Reminder: Totient cluster structure

Consider:

- Each core has vector parallelism
- Each chip has six cores, shares memory with others
- Each box has two chips, shares memory
- Each box has two Xeon Phi accelerators
- Eight instructional nodes, communicate via Ethernet

Common layout (more nodes and better networks at high end)
Accelerator devices

- NVidia GPUs
- Intel Xeon Phi (aka MIC)
- AMD Radeon Pro
- Google Tensor Processing Units (TPUs)
- Arria (Intel) and Altera FPGAs
- Lake Crest, Knights Mill, etc?
If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?

— Seymour Cray

- Host computer
  - General purpose
  - Usually multi-core
- Accelerator
  - Specialized for particular workloads
  - Often many specialized cores (many-core)
  - May have a non-x86 ISA, needs different compilers
  - More “exotic” HW support (half precision, wide vecs, etc)
  - Often has *independent memory*
Some historical perspective

• 1970s – early 1990s: vector supercomputers
• But games pay more than science!
  • Mid-late 90s: SIMD vectors in CPUs (for graphics)
  • Also 90s: Special-purpose GPUs
  • Early 2000s: Programmable GPUs, rise of GPGPU
• And the pendulum swings
  • 2007: NVidia Tesla + first version of NVidia CUDA
  • 2010: Knight’s Ferry
  • 2012-13: Knight’s Landing (first commercial Xeon Phi)
  • Today: mostly NVidia, Intel trailing, AMD a ways back
    • NB: Knight’s Landing Xeon Phi can operate independently!
• More recent accelerators target deep learning
Accelerator options

- NVidia GPUs
  - Amazon EC2, Google GCE, MS Azure
  - Summit (ORNL)
  - Sierra (LLNL)

- Intel Xeon Phi
  - Totient cluster!
  - Tianhe-2
  - TACC Stampede
  - Aurora (Argonne)
Same old song...

- For performance, we need:
  - Stern warnings against magical thinking
  - Enough about HW to reason about performance gotchas
  - Careful attention to memory issues
  - Pointers to appropriate programming models

- What’s different?
  - Many more cores/threads
  - Lots of data parallelism
  - New? NVidia lore harkens to Cray vector days!
• Call a library!
  • This is often the fastest way to faster code
  • Remember trying to beat BLAS in P1?
• CUDA (NVidia only)
• OpenCL (clunkier, works with more)
• OpenACC
• OpenMP
• Novel languages (Simit, Julia, ...)

Programming models
Xeon Phi 5110P

- Came out late 2012 (now end-of-life)
- 60 cores (modified Pentium)
- 4 way hyperthreading / 240 hardware threads
- AVX512 support (wide vector units)
- Base frequency of 1.05 GHz
- Ring network on chip
- 32K L1 data/instruction cache per core
- 30 MB L2 (512K/core) and 8 GB RAM

Program with OpenMP + directives, OpenCL, Cilk/Cilk+, libraries
Phi programming

- Knight’s Landing – maybe just ssh in
- Offload mode slides adapted from TACC talk
Supposing \texttt{foo} uses BLAS for performance:

1. \# In Makefile
   \texttt{icc -qopenmp -mkl foo.c -o foo.x}

2. \# In PBS script
   \texttt{export MKL_MIC_ENABLE=1}
   \texttt{export OMP_NUM_THREADS=12}
   \texttt{export MIC_OMP_NUM_THREADS=240}
   \texttt{./foo.x}

Actually divides work across host and MIC
```c
#include <stdio.h>
#include <omp.h>

int main()
{
    int nprocs;

    #pragma offload target(mic)
    nprocs = omp_get_num_procs(); // On MIC

    printf("nprocs = %d\n", nprocs); // On host
    return 0;
}

Can have OpenMP on either host or MIC.
```
```c
#include <stdio.h>
#include <omp.h>

int main()
{
    int nprocs;

    #pragma offload target(mic:0)
    nprocs = omp_get_num_procs(); // On MIC 0 (vs MIC 1)

    printf("nprocs = %d\n", nprocs); // On host
    return 0;
}
```
Always generate host code, generate code for MIC in

- **offload** regions
- Functions marked with `__declspec(target(mic))`

Can also mark global variables with `__declspec(target(mic))`
Offload off-stage

Execution behind the scenes:

- Detect MICs
- Allocate/associate MIC memory
- Transfer data to MIC
- Execute on MIC
- Transfer data from MIC
- Deallocate on MIC

Can control with clauses
• **in, out, inout** clauses: declare how variables transfer
• **alloc_if, free_if**: manage allocation for associated dynamic arrays on host and accelerator
__declspec(target(mic))
void something_fancy(int n, double* x) {...}

int main()
{
    int n = 100;
    double* x = (double*) memalign(64, n*sizeof(double));
    #pragma offload mic \
    inout(x : length(n) alloc_if(1) free_if(1))
    something_fancy(n, x);
    // Do something with x on host
    free(x);
    return 0;
}
int n = 123;
#pragma offload target(mic) signal(&n)
act_very_slowly();
do_something_on_host();
#pragma mic offload_wait target(mic) wait(&n)
Desiderata

- Lots of parallel work
  - Vectorized, OpenMP, etc – both host and MIC
- Not too much data transfer
  - It’s expensive!
  - Re-use data transfers to MIC if possible

Writing “modern” code tends to be good for both sides...
What about GPUs?

Lots of good references out there:

- Programming Massively Parallel Processors (Kirk and Hwu)
  - available online via Cornell library subscription (Safari)
- CUDA C Programming Guide
- CUDA C Best Practices Guide
- Oxford CUDA short course

Lots of details! But basic ideas constant: regular computation, expose parallelism, exploit locality, minimize memory traffic.
Basic architecture (NVidia GPUs)

- Array of Streaming Multiprocessors (SMs)
- Single Instruction Multiple Thread (SIMT)
  - Operate with warp of 32 threads
  - Each thread execs same instructions at once
  - Some may be inactive (for conditional exec)
- Exec a warp at a time (want lots of parallel work!)
- Organize threads into logical grids of blocks
- Several types of device memory
How to program?

Call a library!

• MAGMA for doing NLA
• cuBLAS, cuFFT, etc otherwise

But sometimes you need a little lower level.
NVidia CUDA

Compute Unified Device Architecture. Three basic ideas:

- Hierarchy of thread groups
- Shared memories
- Barrier synchronization
Idea:

- Define *kernel* that runs on GPU
- Exec kernel on $N$ parallel threads
- Different work according to thread index
__global__
void vecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // ...
    // Execute with N threads
    vecAdd<<1,N>>(A, B, C);
    // ...
}
• Declare **__global__** to run on device
  • **__device__** for call/exec on device
  • **__host__** for all on host (or don’t annotate)

• Call is `kernel<<nBlk,nThread>>(args)`

• Blocks/threads in 1-3D logical index spaces
  • Threads form blocks, blocks form grids
  • IDs are `blockIdx` and `threadIdx` structs
  • `gridDim` gives blocks/grid
  • `blockDim` gives threads/block
  • Each struct has `x, y, z` fields
  • Under the hood: 1D space
  • At most 1024 threads per block
__global__

void vecAdd(float* A, float* B, float* C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}

int main()
{
    // ...
    // Execute with N threads
    vecAdd<<1,N>>(A, B, C, N);
    // ...
}
Explicitly manage device data and transfers:

```c
1    cudaMemcpy((void**)&d_A, size);
2    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
3    // Do something on device
4    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
5    cudaFree(d_A);
```

... and we have to malloc/free corresponding device data.
Device has several types of memory

- Per-thread: registers, local memory
- Per-block: shared memory (__shared__)  
- Per-grid: global memory, constant memory

Synchronize access to shared/global memory with __syncthreads() (barrier)
And so forth

- Other memory types (texture, surface)
- Asynchronous execution
- Streams and events
- ... and the programming guide is 300 pages!
So now what?

So far we have seen

• Two accelerator HW platforms
• Two programming models
• Same old concerns with lots of new details

Should be asking

• Is there a better way than low-level mucking about?
• What if I want to use this in a larger code?

Both great questions! Let’s pick them up next time.