Many Core Programming
Xeon Phi & GPU

Thoai Nam

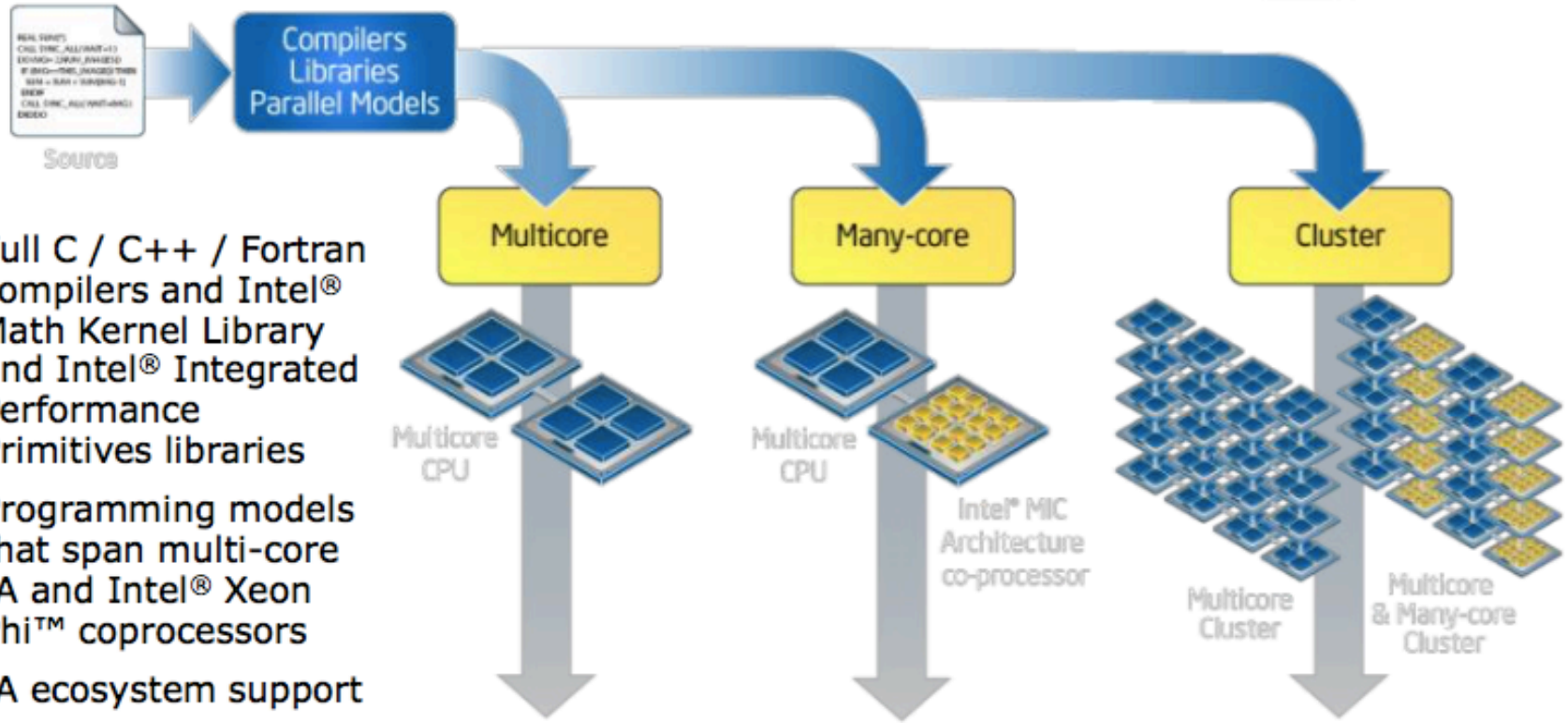
Faculty of Computer Science and Engineering

HCMC University of Technology

Ref

Using slides from Intel, CMU, Patrick Cozzi - University of Pennsylvania, Anthony Lippert

Architecting Scaling

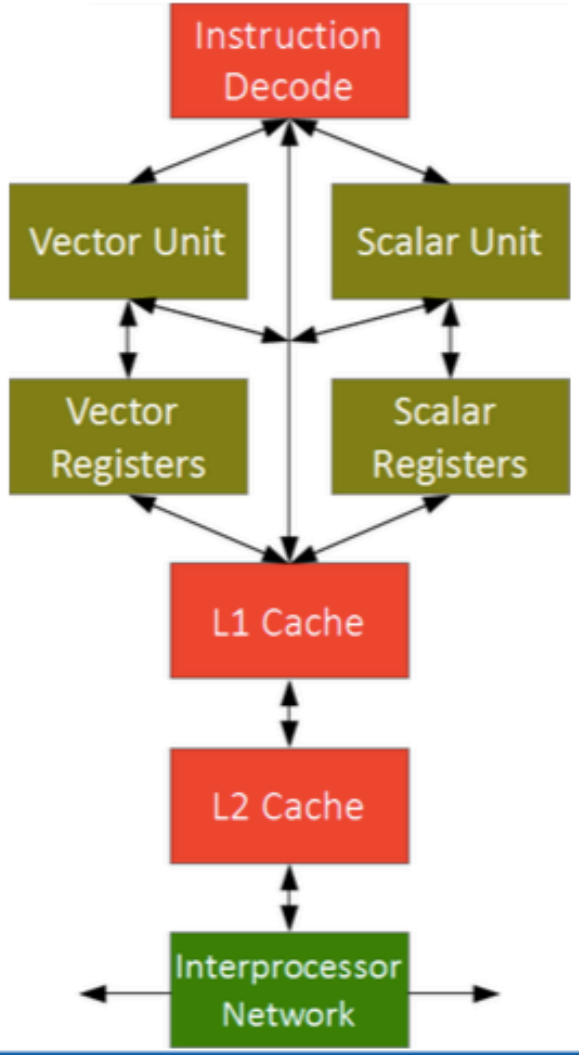


- Full C / C++ / Fortran compilers and Intel® Math Kernel Library and Intel® Integrated Performance Primitives libraries
- Programming models that span multi-core IA and Intel® Xeon Phi™ coprocessors
- IA ecosystem support

Eliminate Need for Dual Programming Architecture

Xeon Phi

Intel® Xeon Phi™ Coprocessor Core Architecture



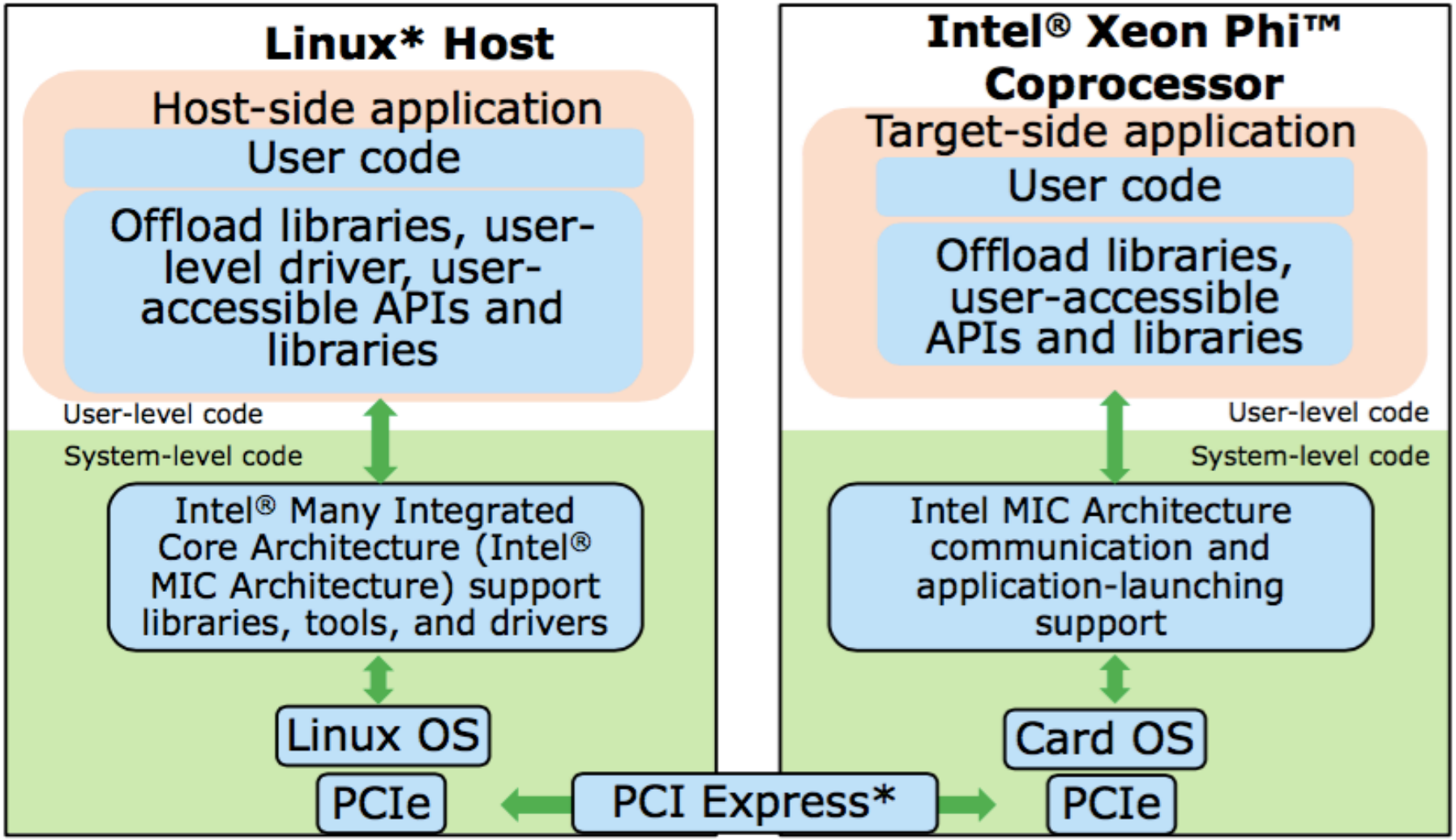
4 Threads per core, 64 bit, in order, specialized instructions

512 bit wide registers
Vector Processing Unit (VPU): integer, SP, DP; 3 operand.

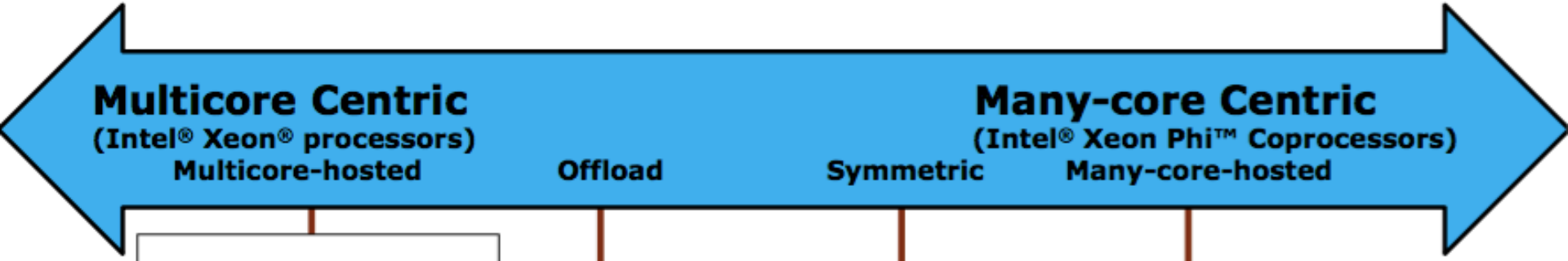
Fully coherent
L2 HW prefetching

Improved ring interconnect

Intel® Xeon Phi™ Coprocessor Software Architecture Overview



Spectrum of Programming & Execution Models



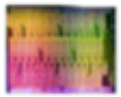
General purpose serial and parallel computing

Codes with highly-parallel phases

Codes with balanced needs

Highly-parallel codes

Multicore



Many-core

```
Main( )
Foo( )
MPI_*()
```

```
Main( )
Foo( )
MPI_*()
```

```
Main( )
Foo( )
MPI_*()
```

```
Foo( )
```

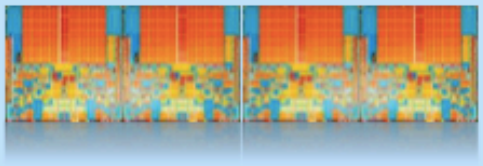
```
Main( )
Foo( )
MPI_*()
```

```
Main()
Foo( )
MPI_*()
```

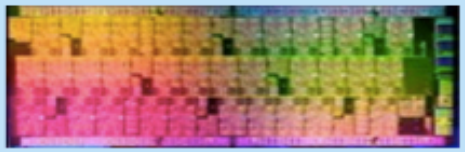
Range of Models to Meet Application Needs

Automatic offloading

Intel® MKL Automatic Offload Model



IA Host



Intel® Xeon Phi™ Coprocessor

User Application calls:

```
sgemm(&transa, &transb, &M,
    &N, &K, &alpha, A,&LDA, B,
    &LDB, &beta, C, &LDC);
```

MKL Library

Transport

MKL Worker + MKL Library

Transport

Xeon and Xeon Phi coprocessor parallelism and performance without changing code.

Use environment variables/functions to control work division & fine tune performance.

Intel® MKL Automatic Offload (AO) Overview

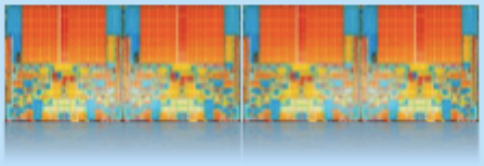
- **Key feature:** No changes to Intel® MKL use are required
  - Function calls and link line stay the same
  - **Speedup is transparent: if there is no Intel® Xeon Phi™ Coprocessor, Intel MKL runs as usual**
- AO-aware functions divide work and data between the host and Intel Xeon Phi Coprocessor(s) automatically
- Not everything can be accelerated: Only functions with sufficiently high flops / bytes ratio like SGEMM, DGEMM and matrix solvers

Enabling Intel® MKL Automatic Offload (AO) for Intel® MIC Architecture

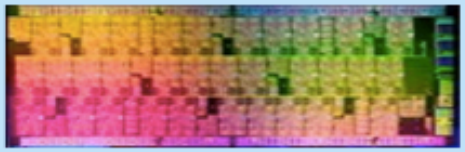
- Via environment variable MKL_MIC_ENABLE=1
- With MKL functions:
  - `int mkl_mic_enable()`
    enables automatic offload
    returns 0 if the operation was successful
    returns -1 if the operation failed
  - `int mkl_mic_get_device_count()`
    returns the number of Intel® Xeon Phi™ coprocessors in the system

Offloading by pragmas and keywords

User Controlled Offload Model



IA Host



Intel® Xeon Phi™ Coprocessor

User Application calls:
#pragma offload target(mic)
foo(a,b,c,d);

Worker
+ Libraries
(Intel® MKL, TBB….)

Transport

Transport

Xeon and Xeon Phi coprocessor parallelism and performance but needs (usually) small changes to existing code.

Additional libraries need native port to the Intel Xeon Phi coprocessor.

Using the Offload Compiler

- The programmer uses **#pragma offload target(mic)** to mark statements (offload constructs) that should execute on the Intel® Xeon Phi™ coprocessor.
- The offloaded region is defined as the offload construct plus the additional regions of code that run on the target as the result of function calls.
- Execution of the statements on the host will resume once the statements on the target have executed and the results are available on the host
- The in, out, and inout clauses specify the direction of data to be transferred between the host and the target.

Using the Offload Compiler – Explicit Memory Copy Model

Reduction code example :

ans = a[0] + a[1] + … + a[n-1]

C code to implement this version of the reduction :

```
float reduction(float *data, int size) {
float ret = 0.f;
for (int i=0; i<size; ++i){
ret += data[i];
}
return ret;
}
```

Serial Reduction with Offload :

```
float reduction(float *data, int size) {
 float ret = 0.f;
#pragma offload target(mic) in(data:length(size))
for (int i=0; i<size; ++i) {
ret += data[i];
}
return ret;
}
```

Vector Reduction with offload

- Each core on the Intel® Xeon Phi™ coprocessor has a VPU. The auto vectorization option is enabled by default on the offload compiler.
- Alternately, as seen in the example below, the programmer can use the Intel® Cilk™ Plus Extended Array Notation to maximize vectorization and take advantage of the Intel MIC Architecture core's 32 512-bit registers.
- The offloaded code is executed by a single thread on a single core.

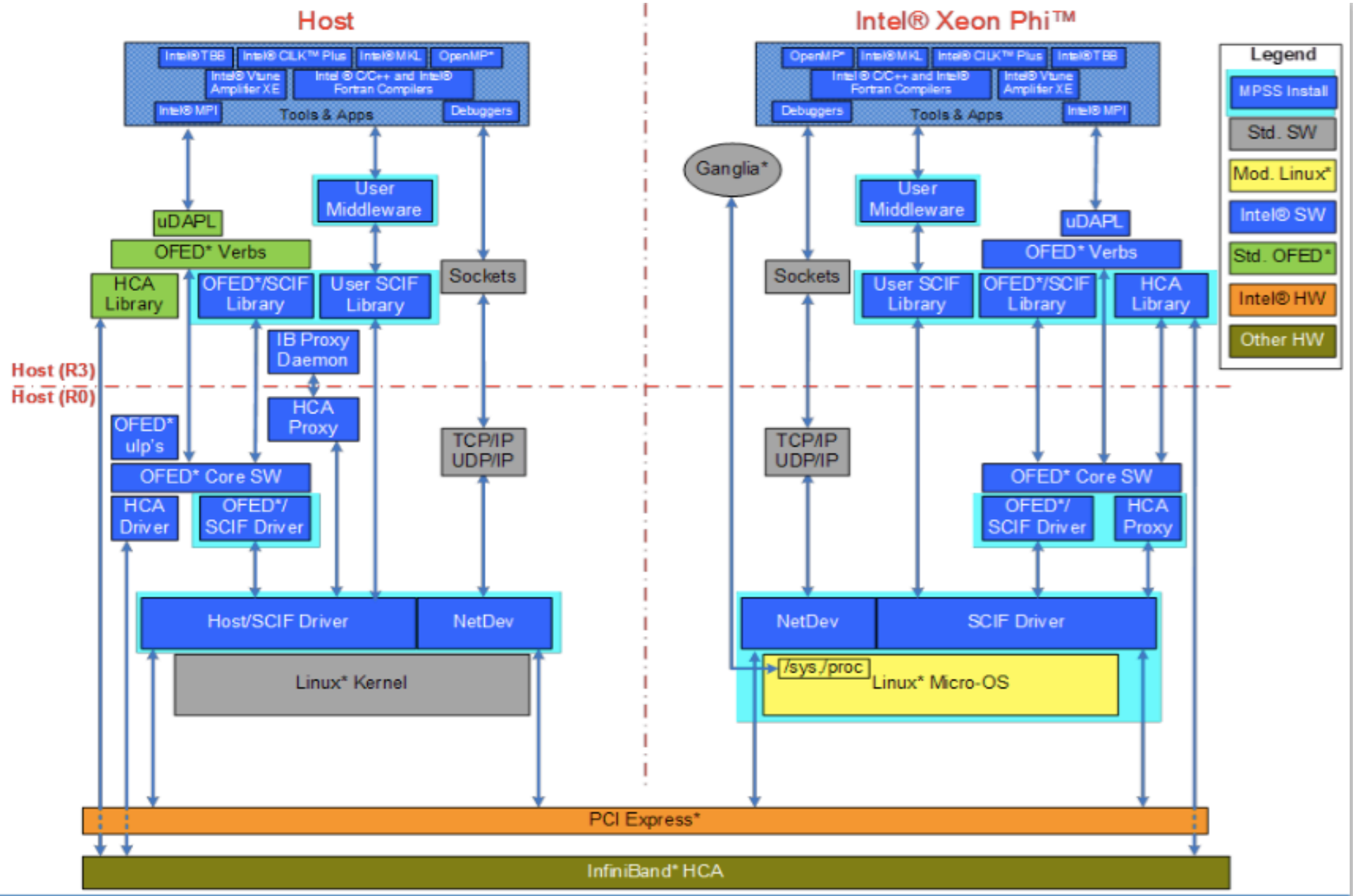
Vector Reduction with Offload :

```
float reduction(float *data, int size)
{
float ret = 0;
#pragma offload target(mic) in(data:length(size))
ret = __sec_reduce_add(data[0:size]); //Intel Cilk
Plus
//Extended Array Notation
return ret;
}
```

Native application

How Micro is the µOS of the Intel® Xeon Phi™ Coprocessor?

- Fully featured Linux* kernel derived form 2.6.34
- Busybox* toolkit
- Drivers for virtual Ethernet
- Ethernet Bridging possible
- nfs support
- MICdirect driver for InfiniBand* HCAs
- sep driver (event based sampling with Intel® VTune™ Amplifier XE performance profiler)
- ssh access



Running Native Applications

- Build application with "-mmic" switch:
  `icpc -mmic -o foo  foo.cpp`
- Upload binary and required libraries to the Intel® Xeon Phi™ coprocessor using scp:
  `scp foo `hostname`-mic0:`
- Execute binary:
  `ssh `hostname`-mic0 foo`
- Direct support for OpenMP*, Intel® Threading Building Blocks, Intel® Cilk™ Plus, Intel® Math Kernel Library, Intel® Integrated Performance Primitives, GNU libc, GNU stdc++

Cluster and MPI support

Cluster Support for the Intel® Xeon Phi™ Coprocessor

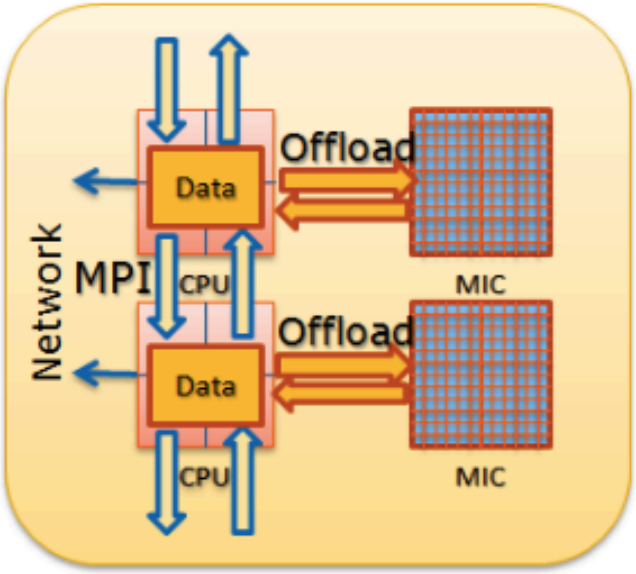
- Standard Linux* environment
- Ethernet bridging support
- Native NFS
- Basic tool chain to adapt SW stack on the µOS
- Yocto Project* initiative to provide completely configurable µOS
- Proof of concept for native Lustre* and Panasas* panfs file systems
- InfiniBand* support on selected HCA

MPI Support for the Intel® Xeon Phi™ Coprocessor

- MPI is the message passing standard for High Performance computing
- Intel is a leading vendor of MPI implementations and tools
- Optimized MPI application performance
  - Application-specific tuning
  - Automatic tuning
- Interconnect Independence and Runtime Selection
- Seamless interoperability with Intel® Trace Analyzer and Collector

Offload MPI Model

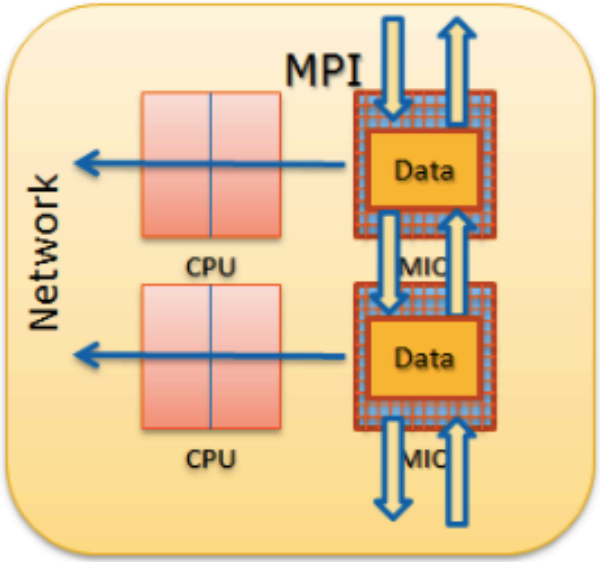
- MPI communication taking place only between the host processors
- The coprocessors are used exclusively through offload capabilities



- Use with Intel® C, C++, and Fortran Compiler for Intel® Many Integrated Core Architecture, Intel® Math Kernel Library, etc.
- Using MPI calls inside offloaded code are not supported

Coprocessor-only Model MPI Model

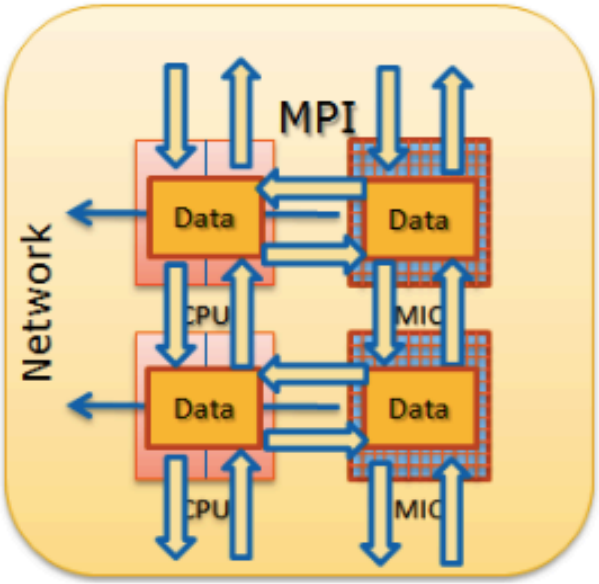
- MPI communication taking place only between the coprocessors
- The host processors are not used



- Supported with Intel® C, C++, and Fortran Compiler for Intel® Many Integrated Core Architecture, Intel® Math Kernel Library, etc.

Symmetric MPI Model

- Both the host CPUs and the coprocessors execute MPI processes and take part in MPI communication
- Message passing is supported between all partners using shared memory, TCP or InfiniBand*



- The best fabric is chosen automatically
- Use with Intel® C, C++, and Fortran Compiler Intel® Many Integrated Core Architecture, Intel® Math Kernel Library, etc.

Compiling a Symmetric MPI Program for the Intel® Xeon Phi™ Coprocessor

- Source the environment for the Intel® compiler and the Intel® MPI Library
- Use mpiicc to compile your MPI program for host.
- Use mpiicc to compile your MPI program for the Xeon Phi coprocessor adding the -mmic switch:

```
$ . /PATH_TO_INSTALL/bin/compilervars.sh intel64
$ . /PATH_TO_INSTALL/intel64/bin/mpivars.sh
$ mpiicc -o mpifoo.x86_64 mpifoo.c
$ mpiicc -mmic -o mpifoo.mic mpifoo.c
```

GPU

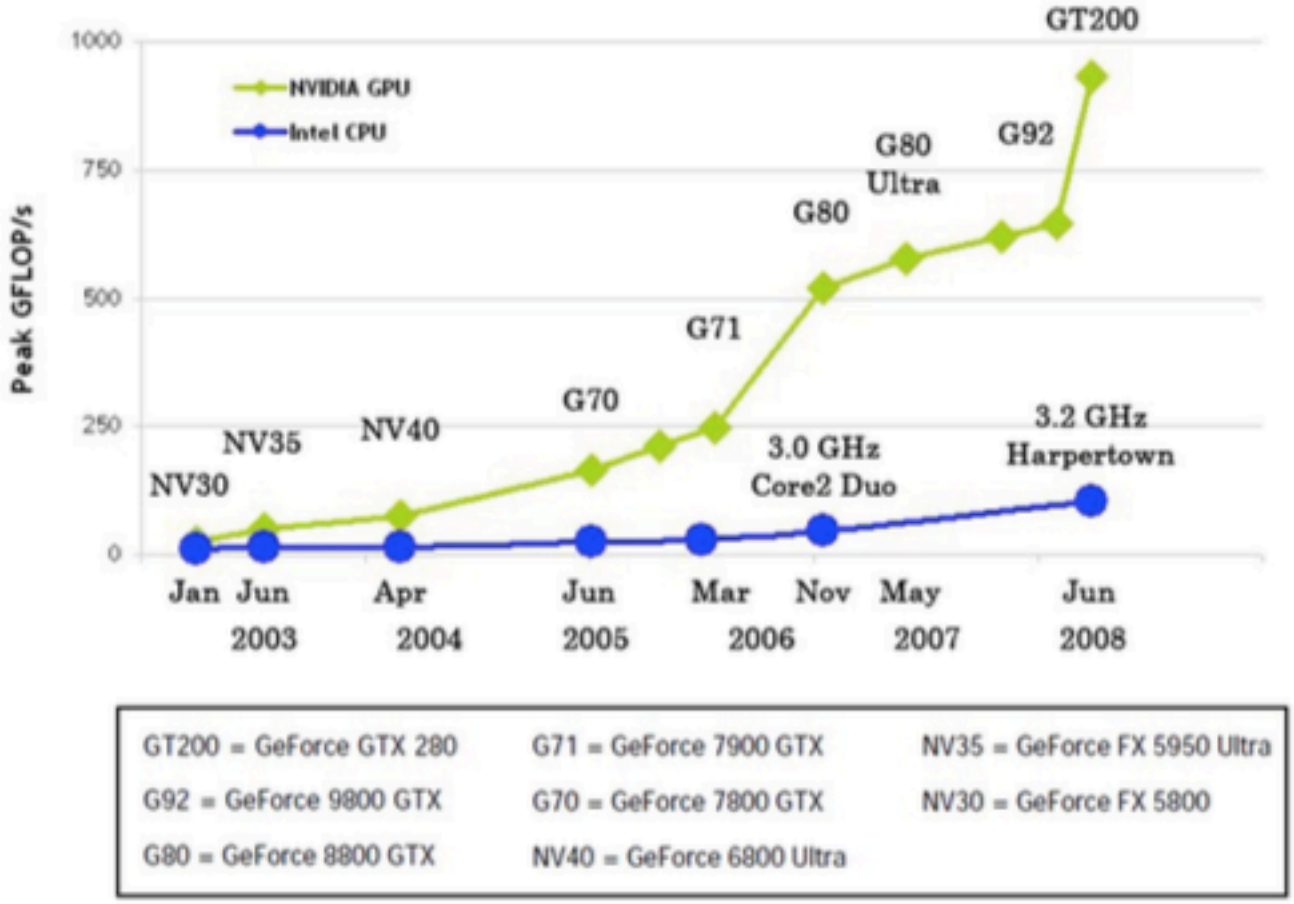
Intoduction

GPU: Graphics Processing Unit

- Hundreds of Cores
- Programmable
- Can be easily installed in most desktops
- Similar price to CPU
- GPU follows Moore's Law better than CPU

Introduction

Motivation:



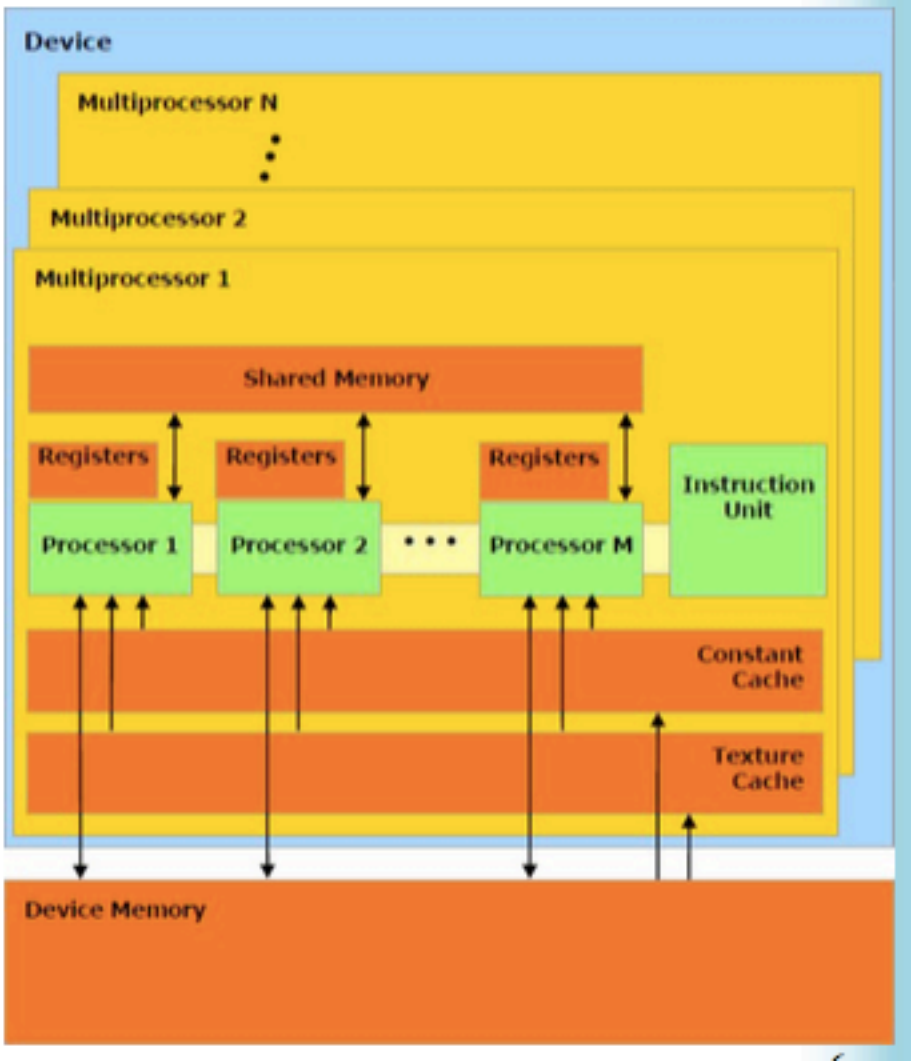
GPU Hardware

Multiprocessor Structure:

GPU Hardware

Multiprocessor Structure:

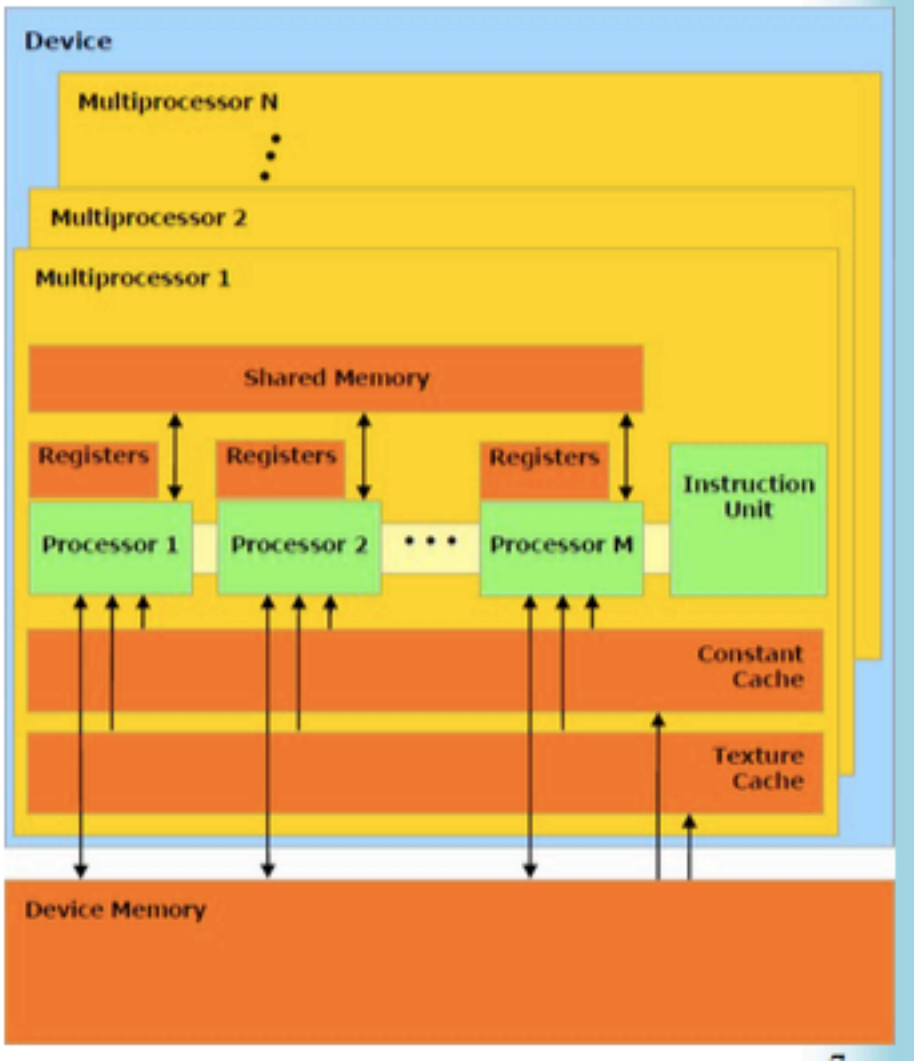
- N multiprocessors with M cores each
- SIMD – Cores share an Instruction Unit with other cores in a multiprocessor.
- Diverging threads may not execute in parallel.



GPU Hardware

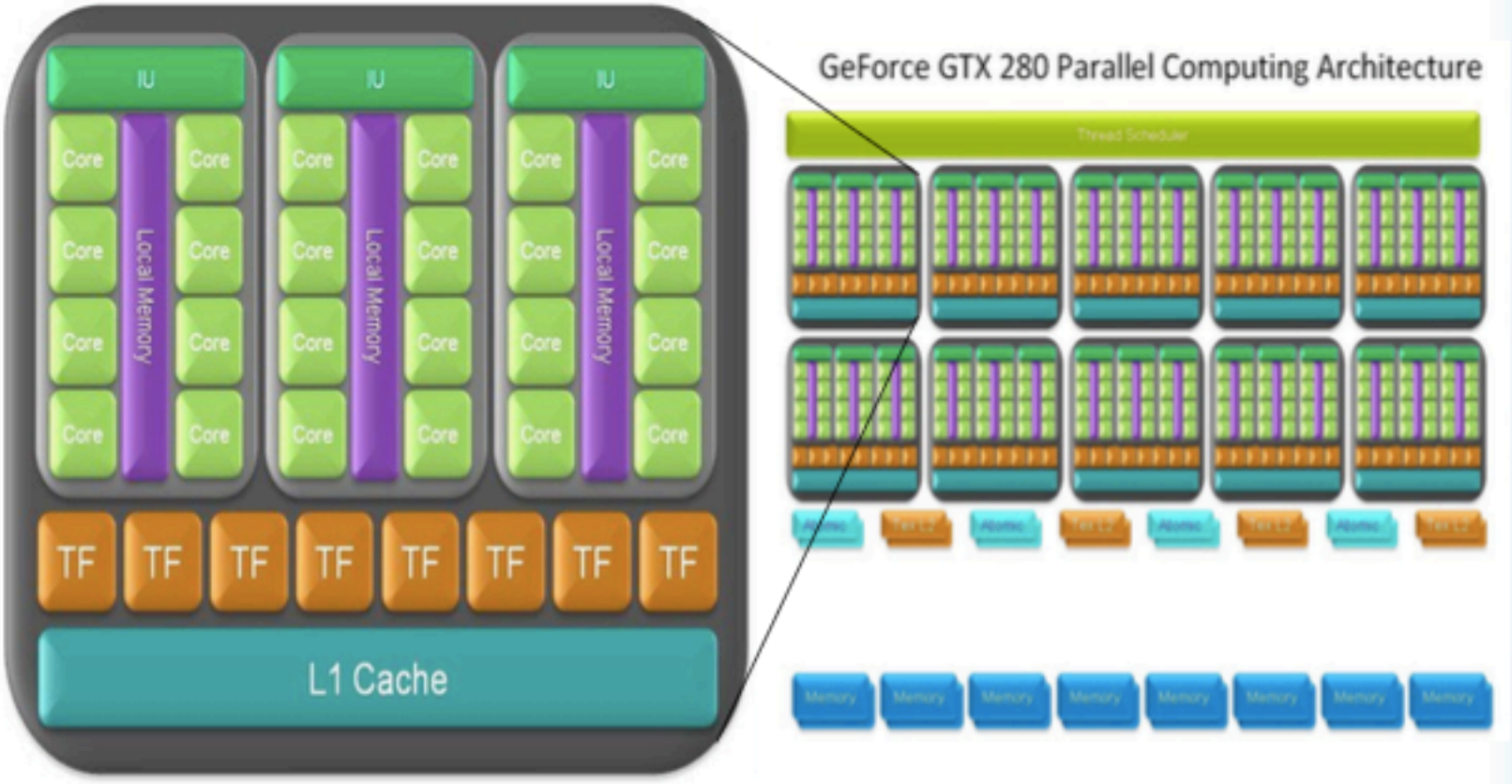
Memory Hierarchy:

- Processors have 32-bit registers
- Multiprocessors have shared memory, constant cache, and texture cache
- Constant/texture cache are read-only and have faster access than shared memory.



GPU Hardware

Thread Processing Cluster:



Programming Model

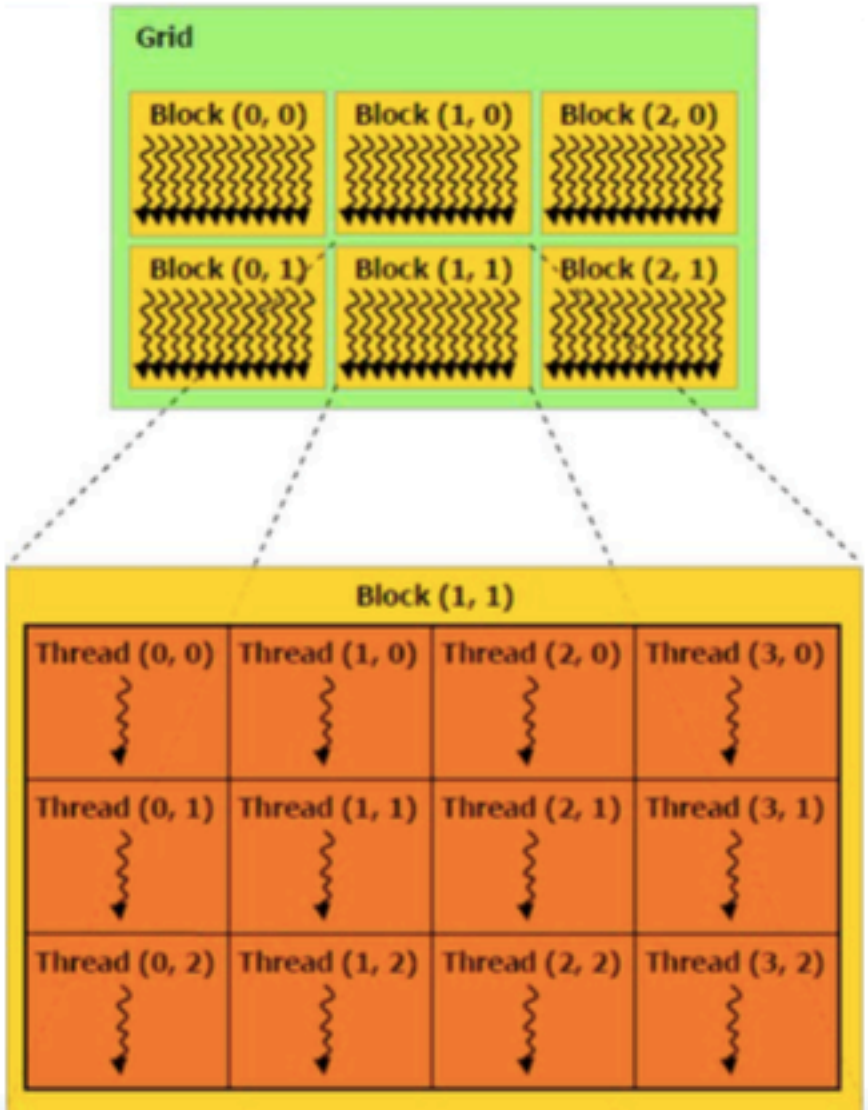
CUDA:

- Compute Unified Device Architecture
- Extension of the C language
- Used to control the device
- The programmer specifies CPU and GPU functions
  - The host code can be C++
  - Device code may only be C
- The programmer specifies thread layout

Programming Model

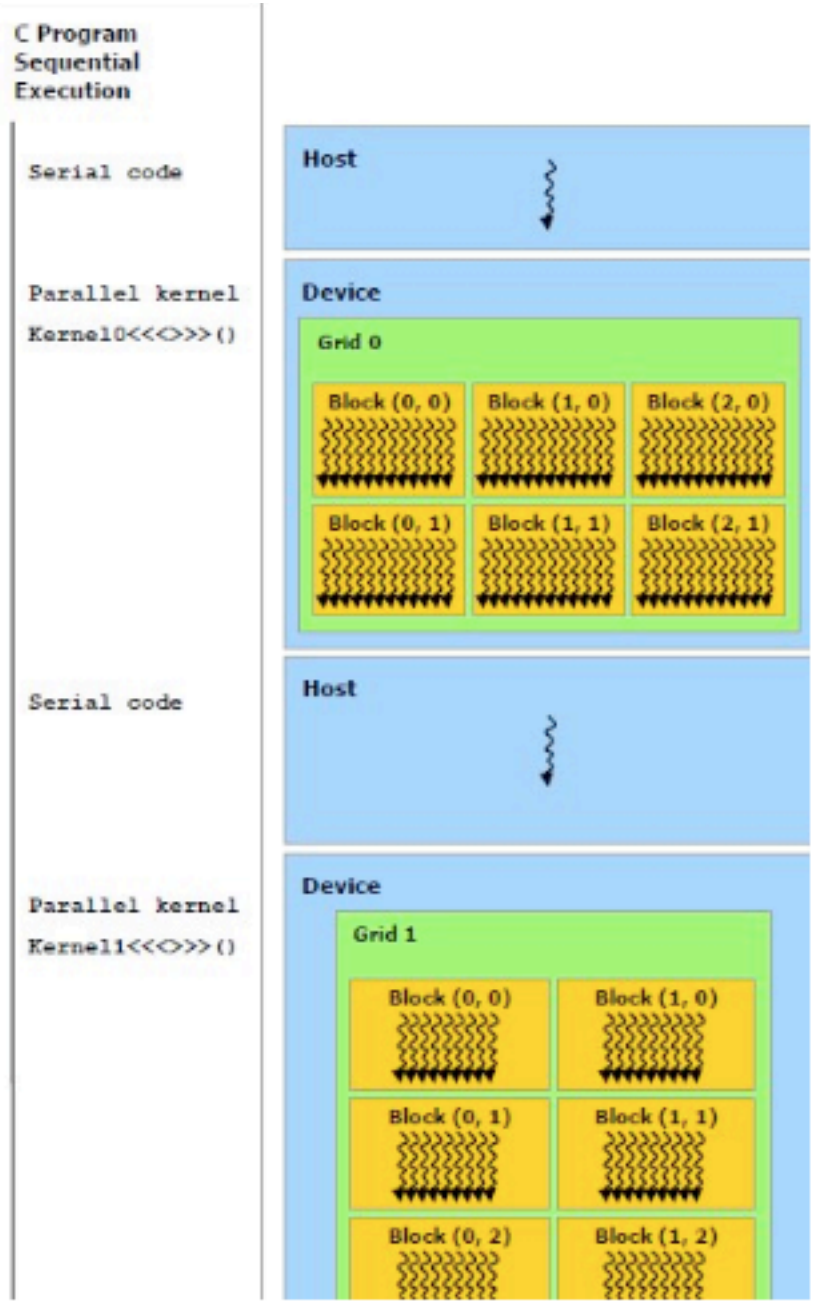
Thread Layout:

- Threads are organized into *blocks*.
- Blocks are organized into a *grid*.
- A multiprocessor executes one block at a time.
- A *warp* is the set of threads executed in parallel
- 32 threads in a warp



Programming Model

- Heterogeneous Computing:
  - GPU and CPU execute different types of code.
  - CPU runs the main program, sending tasks to the GPU in the form of kernel functions
  - Multiple kernel functions may be declared and called.
  - Only one kernel may be called at a time.



Algorithms

- SIMD
  - **S**ingle **I**nstruction **M**ultiple **D**ata
- Small memory
- Parallel Algorithms
  - Parallel Reduction
  - Scan (Naive and Work-Efficient)
  - Stream Compression
  - *S*ummed *A*rea *T*ables
  - Sparse Matrix Multiplication with Scan

Segmented scan

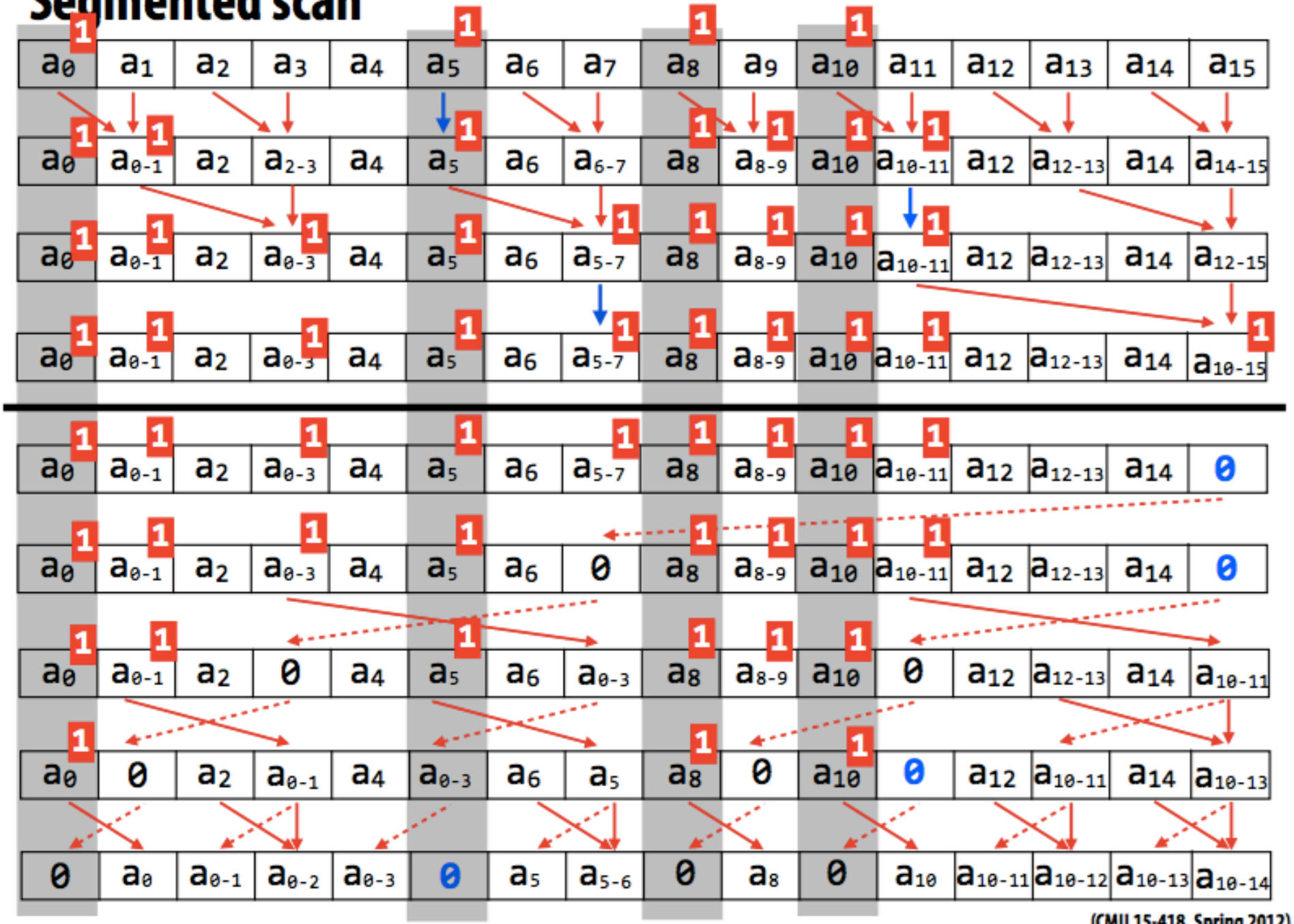
- Generalization of scan
- Simultaneously perform scans on arbitrary contiguous partitions of input collection

```
let a  = [[1,2],[6],[1,2,3,4]]
let ⊕ = +
segmented_scan_exclusive(⊕,a) = [[0,1], [0], [0,1,3,6]]
```

Assume simple "head-flag" representation:

```
a = [[1,2,3],[4,5,6,7,8]]
flag: 0 0 0 1 0 0 0 0
data: 1 2 3 4 5 6 7 8
```

Segmented scan



Sparse matrix multiplication example

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 3 & 0 & 1 & \cdots & 0 \\ 0 & 2 & 0 & \cdots & 0 \\ 0 & 0 & 4 & \cdots & 0 \\ & & \vdots & & \\ 0 & 2 & 6 & \cdots & 8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

- **Most values in matrix are zero**
  - Note: logical parallelization is across per-row dot products
  - But different amounts of work per row (complicates wide SIMD execution)
- **Example sparse storage format: compressed sparse row**

values = [[3,1], [2], [4], ..., [2,6,8]]

row_starts = [0, 2, 3, 4, ...]

cols = [[0,2], [1], [2], ...,]

Sparse matrix multiplication with scan

values = [[3,1], [2], [4], [2,6,8]]

cols = [[0,2], [1], [2], [1,2,3]]

row_starts = [0, 2, 3, 4]

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 6 & 8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

1. Map over all non-zeros: multiples[i] = values[i] * x[cols[i]]
   - multiples = [$3x_0$, x_2, $2x_1$, $4x_2$, $2x_1$, $6x_2$, $8x_3$]
2. Create flags vector from row_starts: flags = [1,0,1,1,0,0]
3. Inclusive segmented-scan on (multiples, flags) using addition operator
   - [$3x_0$, $3x_0+x_2$, $2x_1$, $4x_2$, $2x_1$, $2x_1+6x_2$, $2x_1+6x_2+8x_3$]
4. Take last element in each segment:
   - y = [$3x_0+x_2$, $2x_1$, $4x_2$, $2x_1+6x_2+8x_3$]