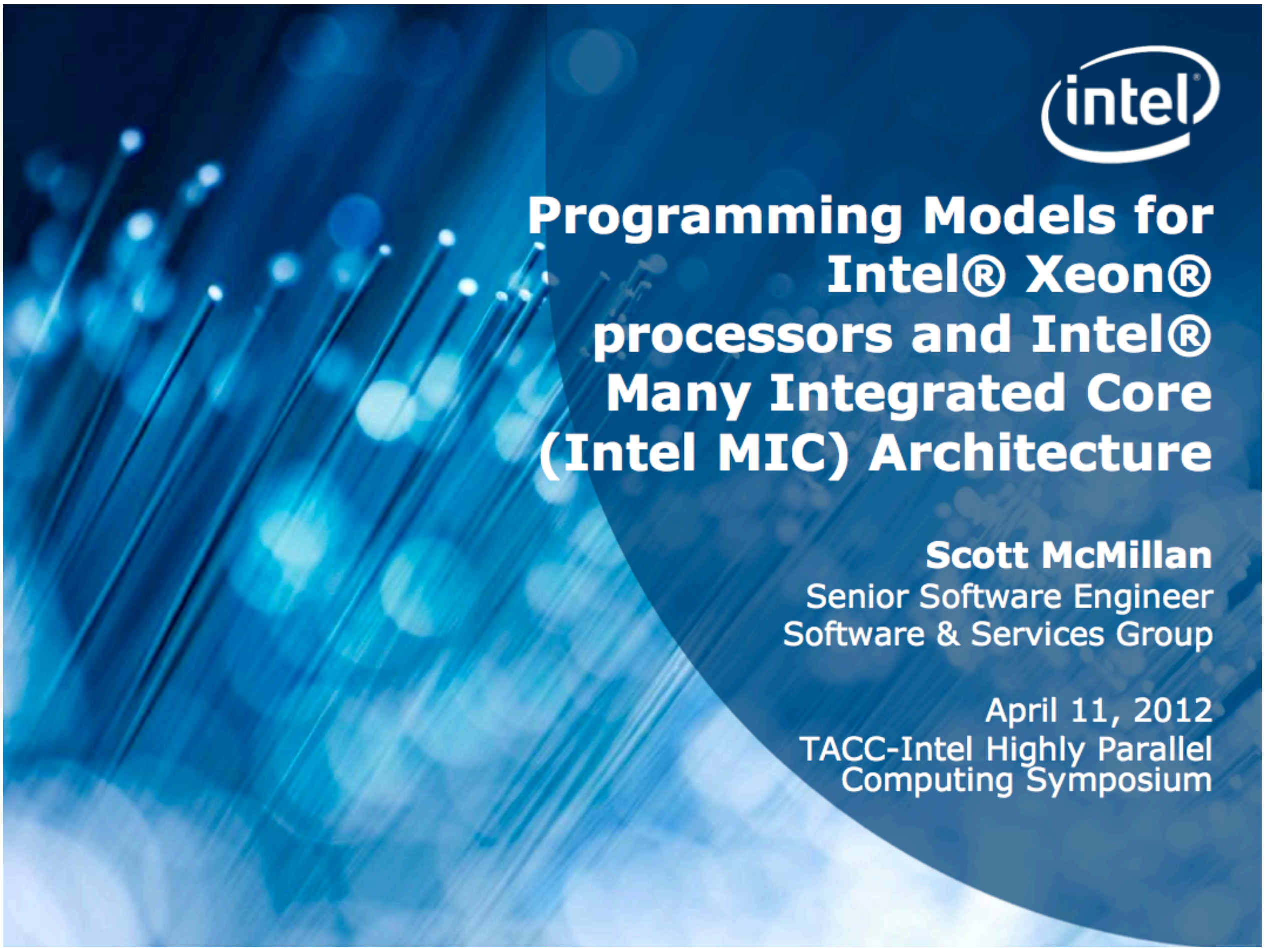
MSc Informatics Eng.

2015/16

A.J.Proença

Data Parallelism 3 (*MIC/CUDA programming*)

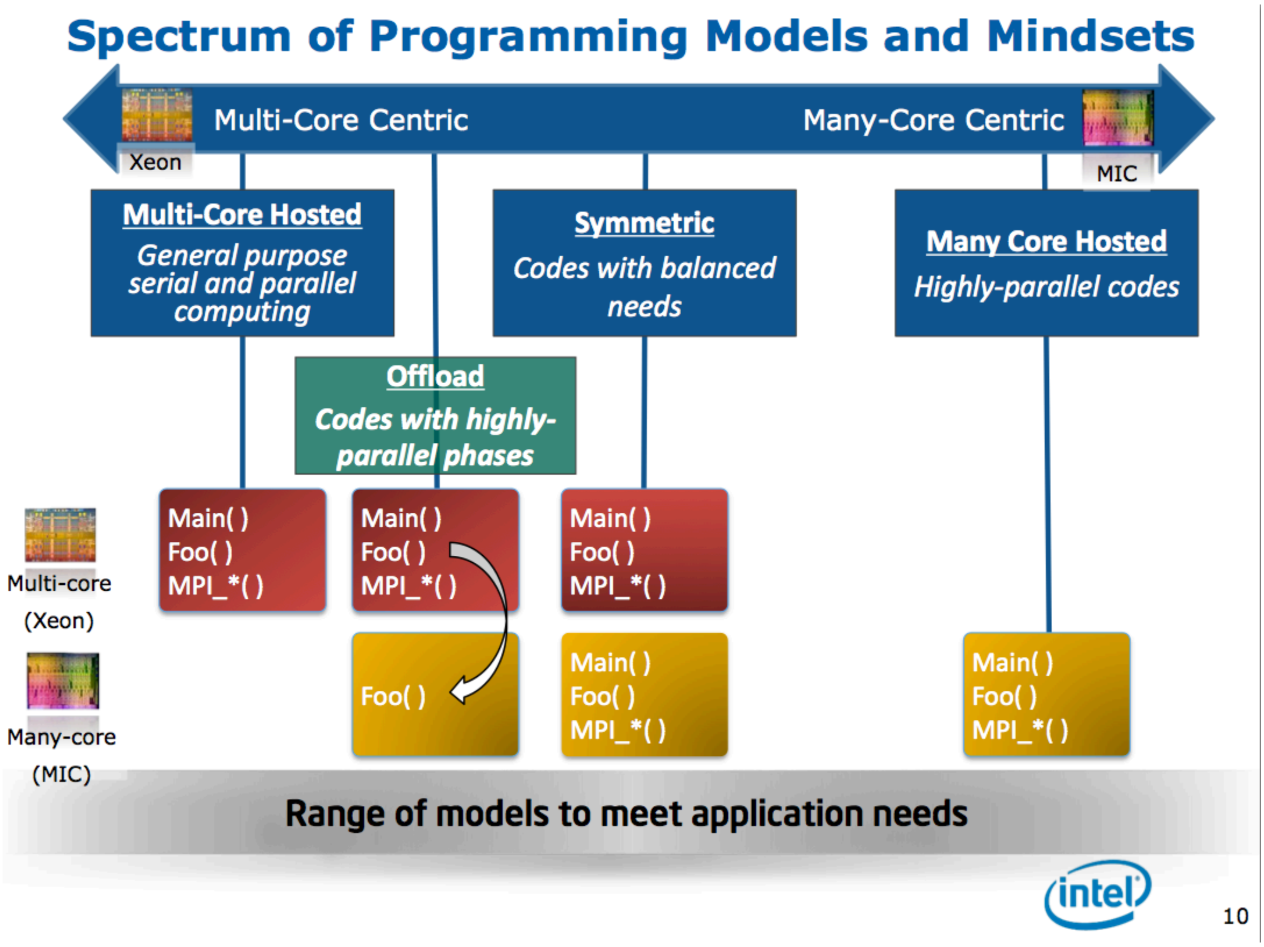
(most slides are borrowed)



Programming Models for Intel® Xeon® processors and Intel® Many Integrated Core (Intel MIC) Architecture

Scott McMillan
Senior Software Engineer
Software & Services Group

April 11, 2012
TACC-Intel Highly Parallel
Computing Symposium



Spectrum of Programming Models and Mindsets

Xeon Multi-Core Centric

Many-Core Centric MIC

Multi-Core Hosted

General purpose serial and parallel computing

Offload

Codes with highly-parallel phases

Symmetric

Codes with balanced needs

Many Core Hosted

Highly-parallel codes

Multi-core (Xeon)

Many-core (MIC)

Main( )
Foo( )
MPI_*( )

Main( )
Foo( )
MPI_*( )

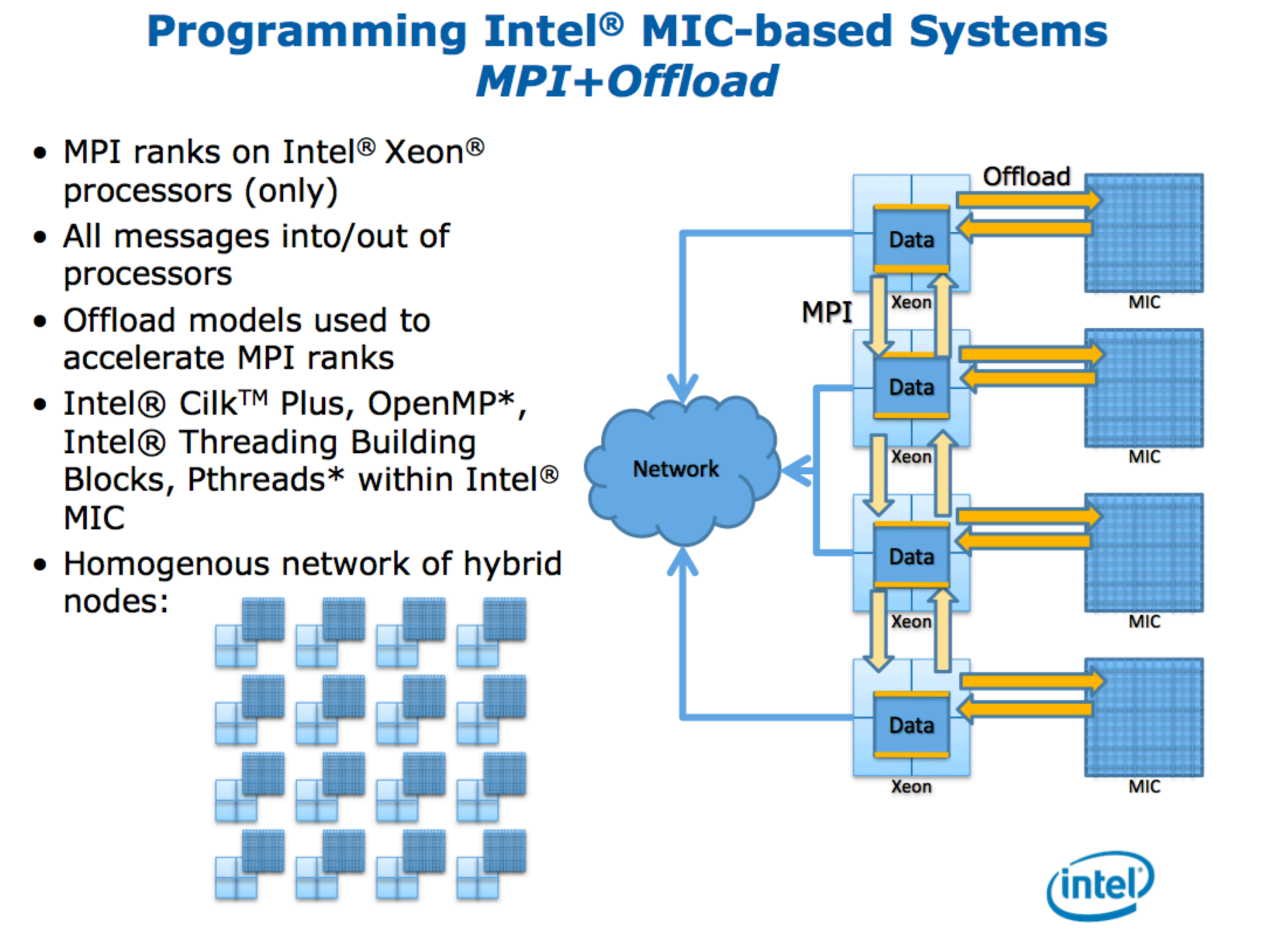
Foo( )

Main( )
Foo( )
MPI_*( )

Main( )
Foo( )
MPI_*( )

Main( )
Foo( )
MPI_*( )

Range of models to meet application needs



Programming Intel® MIC-based Systems *MPI+Offload*

- MPI ranks on Intel® Xeon® processors (only)
- All messages into/out of processors
- Offload models used to accelerate MPI ranks
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* within Intel® MIC
- Homogenous network of hybrid nodes:

Offload Code Examples

- C/C++ Offload Pragma

```
#pragma offload target (mic)
#pragma omp parallel for reduction(+:pi)
 for (i=0; i<count; i++)  {
      float t = (float)((i+0.5)/count);
      pi += 4.0/(1.0+t*t);
 }
 pi /= count;
```

- Function Offload Example

```
#pragma offload target(mic)
      in(transa, transb, N, alpha, beta) \
      in(A:length(matrix_elements)) \
      in(B:length(matrix_elements)) \
      inout(C:length(matrix_elements))
        sgemm(&transa, &transb, &N, &N, &N,
&alpha, A, &N, B, &N, &beta, C, &N);
```

- Fortran Offload Directive

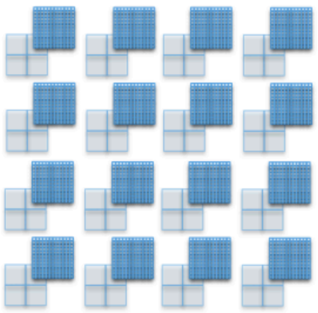
```
!dir$ omp offload target(mic)
!$omp parallel do
    do i=1,10
              A(i) = B(i) * C(i)
    enddo
```

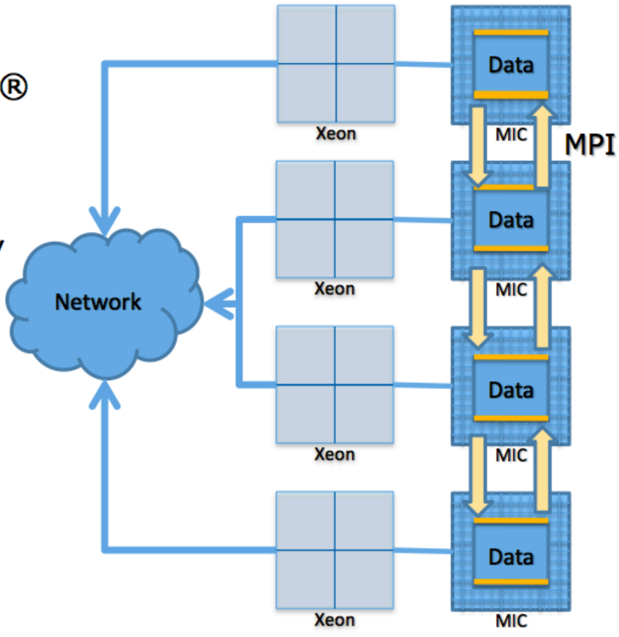
- C/C++ Language Extension

```
class _Cilk_Shared common {
   int data1;
   char *data2;
   class common *next;
   void process();
};
_Cilk_Shared class common  obj1, obj2;
_Cilk_spawn   _Offload obj1.process();
_Cilk_spawn              obj2.process();
```

Programming Intel® MIC-based Systems
Many-core Hosted

- MPI ranks on Intel® MIC (only)
- All messages into/out of Intel® MIC
- Intel® Cilk^{TM} Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads used directly within MPI processes
- Programmed as homogenous network of many-core CPUs:

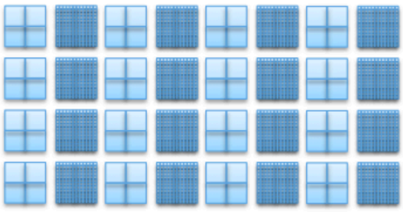


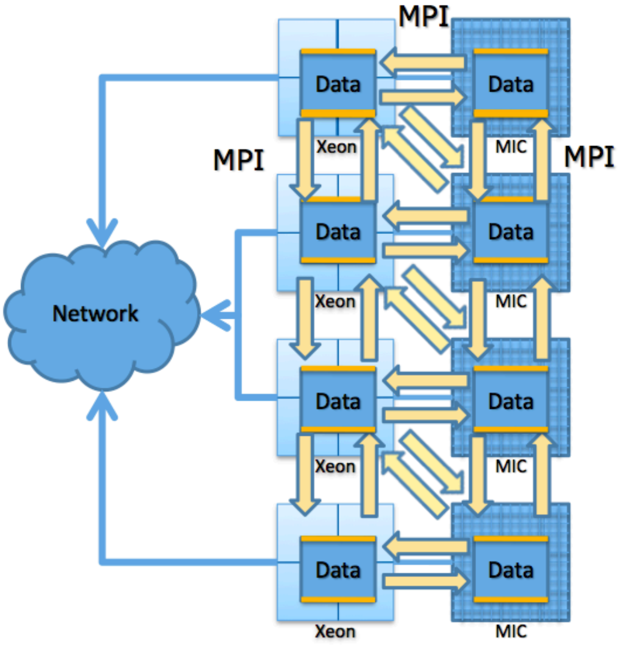


Programming Intel® MIC-based Systems *Symmetric*

- MPI *ranks* on Intel® MIC and Intel® Xeon® processors
- Messages to/from any core
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* used directly within MPI processes

- Programmed as heterogeneous network of homogeneous nodes:





Keys to Productive Performance on Intel® MIC Architecture

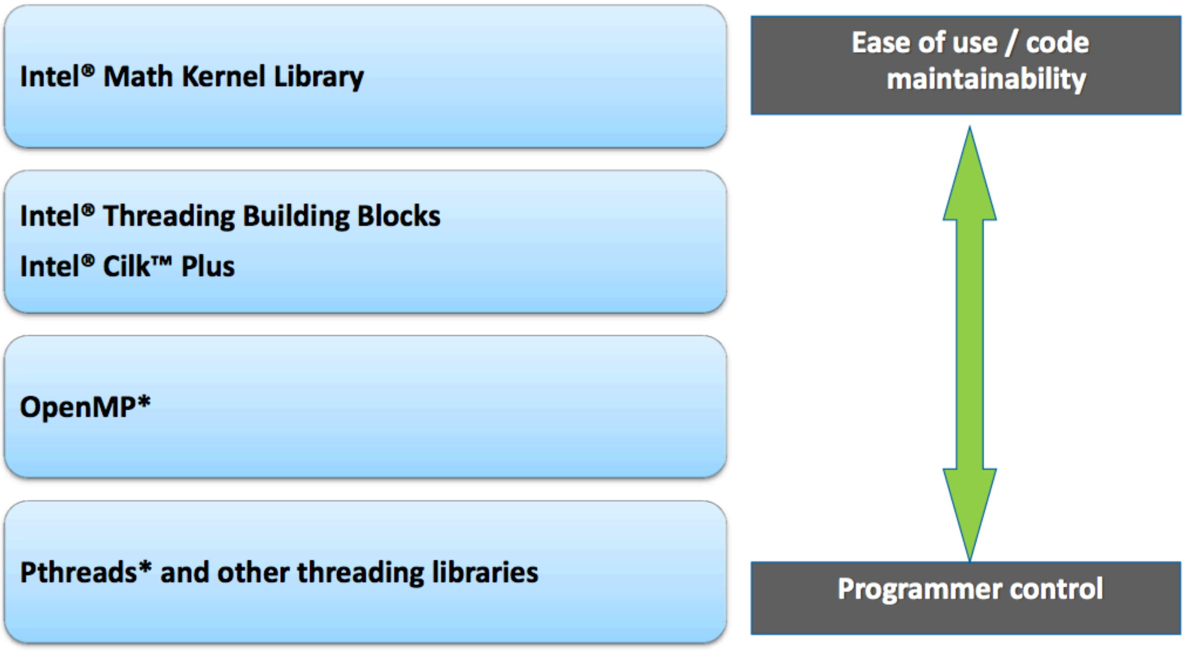
- Choose the right Multi-core centric or Many-core centric model for your application

- Vectorize your application (today)
  - Use the Intel vectorizing compiler

- Parallelize your application (today)
  - With MPI (or other multi-process model)
  - With threads (via Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads, etc.)

- Go asynchronous to overlap computation and communication

Options for Thread Parallelism



Intel® Math Kernel Library

Intel® Threading Building Blocks

Intel® Cilk™ Plus

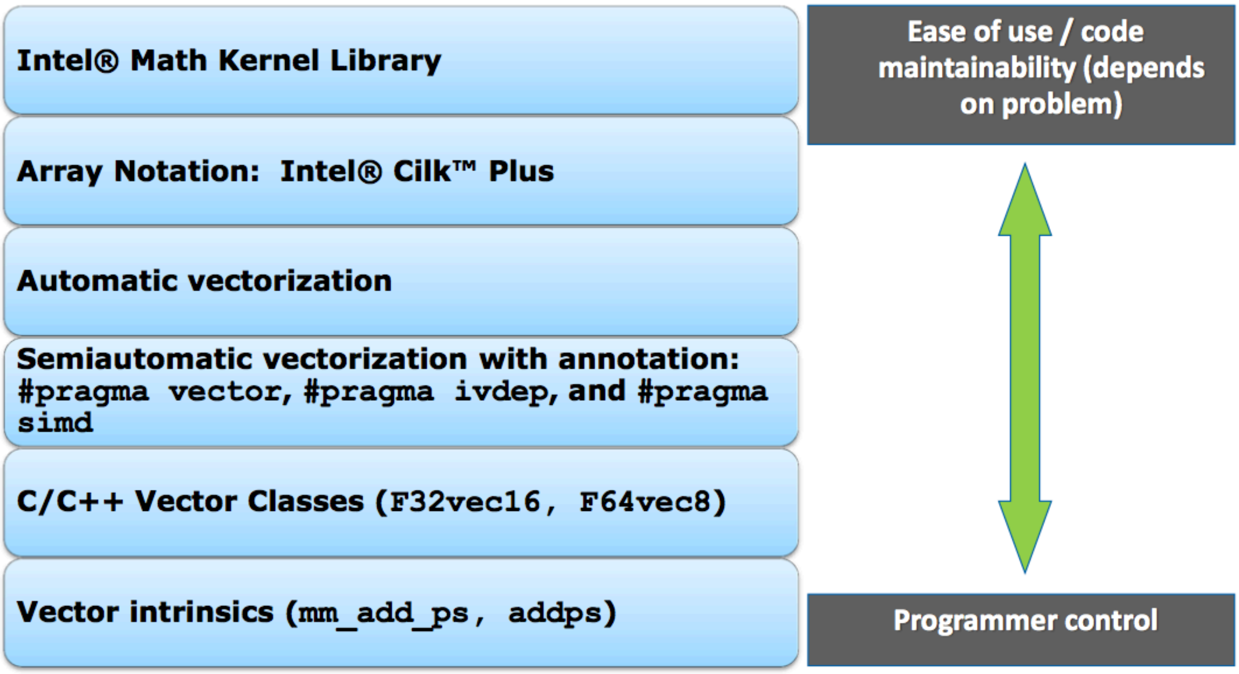
OpenMP*

Pthreads* and other threading libraries

Ease of use / code maintainability

Programmer control

Options for Vectorization



Intel® Math Kernel Library

Array Notation: Intel® Cilk™ Plus

Automatic vectorization

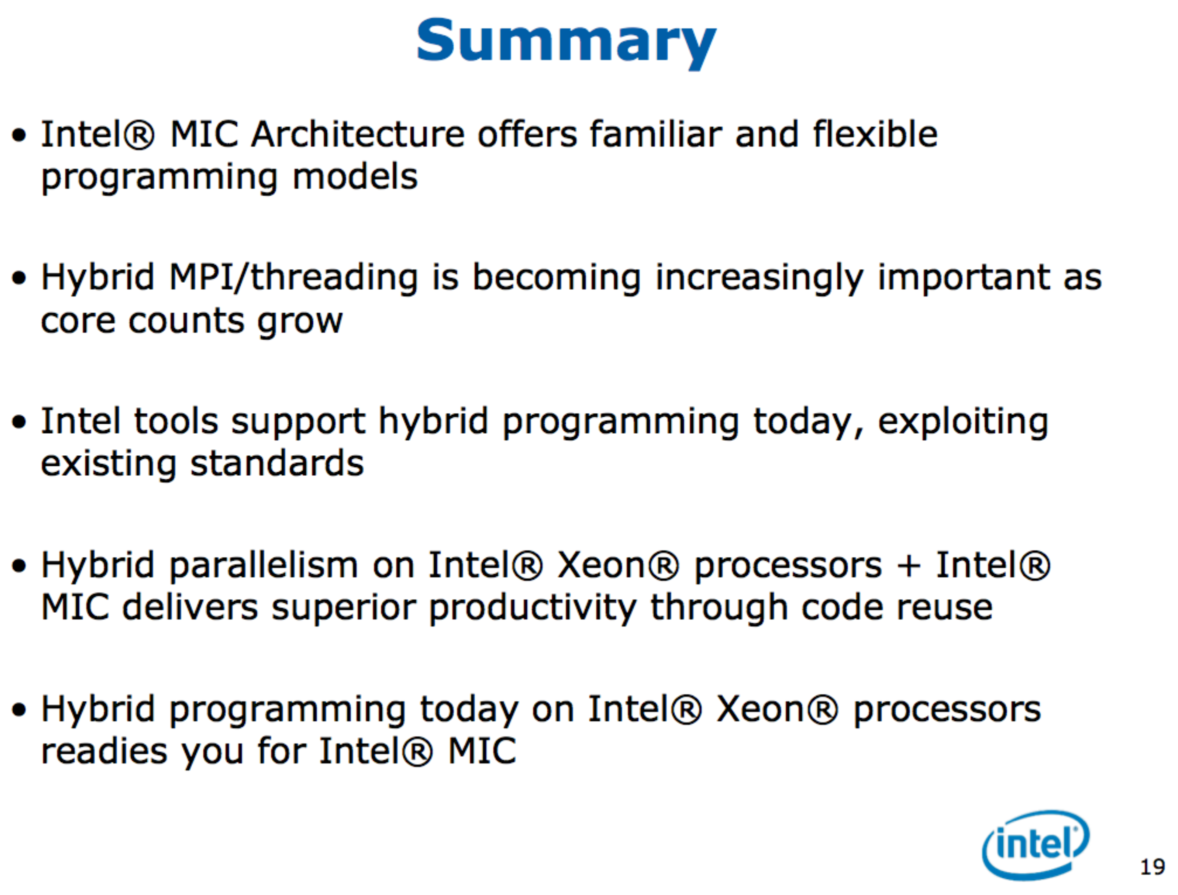
Semiautomatic vectorization with annotation: `#pragma vector`, `#pragma ivdep`, and `#pragma simd`

C/C++ Vector Classes (`F32vec16, F64vec8`)

Vector intrinsics (`mm_add_ps, addps`)

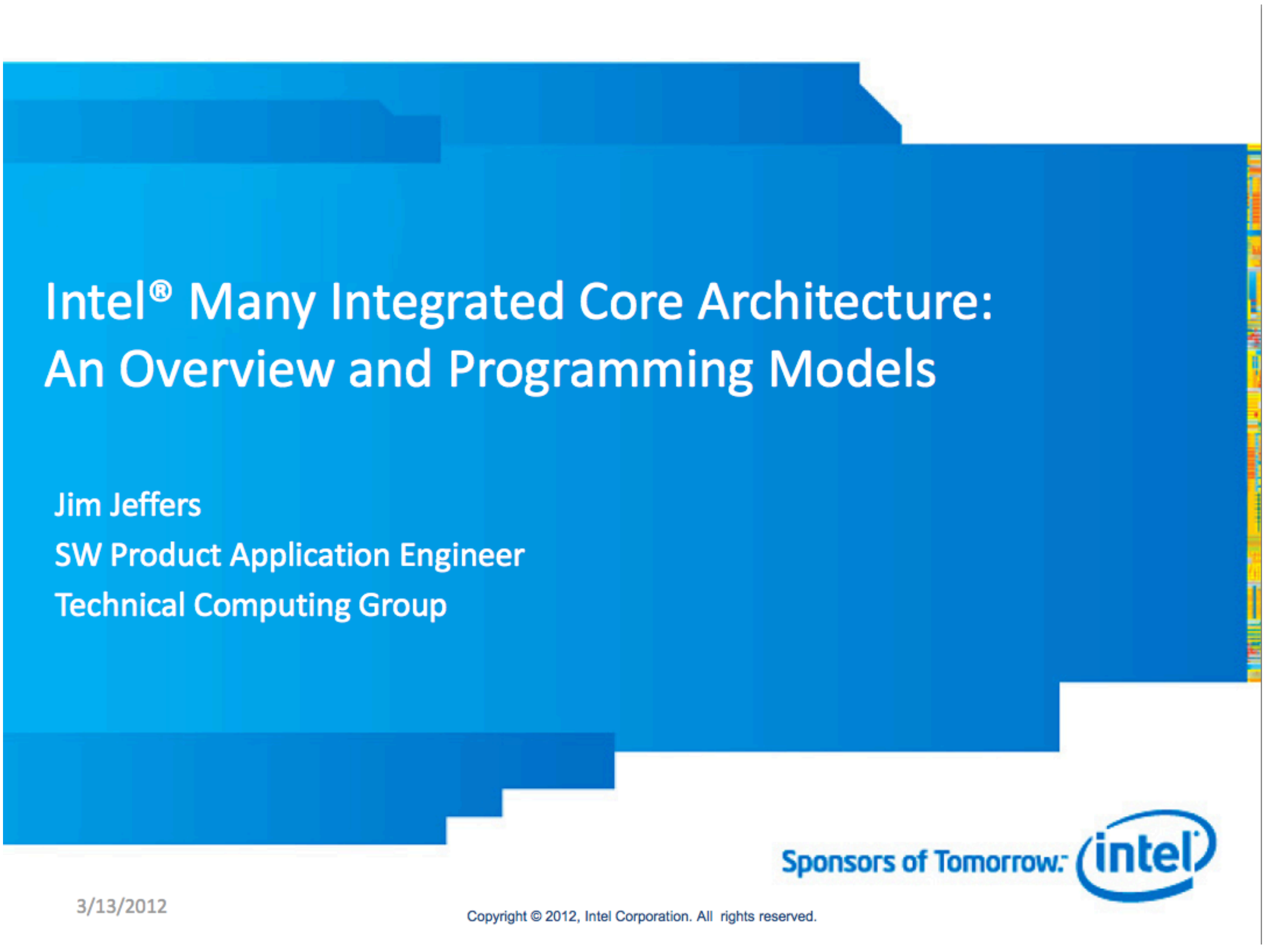
Ease of use / code maintainability (depends on problem)

Programmer control



Summary

- Intel® MIC Architecture offers familiar and flexible programming models
- Hybrid MPI/threading is becoming increasingly important as core counts grow
- Intel tools support hybrid programming today, exploiting existing standards
- Hybrid parallelism on Intel® Xeon® processors + Intel® MIC delivers superior productivity through code reuse
- Hybrid programming today on Intel® Xeon® processors readies you for Intel® MIC



Intel® Many Integrated Core Architecture: An Overview and Programming Models

Jim Jeffers
SW Product Application Engineer
Technical Computing Group

Sponsors of Tomorrow.

3/13/2012

"Stand-alone" Intel® MIC Architecture Computing Environment

- Intel® MIC Architecture software environment includes a highly functional, Linux* OS running on the co-processor with:
  - A familiar interactive shell
  - IP Addressability [headless node]
  - A local file system with subdirectories, file reads, writes, etc
  - standard i/o including printf
  - Virtual memory management
  - Process, thread management & scheduling
  - Interrupt and exception handling
  - Semaphores, mutexes, etc…
- What does this mean?
  - A large majority of existing code even with OS oriented calls like fork() can port with a simple recompile
  - Intel MIC Architecture natively supports parallel coding models like Intel® Cilk™ Plus, Intel® Threading Building Blocks, pThreads*, OpenMP*

fooey.c

```
main( )
{
   printf("running Foo()\n");
   Foo( );
}

Foo()
{
   printf("fooey\n");
}
```

Intel MIC Architecture
(Knights Corner console)

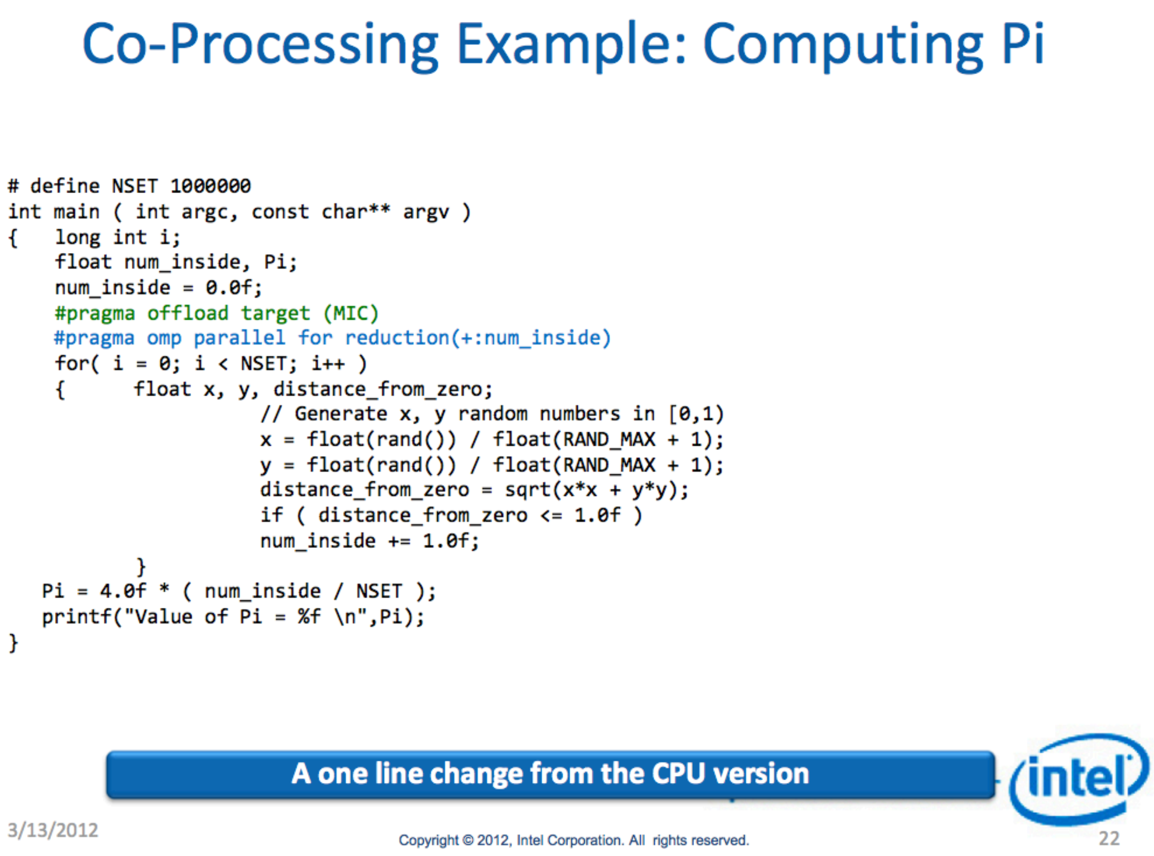
```
mymic>ls
 fooey
mymic>./fooey
running Foo()
fooey
mymic>
```

Sponsors of Tomorrow.

Stand-alone Example: Computing Pi

```
# define NSET 1000000
int main ( int argc, const char** argv )
{   long int i;
    float num_inside, Pi;
    num_inside = 0.0f;
    #pragma omp parallel for reduction(+:num_inside)
    for( i = 0; i < NSET; i++ )
    {      float x, y, distance_from_zero;
                  // Generate x, y random numbers in [0,1)
                  x = float(rand()) / float(RAND_MAX + 1);
                  y = float(rand()) / float(RAND_MAX + 1);
                  distance_from_zero = sqrt(x*x + y*y);
                  if ( distance_from_zero <= 1.0f )
                  num_inside += 1.0f;
           }
   Pi = 4.0f * ( num_inside / NSET );
   printf("Value of Pi = %f \n",Pi);
}
```

Original Source Code
Compiler command line switch targets platform



Co-Processing Example: Computing Pi

```
# define NSET 1000000
int main ( int argc, const char** argv )
{   long int i;
    float num_inside, Pi;
    num_inside = 0.0f;
    #pragma offload target (MIC)
    #pragma omp parallel for reduction(+:num_inside)
    for( i = 0; i < NSET; i++ )
    {      float x, y, distance_from_zero;
                     // Generate x, y random numbers in [0,1)
                     x = float(rand()) / float(RAND_MAX + 1);
                     y = float(rand()) / float(RAND_MAX + 1);
                     distance_from_zero = sqrt(x*x + y*y);
                     if ( distance_from_zero <= 1.0f )
                     num_inside += 1.0f;
          }
   Pi = 4.0f * ( num_inside / NSET );
   printf("Value of Pi = %f \n",Pi);
}
```

A one line change from the CPU version

3/13/2012

22

The CUDA programming model

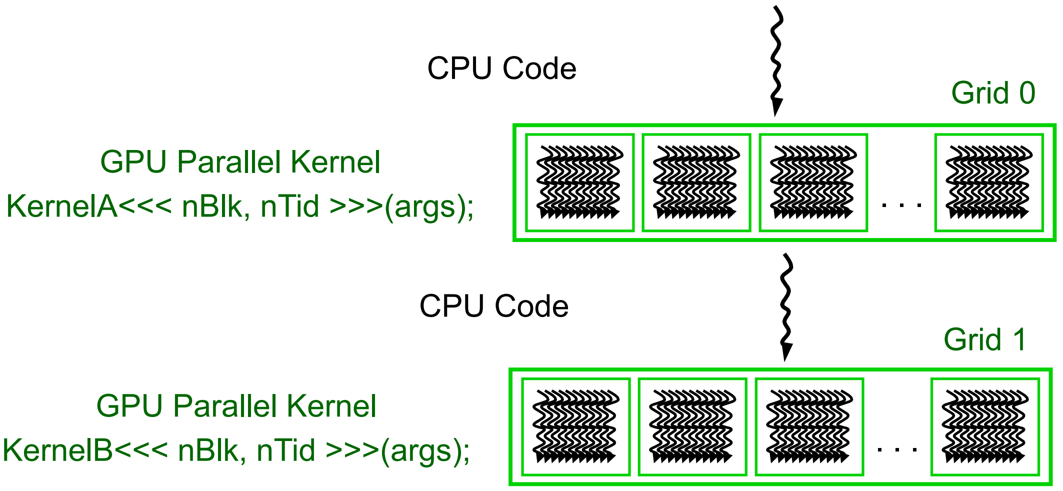
- ***C**ompute **U**nified **D**evice **A**rchitecture*
- CUDA is a recent programming model, designed for
  - a multicore CPU ***host*** coupled to a many-core ***device***, where
  - *devices* have wide SIMD/SIMT parallelism, and
  - the *host* and the *device* do not share memory
- CUDA provides:
  - a thread abstraction to deal with SIMD
  - synchr. & data sharing between small groups of threads
- CUDA programs are written in C with extensions
- OpenCL inspired by CUDA, but hw & sw vendor neutral
  - programming model essentially identical

CUDA Devices and Threads

- A compute device
  - is a coprocessor to the CPU or host
  - has its own DRAM (device memory)
  - runs many threads in parallel
  - is typically a GPU but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device kernels which run on many threads - **SIMT**
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - very little creation overhead, requires LARGE register bank
  - GPU needs 1000s of threads for full efficiency
    - multi-core CPU needs only a few

CUDA basic model: Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks



CPU Code

Grid 0

GPU Parallel Kernel
KernelA<<< nBlk, nTid >>>(args);

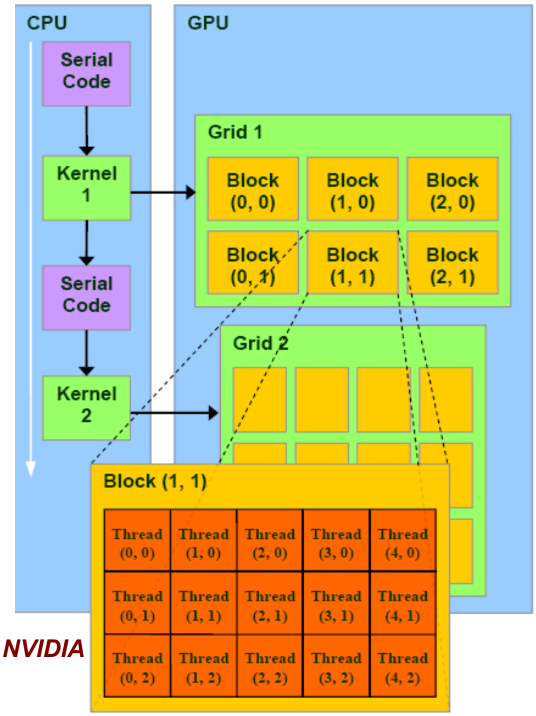
CPU Code

Grid 1

GPU Parallel Kernel
KernelB<<< nBlk, nTid >>>(args);

Programming Model: SPMD + SIMT/SIMD

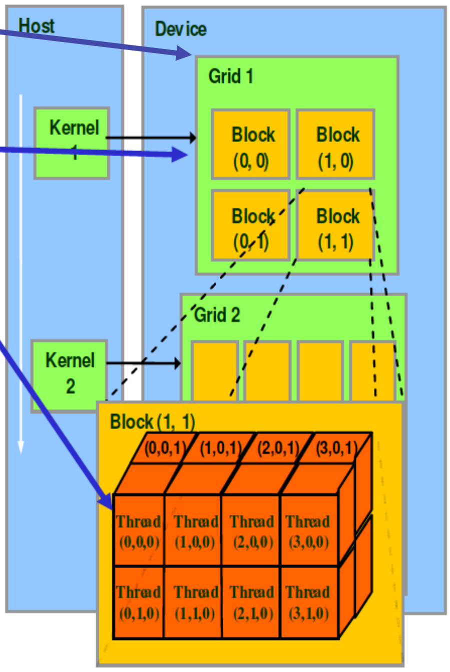
- Hierarchy
  - Device => Grids
  - Grid => Blocks
  - Block => Warps
  - Warp => Threads
- Single kernel runs on multiple blocks (SPMD)
- Threads within a warp are executed in a lock-step way called single-instruction multiple-thread (SIMT)
- Single instruction are executed on multiple threads (SIMD)
  - Warp size defines SIMD granularity (32 threads)
- Synchronization within a block uses shared memory



Courtesy NVIDIA

The Computational Grid: Block IDs and Thread IDs

- A kernel runs on a computational grid of thread blocks
  - Threads share global memory
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- A thread block is a batch of threads that can cooperate by:
  - Sync their execution w/ barrier
  - Efficiently sharing data through a low latency shared memory
  - Two threads from two different blocks cannot cooperate



Example

- Multiply two vectors of length 8192
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - 512 threads per block
  - SIMD instruction executes 32 elements at a time
  - Thus grid size = 16 blocks
  - Block is analogous to a strip-mined vector loop with vector length of 32
  - Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
  - Current-generation GPUs (Fermi) have 7-16 multithreaded SIMD processors

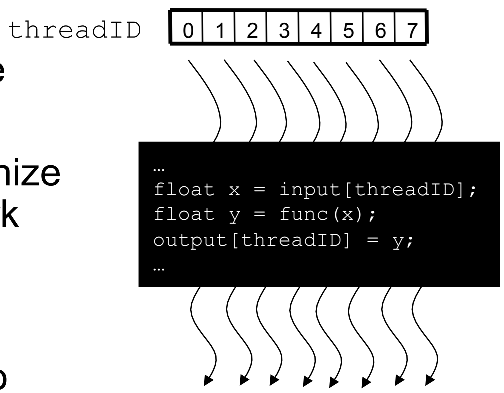
Terminology *(and **in NVidia**)*

- *Threads of SIMD instructions* (***warps***)
  - Each has its own IP (up to 48/64 per SIMD processor, Fermi/Kepler)
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Threads are organized into blocks & executed in groups of 32 threads *(**thread block**)*
    - Blocks are organized into a grid
- The thread block scheduler schedules blocks to SIMD processors *(**Streaming Multiprocessors**)*
- Within each SIMD processor:
  - 32 SIMD lanes *(**thread processors**)*
  - Wide and shallow compared to vector processors

CUDA Thread Block

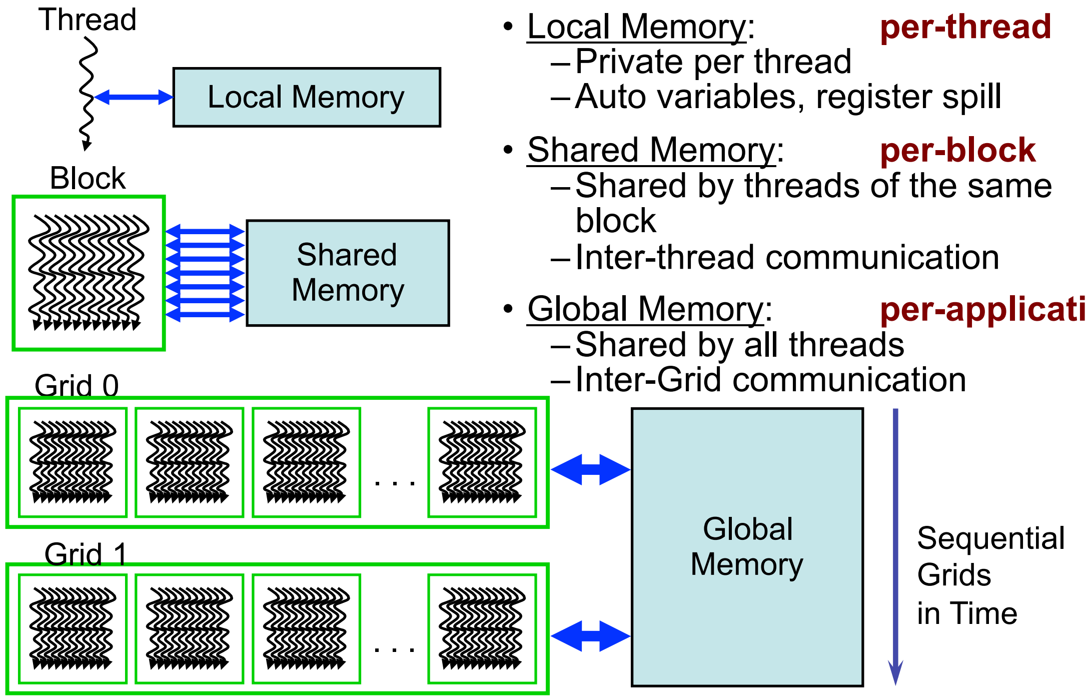
- Programmer declares (Thread) Block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- All threads in a Block execute the same thread program
- Threads share data and synchronize while doing their share of the work
- Threads have thread id numbers within Block
- Thread program uses thread id to select work and address shared data

CUDA Thread Block



```
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

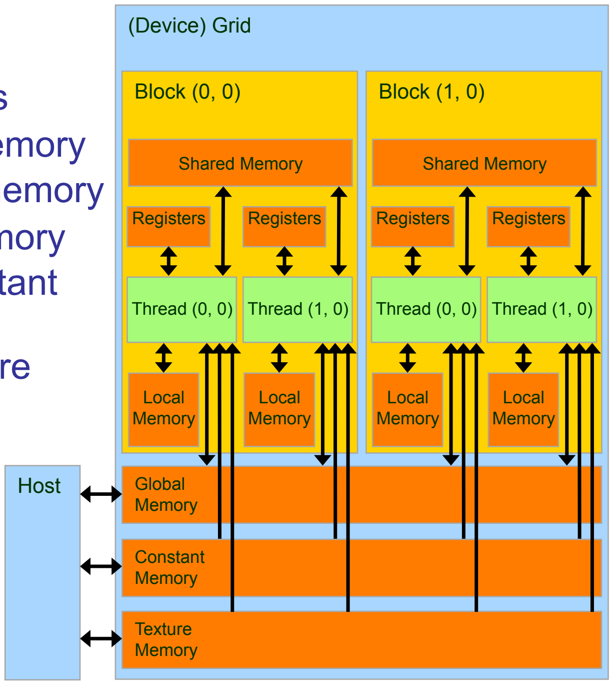
Parallel Memory Sharing



- Local Memory: **per-thread**
  - Private per thread
  - Auto variables, register spill
- Shared Memory: **per-block**
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: **per-application**
  - Shared by all threads
  - Inter-Grid communication

CUDA Memory Model Overview

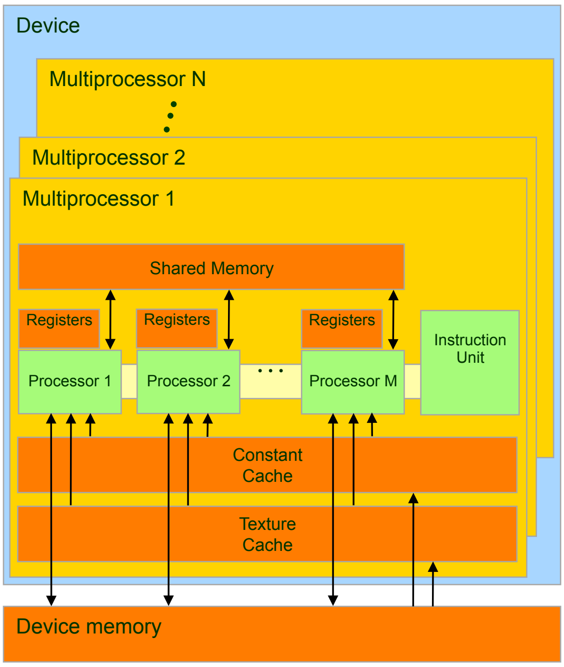
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign

Hardware Implementation: Memory Architecture

- Device memory (DRAM)
  - Slow (2~300 cycles)
  - Local, global, constant, and texture memory
- On-chip memory
  - Fast (1 cycle)
  - Registers, shared memory, constant/texture cache

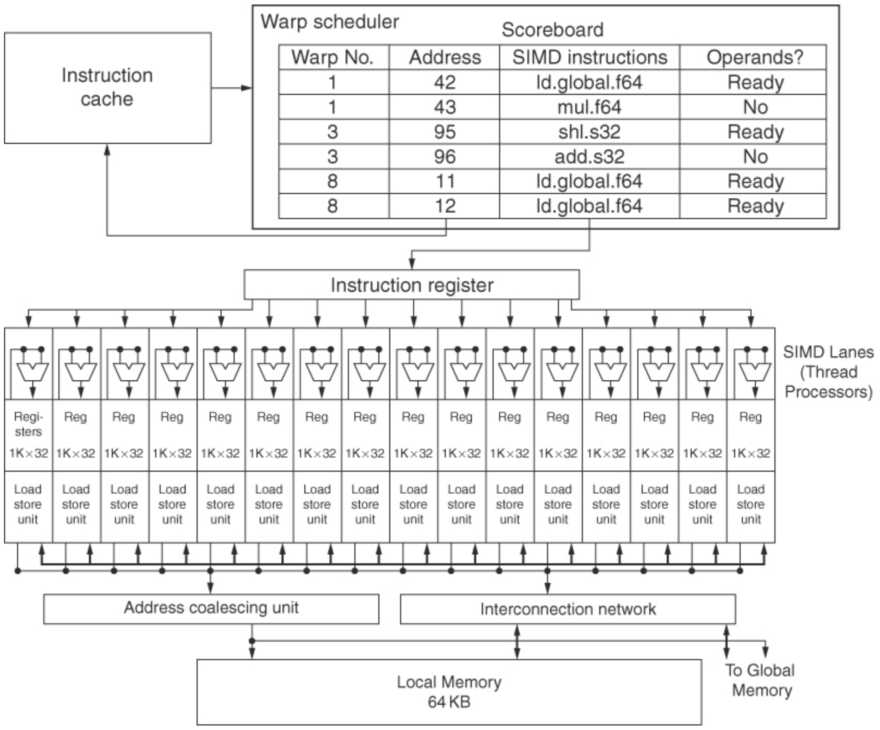


Courtesy NVIDIA

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of **off-chip DRAM**
  - "Private memory" *(Local Memory)*
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory *(Shared Memory)*
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory *(Global Memory)*
  - Host can read and write GPU memory

Example

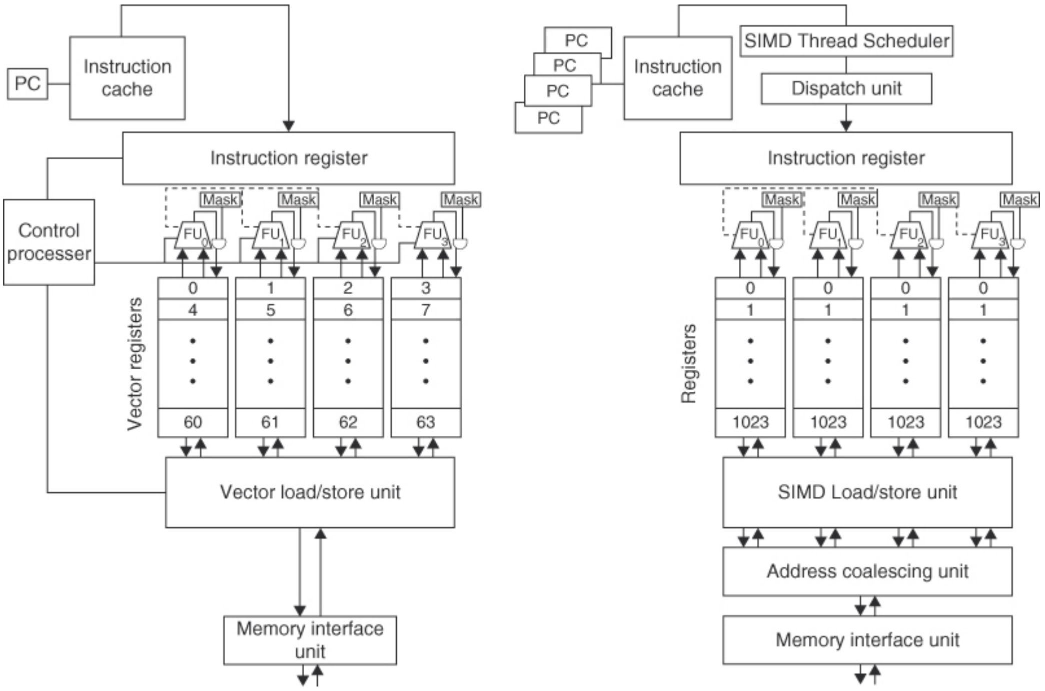


Warp scheduler

Scoreboard

Warp No.	Address	SIMD instructions	Operands?
1	42	ld.global.f64	Ready
1	43	mul.f64	No
3	95	shl.s32	Ready
3	96	add.s32	No
8	11	ld.global.f64	Ready
8	12	ld.global.f64	Ready

Vector Processor *versus* CUDA core



Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer