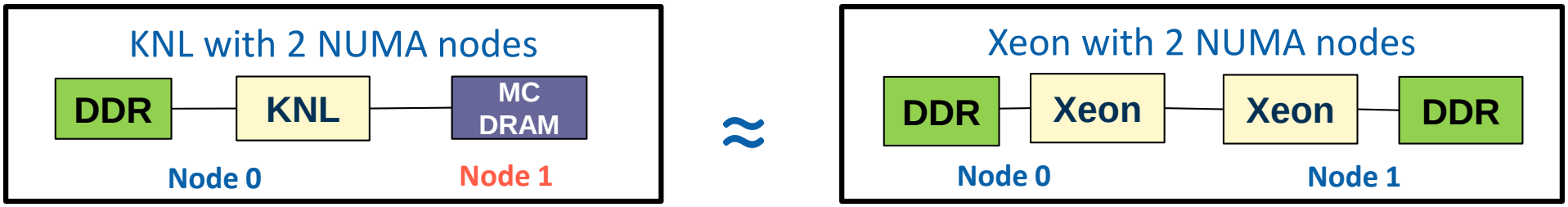
Hybrid Mode



- Part cache, Part memory
- 25% or 50% cache
- Benefits of both

Flat MCDRAM: SW Architecture

MCDRAM exposed as a separate NUMA node



Memory allocated in DDR by default → Keeps non-critical data out of MCDRAM.

Apps explicitly allocate critical data in MCDRAM. Using two methods:

- **"Fast Malloc"** functions in High BW library (https://github.com/memkind)
  - Built on top to existing *libnuma* API
- **"FASTMEM"** Compiler Annotation for Intel Fortran

Flat MCDRAM with existing NUMA support in Legacy OS

Flat MCDRAM SW Usage: Code Snippets

C/C++ (*https://github.com/memkind)

Allocate into DDR

```
float   *fv;
fv = (float *)malloc(sizeof(float)*100);
```

Allocate into MCDRAM

```
float   *fv;
fv = (float *)hbw_malloc(sizeof(float) * 100);
```

Intel Fortran

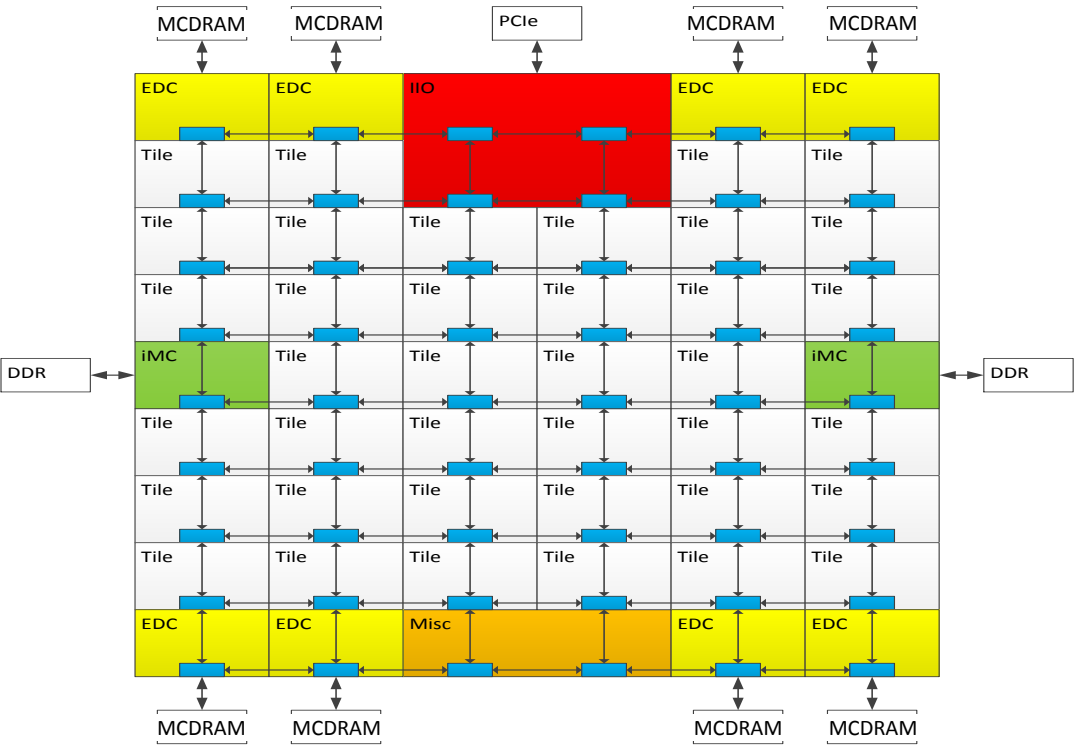
Allocate into MCDRAM

```
c      Declare arrays to be dynamic
       REAL, ALLOCATABLE :: A(:)

!DEC$ ATTRIBUTES, FASTMEM :: A

       NSIZE=1024
c      allocate array 'A' from MCDRAM
c
       ALLOCATE (A(1:NSIZE))
```

KNL Mesh Interconnect



Mesh of Rings

- Every row and column is a (half) ring
- YX routing: Go in Y → Turn → Go in X
- Messages arbitrate at injection and on turn

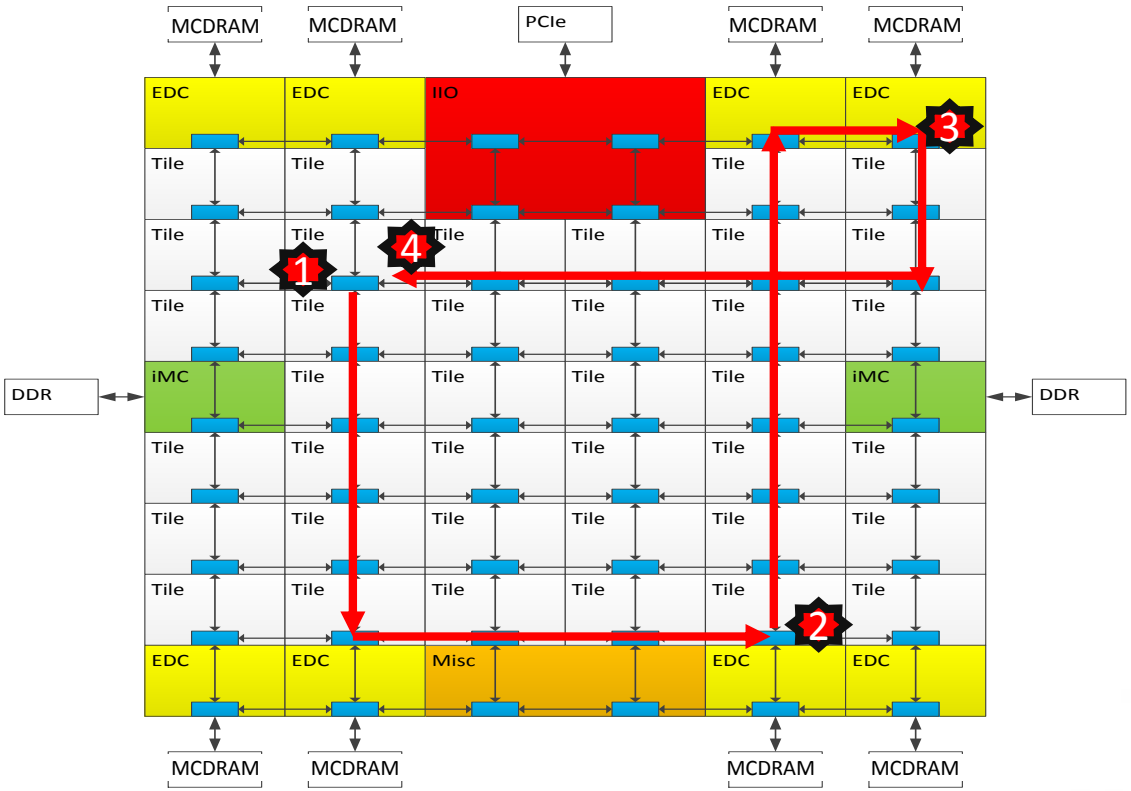
Cache Coherent Interconnect

- MESIF protocol (F = Forward)
- Distributed directory to filter snoops

Three Cluster Modes

(1) All-to-All (2) Quadrant (3) Sub-NUMA Clustering

Cluster Mode: All-to-All



Address uniformly hashed across all distributed directories

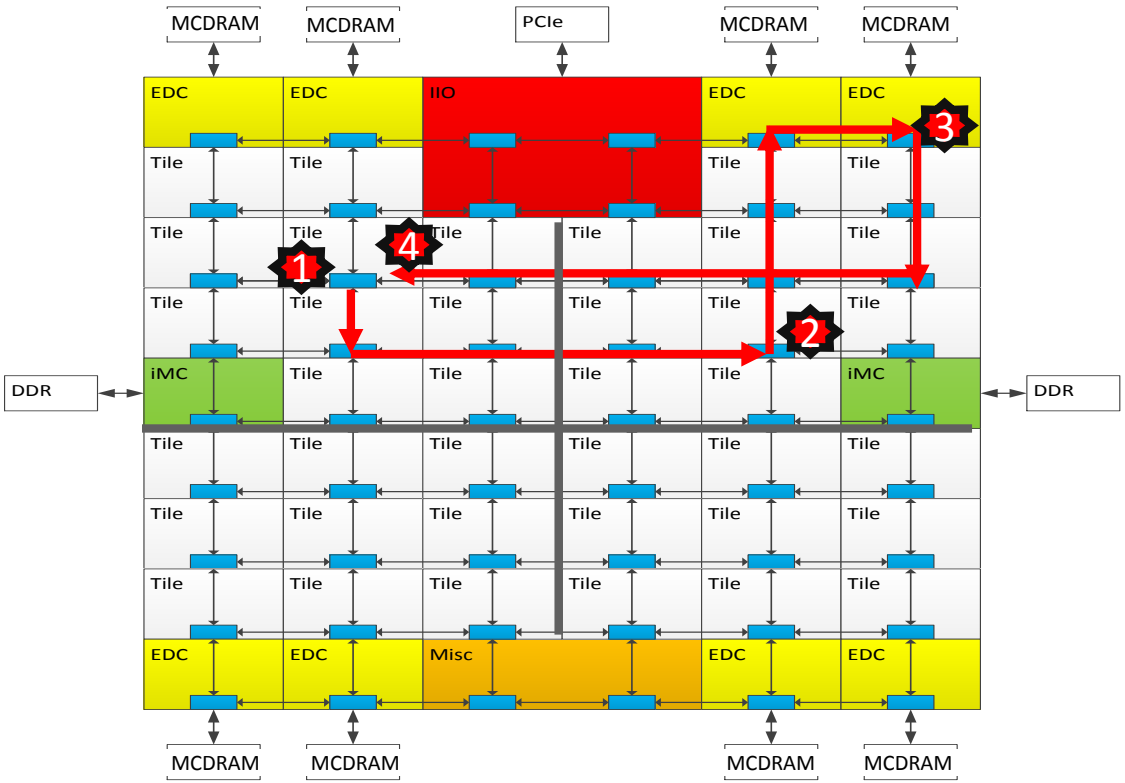
No affinity between Tile, Directory and Memory

Most general mode. Lower performance than other modes.

Typical Read L2 miss

1. L2 miss encountered
2. Send request to the distributed directory
3. Miss in the directory. Forward to memory
4. Memory sends the data to the requestor

Cluster Mode: Quadrant



Chip divided into four virtual Quadrants

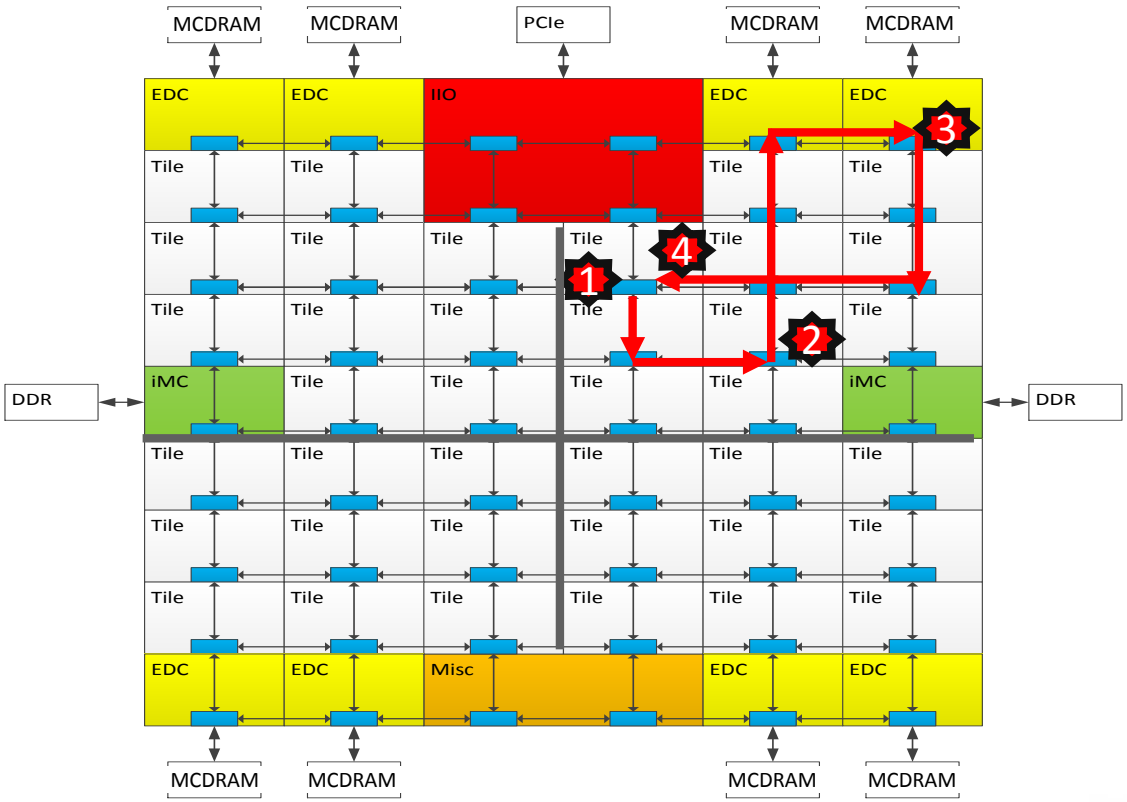
Address hashed to a Directory in the same quadrant as the Memory

Affinity between the Directory and Memory

Lower latency and higher BW than all-to-all. SW Transparent.

1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

Cluster Mode: Sub-NUMA Clustering (SNC)



Each Quadrant (Cluster) exposed as a separate NUMA domain to OS.

Looks analogous to 4-Socket Xeon

Affinity between Tile, Directory and Memory

Local communication. Lowest latency of all modes.

SW needs to NUMA optimize to get benefit.

1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

KNL with Omni-Path™

Omni-Path™ Fabric integrated *on package*

First product with integrated fabric

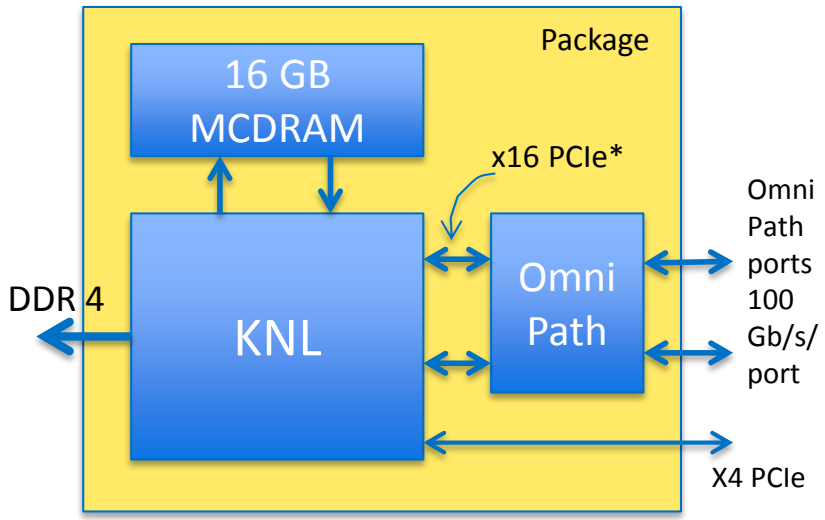
Connected to KNL die via 2 x16 PCIe* ports

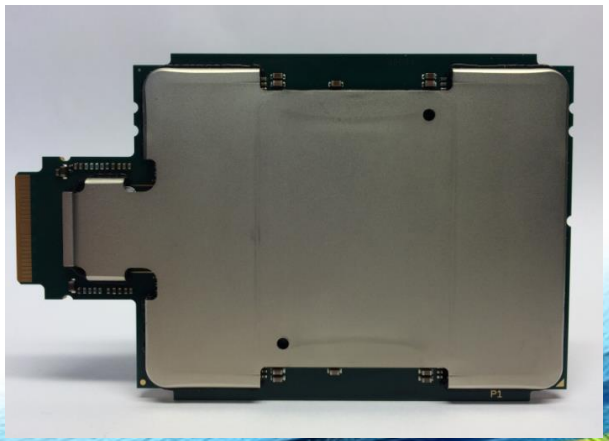
Output: 2 Omni-Path ports

- 25 GB/s/port (bi-dir)

Benefits

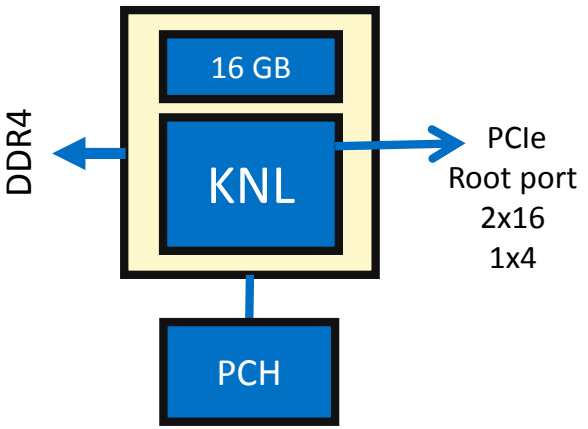
- Lower cost, latency and power
- Higher density and bandwidth
- Higher scalability





*On package connect with PCIe semantics, with MCP optimizations for physical layer

Knights Landing Products

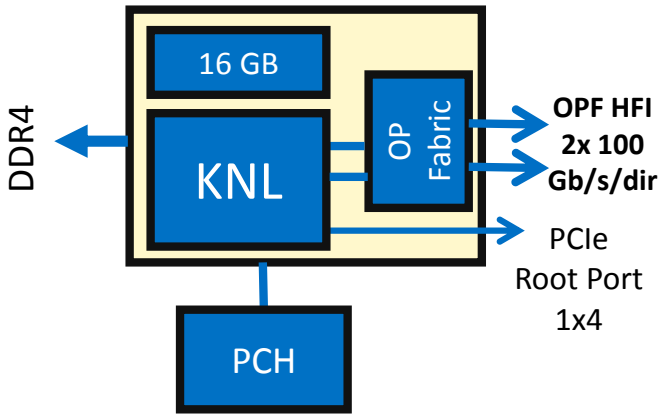


KNL

DDR Channels: 6

MCDRAM: up to 16 GB

Gen3 PCIe (Root port): 36 lanes



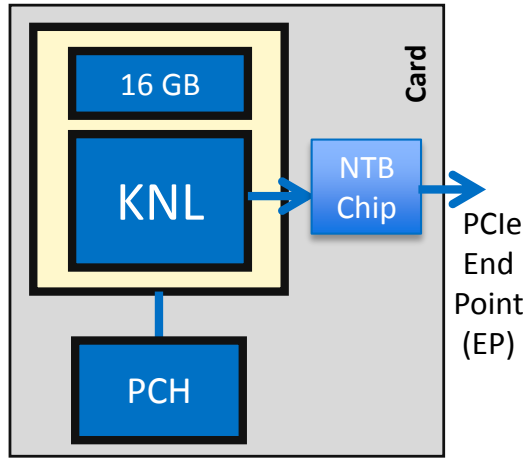
KNL with Omni-Path

DDR Channels: 6

MCDRAM: up to 16 GB

Gen3 PCIe (Root port): 4 lanes

Omni-Path Fabric: 200 Gb/s/dir



KNL Card

No DDR Channels

MCDRAM: up to 16 GB

Gen3 PCIe (End point): 16 lanes

NTB Chip to create PCIe EP

Self Boot Socket

PCIe Card

High Performance Graph Analytics on Manycore Processors

George M. Slota[1,2] Sivasankaran Rajamanickam[2]
Kamesh Madduri[1]

[1]Penn State University, [2]Sandia National Laboratories
gslota@psu.edu, madduri@cse.psu.edu, srajama@sandia.gov

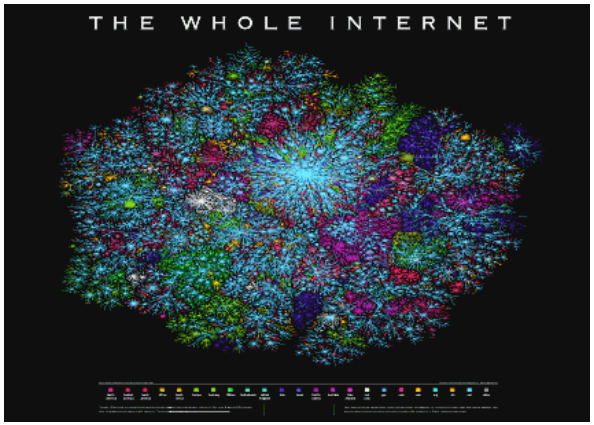
IPDPS 2015, Hyderabad, India
May 26, 2015

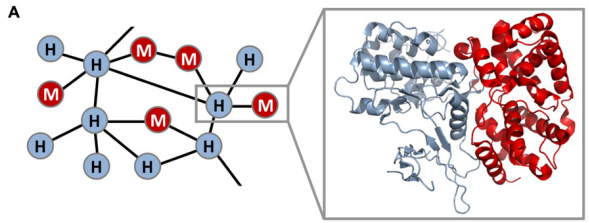
Graphs are...

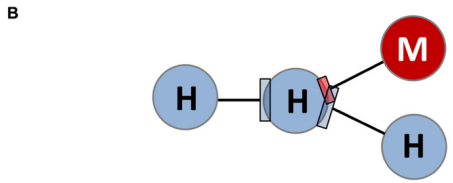
- **Everywhere**

Graphs are...

- **Everywhere**
  - Internet
  - Social, communication networks
  - Computational biology and chemistry
  - Scientific computing, meshing, interactions







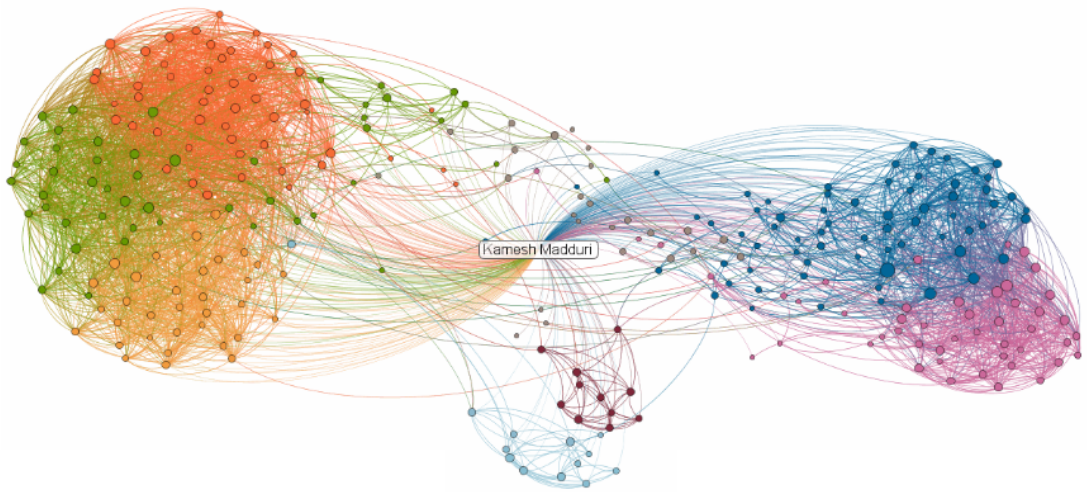


Figure sources: Franzosa et al. 2012, http://www.unc.edu/ unclng/Internet_History.htm

Graphs are...

- **Complex**

Graphs are...

- **Complex**
  - Graph analytics is listed as one of DARPA's 23 toughest mathematical challenges
  - Highly diverse – graph structure and problems vary from application to application
  - Real-world graph characteristics makes computational analysis challenging

Graphs are...

- **Complex**
  - Graph analytics is listed as one of DARPA's 23 toughest mathematical challenges
  - Highly diverse – graph structure and problems vary from application to application
  - Real-world graph characteristics makes computational analysis challenging
    - Skewed degree distributions
    - 'Small-world' nature
    - Dynamic

Accelerators (GPUs, Xeon Phi) are also ...

- **Everywhere**
  - Most of the top supercomputers and academic clusters use GPUs and Intel Xeon Phi co-processors
  - Manycore processors might replace multicore in future

Accelerators (GPUs, Xeon Phi) are also ...

- **Everywhere**
  - Most of the top supercomputers and academic clusters use GPUs and Intel Xeon Phi co-processors
  - Manycore processors might replace multicore in future
- **Complex**
  - Multilevel memory, processing hierarchy
  - Explicit communication and data handling
  - Require programming for wide parallelism

Figure sources: NCSA, NVIDIA

Motivating questions for this work

- **Q**: What are some **common abstractions** that we can use to develop parallel graph algorithms for manycores?
- **Q**: What **key optimization strategies** can we identify to design new parallel graph algorithms for manycores?
- **Q**: Is it possible to develop **performance-portable implementations** of graph algorithms using advanced **libraries and frameworks** using the above optimizations and abstractions?

Our contributions

- **Q**: **Common abstractions** for manycores?
  - We use array-based data structures, express computation in the form of nested loops.
- **Q**: **Key optimization strategies**
  - We improve load balance by manual loop collapse.
- **Q**: **Performance-portable implementations** of graph algorithms using advanced **libraries and frameworks**?
  - We use Kokkos (Edwards et al., JPDC 2014).
- We compare high-level implementations using new framework to hand-optimized code + vary graph computations + vary graph inputs + vary manycore platform.

Talk Overview

- Manycores and the Kokkos programming model
- Abstracting graph algorithms
- Optimizing for manycore processing
- Algorithms
- Results

Background
GPU and Xeon Phi microarchitecture

- **GPU**
  - Multiprocessors (up to about 15/GPU)
  - Multiple groups of stream processors per MP (12×16)
  - *Warps* of threads all execute SIMT on single group of stream processors (32 threads/warp, two cycles per instruction)
  - Irregular computation (high degree verts, if/else, etc.) can result in most threads in warp doing NOOPs
- **Xeon Phi (MIC)**
  - Many simple (Pentium 4) cores, 57-61
  - 4 threads per core, need at least 2 threads/core for OPs on each cycle
  - Highly vectorized (512 bit width) - difficult for irregular computations to exploit

Background
Kokkos and GPU microarchitecture

- Kokkos
  - Developed as back-end for portable scientific computing
  - Polymorphic multi-dimensional arrays for varying access patterns
  - Thread parallel execution for fine-grained parallelism
- Kokkos model - performance portable programming to multi/manycores
  - Thread team - multiple warps on same multiprocessor, but all still SIMT for GPU
  - Thread league - multiple thread teams, over all teams all work is performed
  - Work statically partitioned to teams before parallel code is called

Abstracting graph algorithms
for large sparse graph analysis

- **Observation**: most (synchronous) graph algorithms follow a tri-nested loop structure
  - Optimize for this general algorithmic template
  - Transform structure for more parallelism

```
1:  Initialize temp/result arrays A_t[1..n], 1 ≤ t ≤ l.        ▷ l = O(1)
2:  Initialize S_1[1..n].
3:  for i = 1 to niter do                                      ▷ niter = O(log n)
4:      Initialize S_{i+1}[1..n].                              ▷ Σ_i |S_i| = O(m)
5:      for j = 1 to |S_i| do                                  ▷ |S_i| = O(n)
6:          u ← S_i[j]
7:          Read/update A_t[u], 1 ≤ t ≤ l.
8:          for k = 1 to |E[u]| do                             ▷ |E[u]| = O(n)
9:              v ← E[u][k]
10:             Read/update A_t[v].
11:             Read/update S_{i+1}.
12:         Read/update A_t[u].
```

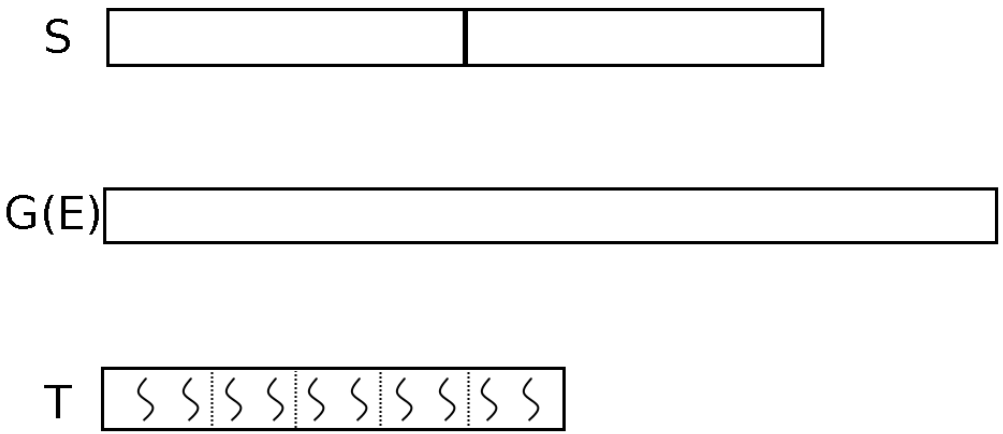
Abstracting graph algorithms
for large sparse graph analysis

- **Observation**: most (synchronous) graph algorithms follow a tri-nested loop structure
  - Optimize for this general algorithmic template
  - Transform structure for more parallelism

```
1:  Initialize temp/result arrays A_t[1..n], 1 ≤ t ≤ l.        ▷ l = O(1)
2:  Initialize S_1[1..n].
3:  for i = 1 to niter do                                      ▷ niter = O(log n)
4:      Initialize S_{i+1}[1..n].                              ▷ Σ_i |S_i| = O(m)
5:      for j = 1 to |S_i| do                                  ▷ |S_i| = O(n)
6:          u ← S_i[j]
7:          Read/update A_t[u], 1 ≤ t ≤ l.
8:          for k = 1 to |E[u]| do                             ▷ |E[u]| = O(n)
9:              v ← E[u][k]
10:             Read/update A_t[v].
11:             Read/update S_{i+1}.
12:         Read/update A_t[u].
```

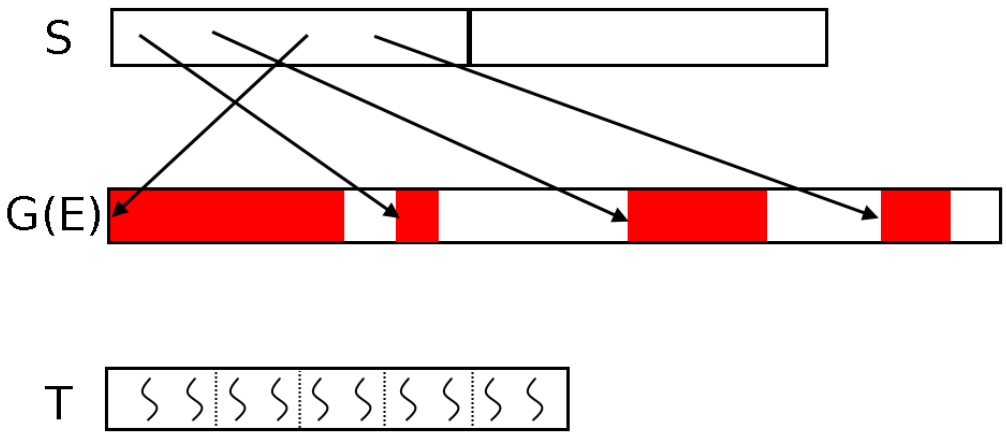
Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization



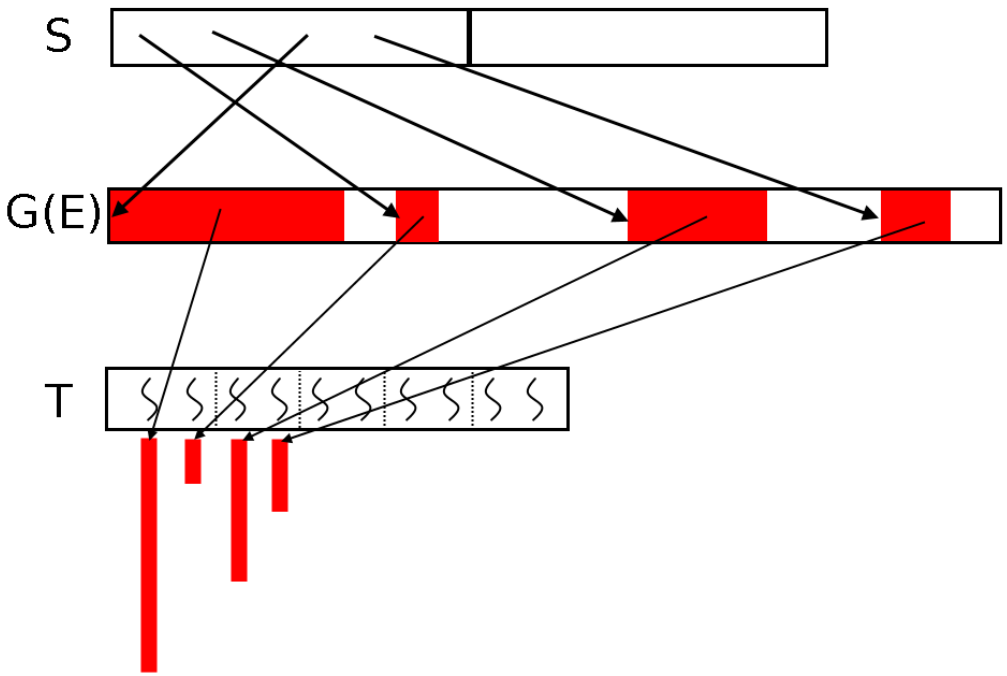
Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization



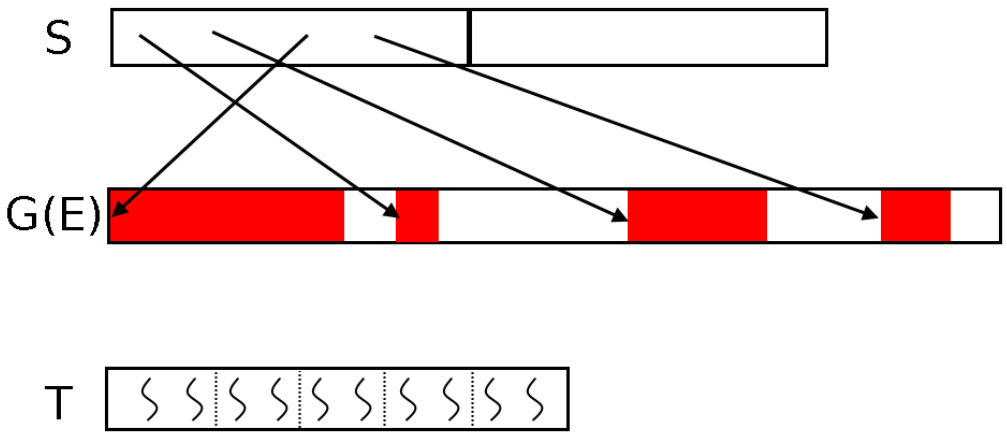
Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization



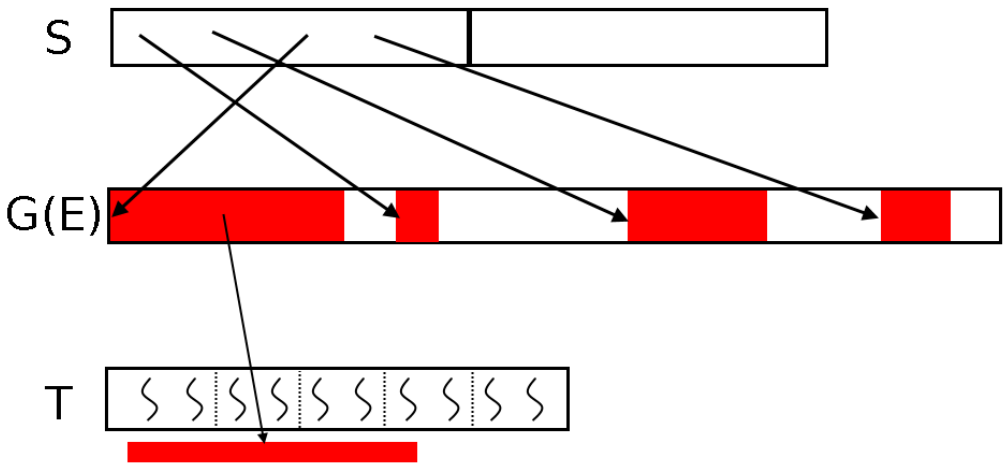
Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g., Hong et al., PPoPP 2011)



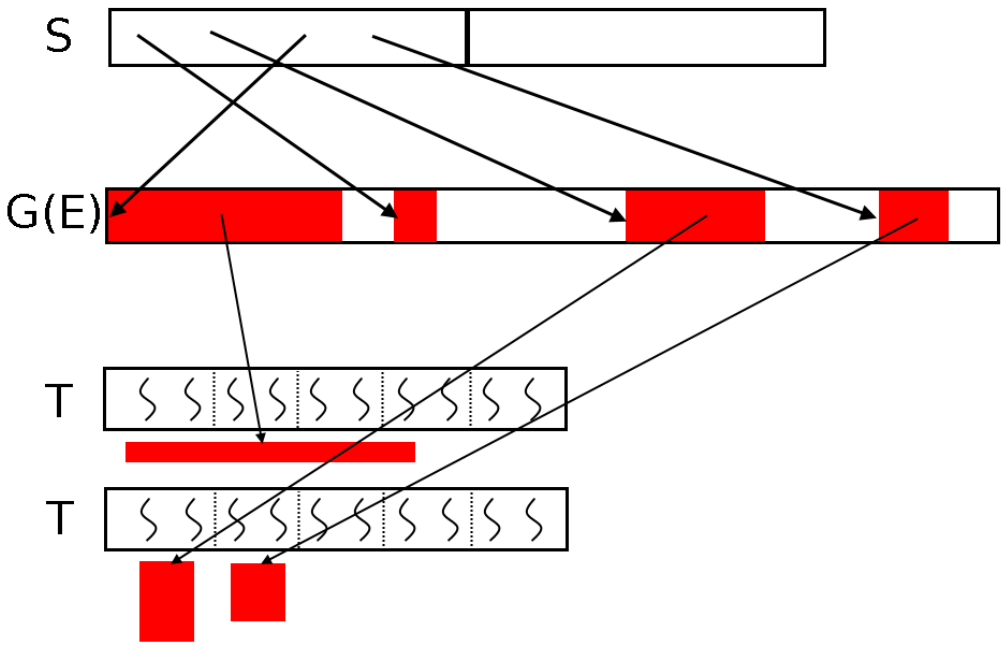
Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g., Hong et al., PPoPP 2011)



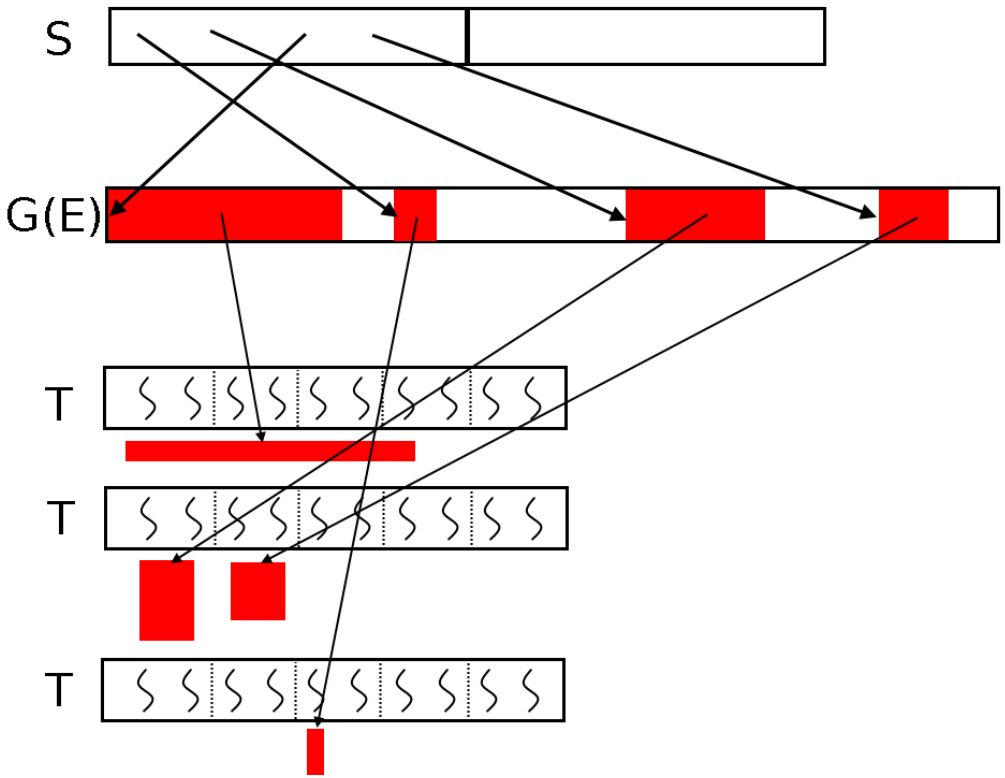
Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g., Hong et al., PPoPP 2011)



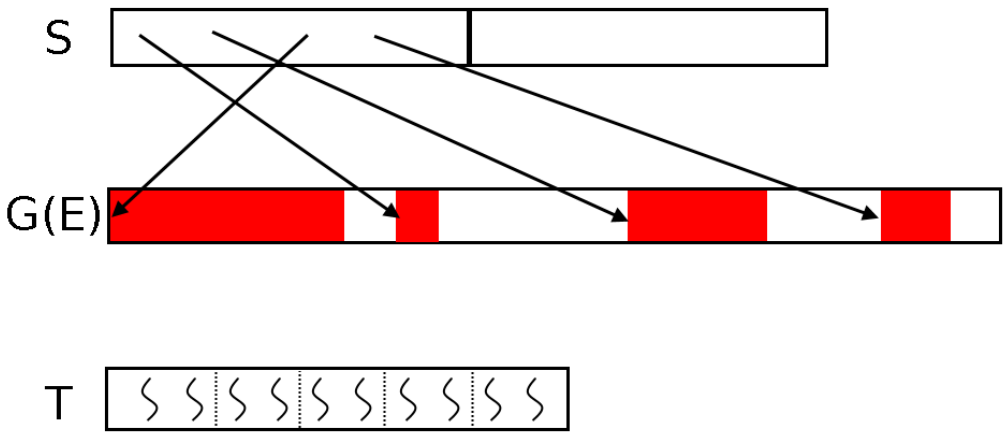
Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g., Hong et al., PPoPP 2011)



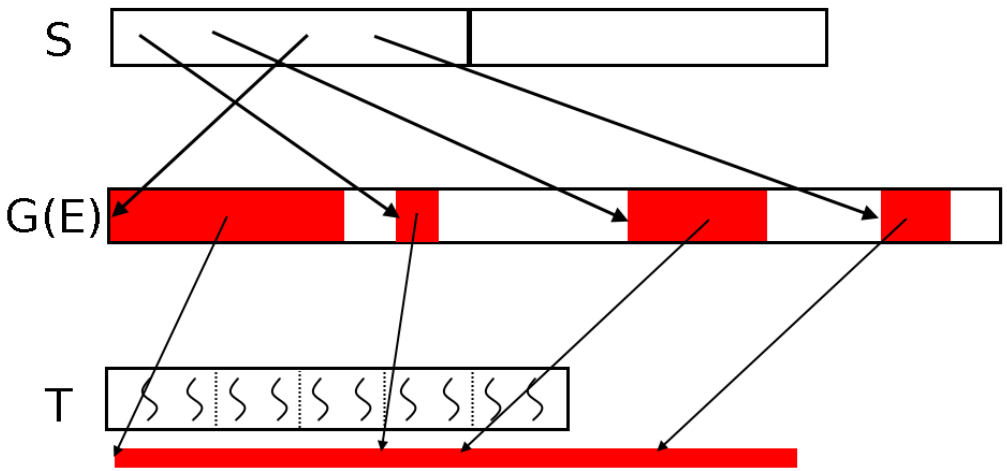
Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g., Hong et al., PPoPP 2011)
- 'Manhattan collapse - local' (e.g.m Merrill et al., PPoPP 2012)



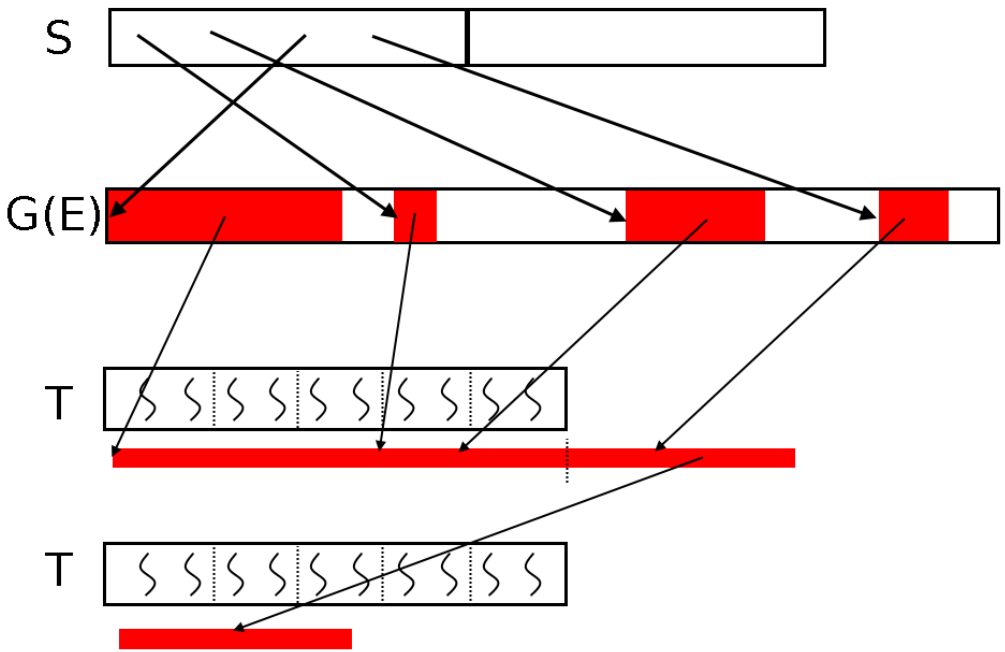
Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g., Hong et al., PPoPP 2011)
- 'Manhattan collapse - local' (e.g.m Merrill et al., PPoPP 2012)



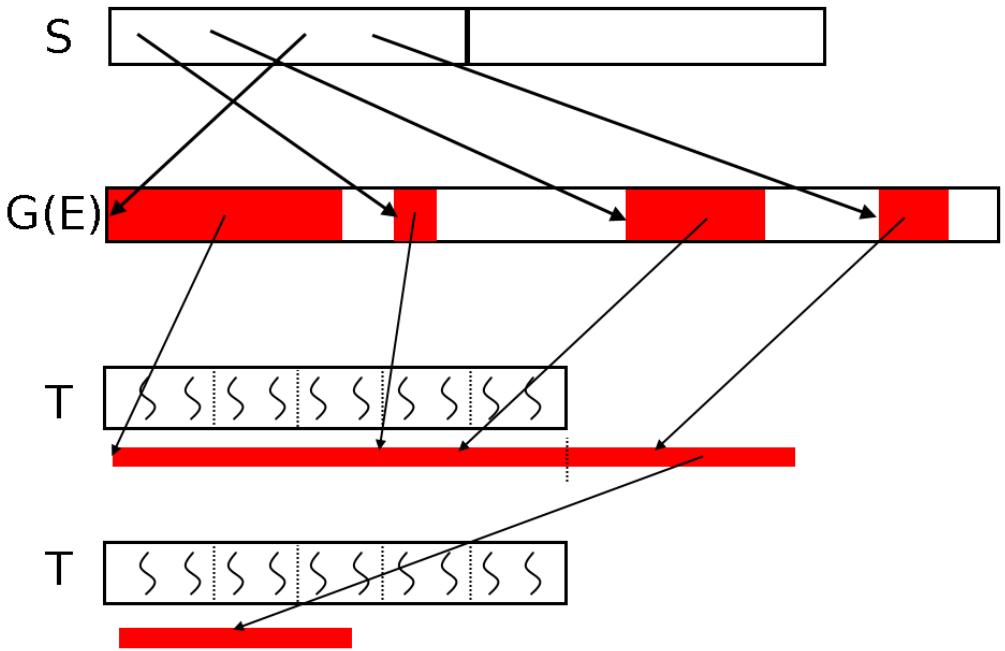
Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g., Hong et al., PPoPP 2011)
- 'Manhattan collapse - local' (e.g.m Merrill et al., PPoPP 2012)



Optimizations for Manycore Processors
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g., Hong et al., PPoPP 2011)
- 'Manhattan collapse - local' (e.g.m Merrill et al., PPoPP 2012)
- 'Manhattan collapse - global' (e.g., Davidson et al., IPDPS 2014)



Optimizations for Manycore Processors
Locality and SIMD Parallelism using Kokkos

- Memory access
  - Explicit shared memory utilization on GPU
  - Coalescing memory access (locality)
  - Minimize access to global/higher-level memory
- Collective operations
  - Warp and team-based operations (team scan, team reduce)
  - Minimize global atomics (team-based atomics)

Graph computations
Implemented algorithms

- Breadth-first search
- Color propagation
- Trimming
- The Multistep algorithm (Slota et al., IPDPS 2014) for Strongly Connected Components (SCC) decomposition

Graph computations
Breadth-first search

- Useful subroutine in other graph computations

```
1:  A_1[1..n] ← −1
2:  S_1[1] ← root
3:  level ← 0
4:  while |S_i| ≠ ∅ do
5:      Initialize S_{i+1}
6:      for j = 1 to |S_i| do
7:          u ← S_i[j]
8:          for k = 1 to |E[u]| do
9:              v ← E[u][k]
10:             if A_1[v] < 0 then
11:                 A_1[v] ← level
12:                 S_{i+1} ← v
13:     level ← level + 1
```

Graph computations
Breadth-first search

- Useful subroutine in other graph computations

```
1:  A_1[1..n] ← −1
2:  S_1[1] ← root
3:  level ← 0
4:  while |S_i| ≠ ∅ do
5:      Initialize S_{i+1}
6:      for j = 1 to |S_i| do
7:          u ← S_i[j]
8:          for k = 1 to |E[u]| do
9:              v ← E[u][k]
10:             if A_1[v] < 0 then
11:                 A_1[v] ← level
12:                 S_{i+1} ← v
13:     level ← level + 1
```

Graph computations
Color propagation

- Basic algorithm for connectivity
- General approach applies to other algorithms (e.g., label propagation)

```
1:  A_1[1..n] ← [1..n]
2:  S_1[1..n] ← [1..n]
3:  while |S_i| ≠ ∅ do
4:      Initialize S_{i+1}
5:      for j = 1 to |S_i| do
6:          u ← S_i[j]
7:          for k = 1 to |E[u]| do
8:              v ← E[u][k]
9:              if A_1[u] > A_1[v] then
10:                 A_1[v] ← A_1[u]
11:                 S_{i+1} ← v
```

Graph computations
Color propagation

- Basic algorithm for connectivity
- General approach applies to other algorithms (e.g., label propagation)

```
1:  A_1[1..n] ← [1..n]
2:  S_1[1..n] ← [1..n]
3:  while |S_i| ≠ ∅ do
4:      Initialize S_{i+1}
5:      for j = 1 to |S_i| do
6:          u ← S_i[j]
7:          for k = 1 to |E[u]| do
8:              v ← E[u][k]
9:              if A_1[u] > A_1[v] then
10:                 A_1[v] ← A_1[u]
11:                 S_{i+1} ← v
```

Graph computations
Trimming

- Routine for accelerating connectivity decomposition
- Iteratively trim 0-degree vertices

```
1:  A_1[1..n] ← 1
2:  S_1[1..n] ← [1..n]
3:  while |S_i| ≠ ∅ do
4:      Initialize S_{i+1}
5:      for j = 1 to |S_i| do
6:          u ← S_i[j]
7:          trim ← true
8:          for k = 1 to |E[u]| do
9:              v ← E[u][k]
10:             if A_1[v] = 1 then
11:                 trim ← false
12:         if trim = true then
13:             A_1[u] ← 0
14:             S_{i+1} ← E[u]
```

Graph computations
Trimming

- Routine for accelerating connectivity decomposition
- Iteratively trim 0-degree vertices

```
1:  A_1[1..n] ← 1
2:  S_1[1..n] ← [1..n]
3:  while |S_i| ≠ ∅ do
4:      Initialize S_{i+1}
5:      for j = 1 to |S_i| do
6:          u ← S_i[j]
7:          trim ← true
8:          for k = 1 to |E[u]| do
9:              v ← E[u][k]
10:             if A_1[v] = 1 then
11:                 trim ← false
12:         if trim = true then
13:             A_1[u] ← 0
14:             S_{i+1} ← E[u]
```

Graph computations
Multistep SCC decomposition (Slota et al., IPDPS 2014)

- Combination of trimming, BFS, and color propagation

1: $T \leftarrow \text{Trim}(G)$
2: $V \leftarrow V \setminus T$
3: Select $v \in V$ for which $d_{in}(v) * d_{out}(v)$ is maximal
4: $D \leftarrow \text{BFS}(G(V, E(V)), v)$
5: $S \leftarrow D \cap \text{BFS}(G(D, E'(D)), v)$
6: $V \leftarrow V \setminus S$
7: **while** $\text{NumVerts}(V) > 0$ **do**
8: $\quad C \leftarrow \text{ColorProp}(G(V, E(V)))$
9: $\quad V \leftarrow V \setminus C$

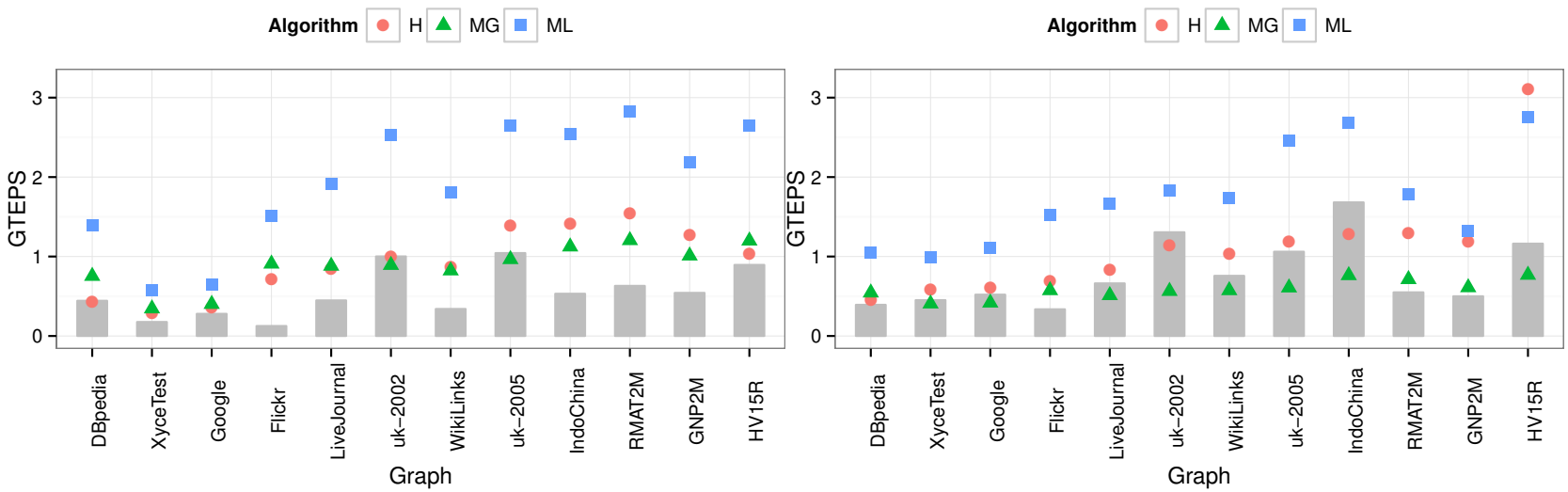
Experimental Setup

- Test systems: One node of *Shannon* and *Compton* at Sandia, Blue Waters at NCSA
  - Intel Xeon E5-2670 (Sandy Bridge), dual-socket, 16 cores, 64-128 GB memory
  - NVIDIA Tesla K40M GPU, 2880 cores, 12 GB memory
  - NVIDIA Tesla K20X GPU, 2688 cores, 6 GB memory
  - Intel Xeon Phi (KNC, $\sim$3120A), 228 cores, 6 GB memory
- Test graphs:
  - Various real and synthetic small-world graphs, 5.1 M to 936 M edges
  - Social networks, circuit, mesh, RDF graph, web crawls, R-MAT and $G(n, p)$, Wikipedia article links

Results

BFS and Coloring versus loop strategies

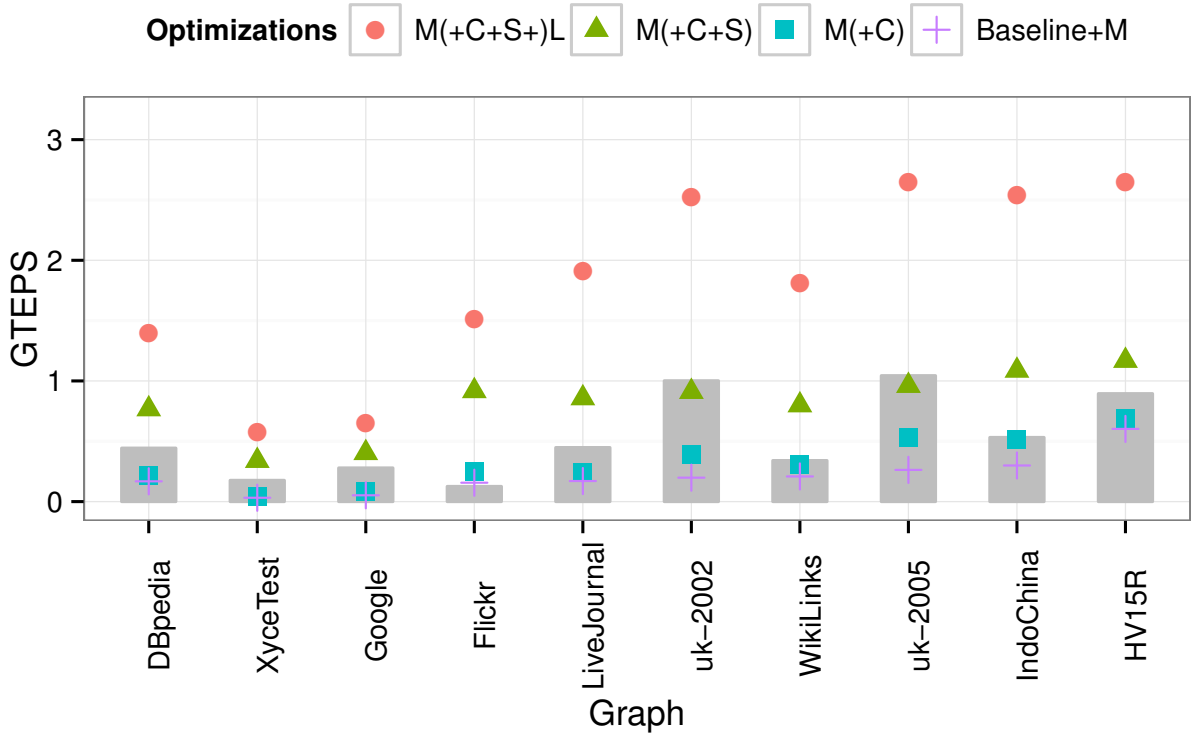
- Performance in GTEPS (10^9 trav. edges per second) for BFS (left) and color propagation (right) on Tesla K40M.
- H: Hierarchical, ML: Local collapse, MG: Global collapse, gray bar: Baseline



Results

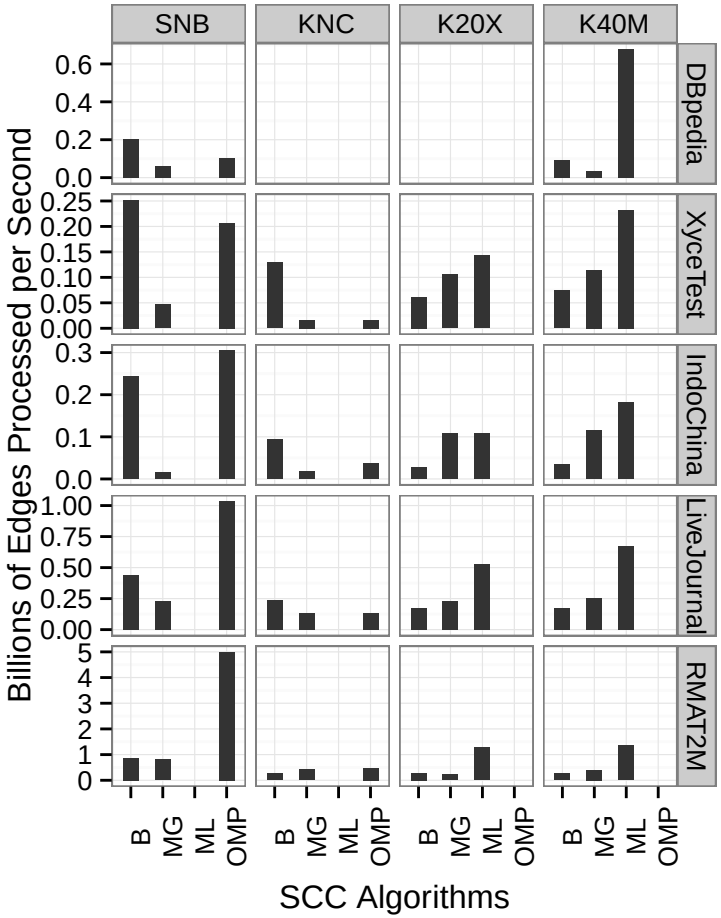
BFS performance and cumulative impact of optimizations, Tesla K40M

- M: local collapse, C: coalescing memory access, S: shared memory use, L: local team-based primitives



Results
SCC cross-platform performance comparison

- B: Baseline, MG: Manhattan Global, ML: Manhattan Local, OMP: Optimized OpenMP code



Conclusions

- We express several graph computations in the **Kokkos** programming model using an **algorithm design abstraction** that allows portability across both multicore platforms and **accelerators**.
- The **SCC** code on GPUs (using the **Local Manhattan Collapse** strategy) demonstrates up to a $3.25\times$ speedup relative to a state-of-the-art parallel CPU implementation running on a dual-socket compute node.
- Future work: Expressing other computations using this framework; Heterogeneous CPU-GPU processing; Newer architectures.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070, ACI-1238993, and ACI-1444747) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This work is also supported by NSF grants ACI-1253881, CCF-1439057, and the DOE Office of Science through the FASTMath SciDAC Institute. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Today: In class work

- Develop Manhattan Collapse method for manycore parallelism
- Implement BFS using Manhattan Collapse
- Implement PageRank using Manhattan Collapse

Blank code and data available on website (Lecture 23)

`www.cs.rpi.edu/~slotag/classes/FA16/index.html`