ACM/CS 114
Parallel algorithms for scientific applications

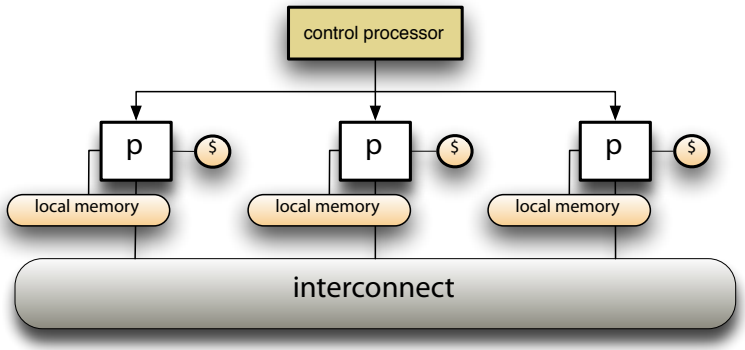
Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

Hybrid architectures

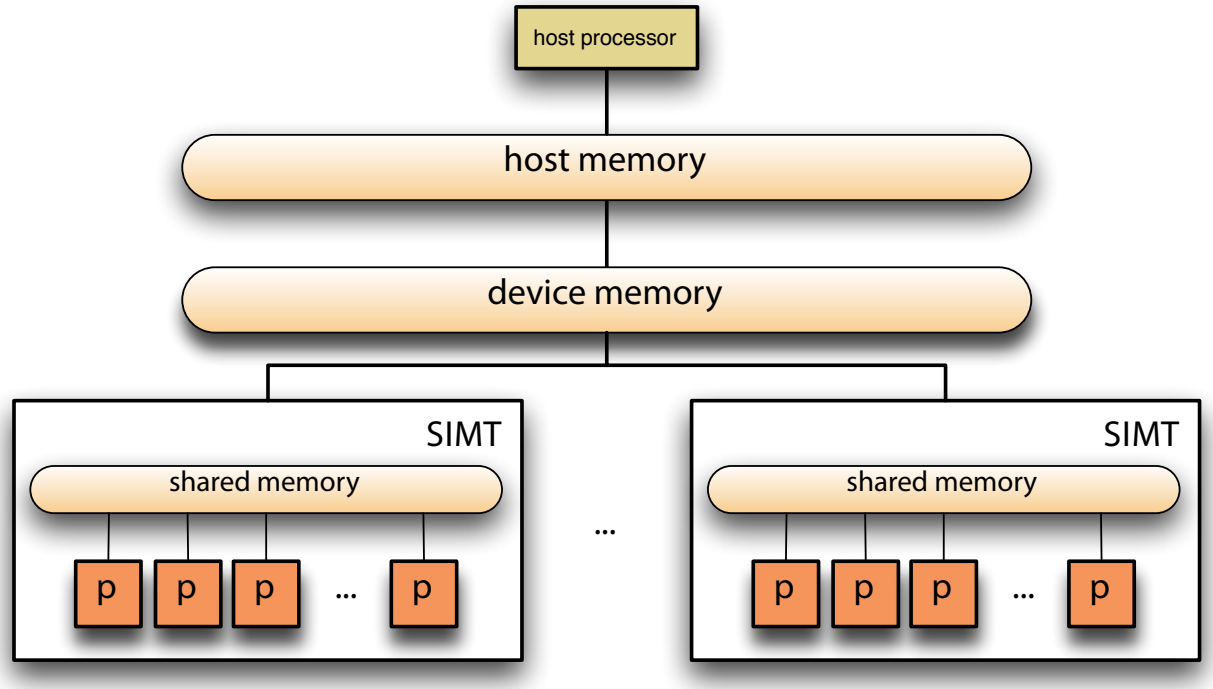
- recall the layout of SIMD machines
  - a large number of small, special purpose processors
  - a single “controller” manages the instruction stream
    - each processor executes the same instruction on its local data
    - may be able to specify which processors are active/idle



- modern hybrid systems based on GPUs have somewhat more elaborate architectures
  - multi-tier memory layouts
  - large core counts per board
  - elaborate access rules that enable the hardware to be very fast
  - recent ones finally support double precision floating point arithmetic
  - broadly available – it’s in your graphics card, thanks to video gaming

nVidia GPU architecture

- The GPU boards are hosted by a conventional processor
  - connected through PCI Express, which is the limiting factor in moving data to and from the device memory
- computing power and memory configurations vary from
  - a single 2-core GPU with 128M of memory on older video cards
  - to 30 $\times$ 8-core SIMTs with 4G of memory on the Tesla C1060
    - 240 cores; peak: 933 Gflops single precision, 78 Gflops double precision
    - `rivulet.cacr.caltech.edu` has four such boards



Getting started

- getting the drivers, tools and code samples
  - visit `http://nvidia.com/cuda`
- compiling and linking
  - `C` for CUDA: a few extensions
  - source is a mixture of the code that runs on the host and the *kernels* that run on the `GPU`
  - there are restrictions on what kind of code you can include in a kernel
  - `nvcc` is the special compiler and linker
- staging and launching
  - the resulting executable runs on the host
  - launches threads on the GPU through special statements
- the emulator
  - the SDK comes with a software emulator
  - extremely useful for debugging
- special hardware
  - your video card
  - Tesla boards

Sanity check

```
1  // memxchng.cu: making sure the compiler and cuda runtime are accessible
2  #include <cuda.h>
3  #include <assert.h>
4
5  int main(int argc, char* argv[]) {
6     const int N = 12;
7     // allocate some buffers on the host
8     float *send_host = (float *) malloc(N*sizeof(float));
9     float *recv_host = (float *) malloc(N*sizeof(float));
10    // allocate matching ones on the device
11    float *send_device, *recv_device;
12    cudaMalloc((void **) &recv_device, N*sizeof(float));
13    cudaMalloc((void **) &send_device, N*sizeof(float));
14    // and initialize the host data
15    for (int i=0; i<N; i++) {
16       send_host[i] = 2.0f + i*i;
17       recv_host[i] = 0.0f;
18    }
19    // send the data from the host to the device
20    cudaMemcpy(recv_device, send_host, N*sizeof(float), cudaMemcpyHostToDevice);
21    // move the data in device memory
22    cudaMemcpy(send_device, recv_device, N*sizeof(float), cudaMemcpyDeviceToDevice);
23    // get it back on the host
24    cudaMemcpy(recv_host, send_device, N*sizeof(float), cudaMemcpyDeviceToHost);
25    // check the result
26    for (int i=0; i<N; i++) {
27      assert(send_host[i] == recv_host[i]);
28    }
29    // free the buffers;
30    free(send_host); free(recv_host);
31    cudaFree(send_device); cudaFree(recv_device);
32
33    return 0;
34 }
```

The execution model

- `nvcc` splits the source code into two parts
  - the code that runs on the host
  - the device *kernel*, the code that runs of the GPU
    - built out of specially marked subroutines in the program
- the program is launched on the host and runs sequentially
- at specific points in the code, the program
  - launches the kernel and runs it on the GPU by many threads in parallel
  - the host continues on without blocking
  - until it encounters some blocking call to the CUDA runtime
- the execution context is specified by organizing
  - groups of threads in *blocks*
  - groups of blocks in *grids*
  - blocks are scheduled and executed in arbitrary order
    - in *warps*: 32 SIMD threads at a time (on currently available devices)
- at runtime, each thread is given
  - `threadIdx`: its own thread id
  - `blockIdx`: the id of the block of active threads
  - `blockDim`: the geometry of the block of active threads

Adding a bit of work

```
1  // scale.cu: multiply each element in an array by a given float
2  #include <cuda.h>
3  #include <assert.h>
4
5  // manipulate the host array
6  void scale_host(float* a, float scale, int N) {
7     // loop over all array elements and multiply them by 2
8     for (int i=0; i<N; i++) {
9        a[i] *= scale;
10    }
11    return;
12 }
13
14 // and here is the corresponding code for the GPU
15 __global__ void scale_device(float* a, float scale, int N) {
16    // this thread is responsible for one element of the array
17    // compute its offset using the block geometry builtins
18    int idx = blockIdx.x * blockDim.x + threadIdx.x;
19    // make sure we don't go past the last one
20    if (idx < N) {
21       // do the arithmetic
22       a[idx] *= scale;
23    }
24    return;
25 }
```

Launching the kernel

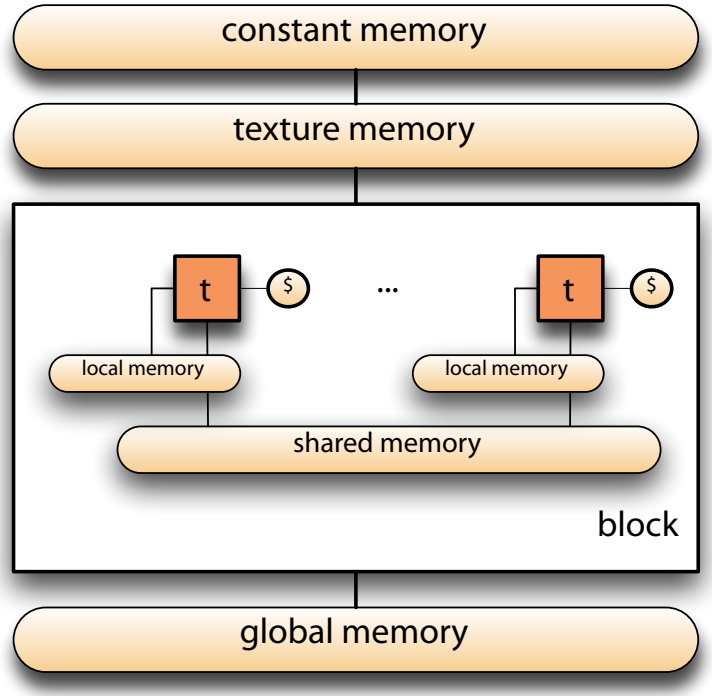
```
1   . . .
2   // send the data from the host to the device
3   cudaMemcpy(
4      array_dev, send_host, N*sizeof(float), cudaMemcpyHostToDevice);
5
6   // set up the device execution context for our threads
7   // each thread will take care of one element
8   int blockSz = 4; // 4 threads per block
9   // compute the number of blocks needed
10  int nBlocks = N/blockSz;
11  // adjust up to make sure we cover the entire array
12  if (N % nBlocks) {
13     nBlocks++;
14  }
15  // scale the array on the device
16  float scale = 2.0f;
17  scale_device <<<nBlocks, blockSz>>> (array_dev, scale, N);
18  // scale the input array on the host
19  scale_host(send_host, scale, N);
20
21  // get it back on the host
22  cudaMemcpy(
23     recv_host, array_dev, N*sizeof(float), cudaMemcpyDeviceToHost);
24  . . .
```

Capabilities of the Tesla C1060 board

```
Device 1: "Tesla C1060"
  CUDA Driver Version:                           2.30
  CUDA Runtime Version:                          2.30
  CUDA Capability Major revision number: 1
  CUDA Capability Minor revision number: 3
  Total amount of global memory:     4294705152 bytes
  Number of multiprocessors:         30
  Number of cores:                   240
  Total amount of constant memory:   65536 bytes
  Total amount of shared memory per block: 16384 bytes
  Total number of registers available per block: 16384
  Warp size:                         32
  Maximum number of threads per block: 512
  Maximum sizes of each dimension of a block: 512 x 512 x 64
  Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
  Maximum memory pitch:              262144 bytes
  Texture alignment:                 256 bytes
  Clock rate:                        1.30 GHz
  Concurrent copy and execution:     Yes
  Run time limit on kernels:         No
  Integrated:                        No
  Support host page-locked memory mapping: Yes
  Compute mode:                      Default
                                     (multiple host threads can
                                     use this device simultaneously)
```

Memory hierarchy

- each thread gets
  - a set of *registers*: really fast but limited memory
  - memory *shared* by all threads in its block
  - private *local* memory allocated from the global address space
  - access to the device *global* memory pool
- *texture* and *constant* memory are beyond scope here
  - not really useful for general purpose programming
- latency and bandwidth for these memory pools are very different
  - the name of the game: resource management
  - the price to pay for the astronomical performance



Summary

- GPUs brought hybrid programming models back to center stage
  - but our parallelization steps don't change
  - just the balance between fine and coarse grain tasks
- high performance computing the way it used to be
  - resource allocation and management strategies define performance
  - true enough for sequential, `MPI` and threaded programs anyway
- barely scratched the surface here
  - let me know if you are interested in pursuing further
- if you must program in a hybrid model
  - why not write multi-threaded host programs
    - to take advantage of more devices per host
    - to overlap calculations on the host and GPU
  - why not use `MPI` as well and scale out to multiple nodes
    - for some really massive calculations!