Graphical Processing Units

Main vendors NVIDIA, AMD (2003 - )

Architectural features:
• Many simple processing elements (16-5140)
• 1000s – 1 000 000s of threads
• Hardware thread scheduling (1 cycle)
• Focus on throughput (data parallel tasks)
• Limited memory (small on chip mem)
• Limited bandwidth to host mem (bottleneck)
CPU vs GPU

- Big caches, hierarchy
- Branch predictors
- Out-of-order
- Multiple-issue
- Speculative execution
- Double-precision

Reduce mem latency with caches and hide with other instructions

- None or small caches
- 1000’s of threads
- 1 cycle context switch
- SIMT instructions (32 threads)
- Single precision

Hide mem latency with work from other threads

CUDA (Compute Unified Device Architecture)

- C programming language on GPUs
- Designed and developed by NVIDIA
- Support only on NVIDIA GPUs
- Requires no knowledge of graphics APIs
- Access to native instructions and memory
- Easy to get started and get good performance
- Stable, documented, supported and free
- For Windows, Linux and MacOS
- More transparent than OpenCL
- Today, CUDA is dominating GPU programming
Software model:

Data parallel work flow with multiple levels of parallelism:

- **Grid**, maps to application data (e.g. Matrix), consists of independent blocks

- **Blocks**, consists of parallel independent threads

- **Threads**, each thread executes the kernel on a data element (data parallel)

Blocks of threads execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.
Program flow:

1. Allocate memory for device (GPU)
2. While "something to do"
   1. Transfer data from host to device
   2. Launch computational kernel on device
   3. Transfer data from device to host
3. Deallocate memory on device

CUDA memory model:
CUDA memory model:

<table>
<thead>
<tr>
<th>Summary</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Per Thread</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Local Memory</td>
<td>Per Thread</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>Per Block</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Global Memory</td>
<td>Per Grid</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>Per Grid</td>
<td>Read-Only</td>
</tr>
<tr>
<td>Texture Memory</td>
<td>Per Grid</td>
<td>Read-Only</td>
</tr>
</tbody>
</table>

- Registers and local memory used for thread local variables.
- Shared and global memory used for allocating data (arrays).
- Constant and texture memory are read-only (constants) and can only be updated by host.

**Note:** You do not need to use all of these to get started, use global memory and optimize later.
CUDA memory model:

**Important:** For best performance memory accesses (read/write) to *global memory* must be contiguous, i.e., residing threads reads/writes residing data addresses. (Not important for *shared memory*)

Contiguous accesses, good performance

Non-contiguous accesses, bad performance
CUDA C-language extensions:

- Function type qualifiers
  - Specify where to call a GPU kernel
    - `__device__ func` (callable on device)
    - `__global__ func` (callable on host)

- Variable type qualifiers for GPU memory
  - `__device__ var` (global memory)
  - `__constant__ var` (constant memory)
  - `__shared__ var` (shared memory)

- Function execution directive
  - `Funcname<<<GridDim, BlockDim>>>(args)`

- Build-in type for GridDim and BlockDim
  - `dim3`: 1, 2 or 3 dimensional grids/blocks
  - Index: `Grid->blockId, Block->threadId`

Example vector addition

**CPU function:**
```c
void vec_add(float *x, const float *y, int N) {
    for (int i=0; i<N, i++)
        x[i]=x[i]+y[i];
}
vec_add(x,y,N);    // call function on CPU
```

**GPU function:**
```c
__global__ void vec_add(float *x, const float *y, int N) {
    int i=threadIdx.x;
    x[i]=x[i]+y[i];
}
vec_add<<<1,N>>>(x,y,N);   // call function on GPU
one thread per element
```
Memory allocation and transfers:

- `cudaMalloc( (void**)&pointer, size)`
  - Allocates global memory on device
- `cudaFree(pointer)`
- `cudaMemcpy(dest, src, transfer)`
  - `transfer = cudaMemcpyHostToDevice`
  - `transfer = cudaMemcpyDeviceToHost`

( More memory options available: page-locked, portable, write-combining and mapped memory )

The NVCC compiler:

- Source files compiled with CUDA compiler `nvcc`
- CUDA programs/kernels stored in files with ending `.cu`
- `nvcc` uses host compiler (gcc) to compile CPU code

Program execution:

- Run as any C-program, > ./a.out
Complete program, ex1.cu:

```c
#include <stdio.h>
#include <cuda.h>

__global__ void first_kernel(float *a)
{
    int thr_id = threadIdx.x + blockDim.x*blockIdx.x;
    x[thr_id] = (float) threadIdx.x;
}

int main(int argc, char **argv)
{
    float *x, *dev_x;
    int nblocks=2, nthreads=8, nsize=2*8;

    x=(float *)malloc(nsize*sizeof(float));
    cudaMalloc((void**)&dev_x,nsize*sizeof(float));

    first_kernel<<<nblocks,nthreads>>>(dev_x);
    cudaMemcpy(x,dev_x,nsize*sizeof(float),
                cudaMemcpyDeviceToHost);

    for (int n=0; n<nsize; n++)
        printf("n, x = %d %f \n",n,x[n]);

    cudaFree(dev_x);
    free(x);
}
```

First example, ex1.cu:

```
>nvcc ex1.cu
>./a.out

n,  x  =  0  0.000000
n,  x  =  1  1.000000
n,  x  =  2  2.000000
n,  x  =  3  3.000000
n,  x  =  4  4.000000
n,  x  =  5  5.000000
n,  x  =  6  6.000000
n,  x  =  7  7.000000
n,  x  =  8  0.000000
n,  x  =  9  1.000000
n,  x  = 10  2.000000
n,  x  = 11  3.000000
n,  x  = 12  4.000000
n,  x  = 13  5.000000
n,  x  = 14  6.000000
n,  x  = 15  7.000000
```
Example, MatAdd.cu:

// Host code
int main() {
    int N=1024;
    int blocksize=16;
    int size=N*N*sizeof(float);

    // Host arrays
    float *a = (float *)malloc(size);
    float *b = (float *)malloc(size);
    float *c = (float *)malloc(size);
    for ( int j = 0; j < N; j++ )
        for ( int i = 0; i < N; i++ )
            { a[i+j*N] = 1.0f; b[i+j*N] = 3.5f; }

    // Create 2D blocks and grid (maps to matrix, 1 thread/element)
    dim3 dimBlock( blocksize, blocksize );
    dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );

    // Device (global memory) arrays
    float *ad, *bd, *cd;
    cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );
    matrix_add<<<dimGrid, dimBlock>>>( ad, bd, cd, N );
    cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

    // use results to something on host
    cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
    free(a); free(b); free(c);
}

// device code (each thread updates one matrix element)
__global__ void matrix_add(float *a, float *b, float *c, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N; // Note: contiguous order (row-wise)
    if ( i < N && j < N )
        c[index] = a[index] + b[index];
}
Example, MatMul.cu:

```c
// Host code
int main() {

  ... (Same as above, MatAdd.cu)

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice);
  matrix_mul<<<dimGrid, dimBlock>>>( ad, bd, cd, N );
  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost);

  ...
}

// Device code
__global__ void matrix_mul(float *a, float *b, float *c, int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  float s=0.0f

  if ( i < N && j < N ) {
    for (int k=0; k<N; k++)
      s += a[i+k*N] * b[k+j*N];    // Note: Non-contiguous
    c[i+j*N]=s;
  }
}
```

Block Matrix Multiplication

\[ C_{ij} = \sum_k A_{ik} x B_{kj} \] where \( A_{ik}, B_{kj}, C_{ij} \) are blocks and \( A_{ik} x B_{kj} \) is a matrix-matrix multiplication.

Loop over the blocks \( (i \text{ and } j) \) and do matrix-matrix multiplication block-wise.
(6 nested loops for traditional CPU code)
Block Matrix Multiplication on GPU

Note:
• CUDA grid corresponds to matrix
• thread blocks corresponds to matrix blocks
• threads corresponds to matrix elements

Each thread is responsible for computing one element in one matrix-block $C_{ij}$ but it needs data from several matrix blocks of A (in block row) and B (in block col).

Each block matrix-matrix multiplication can be done in parallel over the threads in a thread block but each thread needs to communicate with the other threads within its row (of A block) and column (of B block).

\[
C_{ij} + A_{ik} \times B_{kj}
\]

Read in matrix blocks to the faster shared memory and do the block matrix-matrix multiplication in parallel over threads.
// Host code

// Create 2D blocks and grid (maps to matrix)
dim3 dimBlock( blocksize, blocksize );
dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );

// Device (global memory) arrays
float *ad, *bd, *cd;
cudaMalloc( (void**)&ad, size );
cudaMalloc( (void**)&bd, size );
cudaMalloc( (void**)&cd, size );
cudamemcpy( ad, a, size, cudaMemcpyHostToDevice );
cudamemcpy( bd, b, size, cudaMemcpyHostToDevice );
matrix_mul<<<dimGrid, dimBlock>>>( ad, bd, cd, N );
cudamemcpy( c, cd, size, cudaMemcpyDeviceToHost );
cudaFree( ad );
cudaFree( bd );
cudaFree( cd );

// Device code

__global__ void matrix_mul(float *a, float *b, float *c, int N)
{
  float Cvalue = 0;

  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;

  int row = threadIdx.y;
  int col = threadIdx.x;

  // Loop over all the sub-matrices of A and B that are
  // required to compute C-block
  for (int m = 0; m < (N / BLOCK_SIZE); ++m) {

    // Pointer to corresponding A-block and B-block
    int ABlock = blockRow * N * BLOCK_SIZE + m * BLOCK_SIZE;
    int BBlock = m * N * BLOCK_SIZE + blockCol * BLOCK_SIZE;

    // Shared memory used to store A-block and B-block
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
// Load A-block and B-block to shared memory
// Each thread loads one element
As[row][col] = A[Ablock+row*N+col];
Bs[row][col] = B[Bblock+row*N+col];

// Make sure data is loaded on all threads
__syncthreads();

// Multiply Asub and Bsub together
for (int e = 0; e < BLOCK_SIZE; ++e)
    Cvalue += As[row][e] * Bs[e][col];

// Make sure computations are done before loading
// next matrix blocks
__syncthreads();

// Write Cvalue to global memory
// Each thread writes one element
int Cblock = blockRow*N*BLOCK_SIZE+blockCol*BLOCK_SIZE;
C[Cblock+row*N+col]=Cvalue;

Performance of Matrix Multiplication
NVIDIA GeForce 9400M, 16 cores

<table>
<thead>
<tr>
<th>Size (rows)</th>
<th>Ref CPU (sec)</th>
<th>MxM naive (sec)</th>
<th>MxM block (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.31</td>
<td>0.12</td>
<td>0.07</td>
</tr>
<tr>
<td>1024</td>
<td>2.80</td>
<td>1.02</td>
<td>0.41</td>
</tr>
<tr>
<td>2048</td>
<td>22.4</td>
<td>8.44</td>
<td>2.86</td>
</tr>
</tbody>
</table>

2.7 speedup for naïve implementation
7.8 speedup* for block implementation

Lesson: Use shared memory when possible, it is fast and does not require contiguous accesses.

(*16 cores but only half clock frequency on GPU)
Example: Numerical PDE Solver

\[ u_t + u_x + u_y = F(t,x,y) \quad 0 \leq x \leq 1, 0 \leq y \leq 1 \]

\[ \begin{cases} u(t,0,y) = h_1(t,y) & 0 \leq y \leq 1 \\ u(t,x,0) = h_2(t,x) & 0 \leq x \leq 1 \\ u(0,x,y) = g(x,y) & \text{Initial Conditions} \end{cases} \]

Solve with explicit Finite Difference Method (Leapfrog).
(Compare Game-of-life)

Core of the computations:

```plaintext
for k=2,Nt
  t=k*dt; Uold=U; U=Unew;
  for j=1,Ny-1
    for i=1,Nx-1
      x=i/Nx; y=j/Ny
      Unew(i,j)=Uold(i,j)+2*dt*(F(t,x,y)-
                                   (U(i+1,j)-U(i-1,j))/(2*dx)-
                                   (U(i,j+1)-U(i,j-1))/(2*dy))
    end for
  end for
end for
```

Update of each element \((\text{Unew}(i,j))\) is perfectly parallel within the \(k\)-loop.
Computational Stencil:

Each thread updates one element of the grid but needs neighbor data except on the boundaries where we use boundary conditions.

```c
// Host code

// Create 2D blocks and grid (maps to computational grid)
dim3 dimBlock( blocksize, blocksize );
dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );

// Device (global memory) arrays
float *duold, *du, *dunew;
cudamalloc( (void**)&duold, size );
cudamalloc( (void**)&du, size );
cudamalloc( (void**)&dunew, size );
cudamemcpy( duold, uold, size, cudamemcpyHostToDevice );
cudamemcpy( du, u, size, cudamemcpyHostToDevice );
for (k=2; k<Nt; k++) {
    step<<<dimGrid, dimBlock>>>( duold, du, dunew );
    temp=duold; duold=du; du=dunew; dunew=temp;
}
cudamemcpy( unew, du, size, cudamemcpyDeviceToHost );
cudafree( duold );
cudafree( du );
cudafree( dunew );
```
// Device code

__device__ float up(float x, float y)
{ return exp(x+y)*x*x+y*y+y*2*exp((x+y)/2)+cos((x+y)/2); }

__device__ float F(float x, float y)
{ return 2*exp(x+y)+3*x*x+6*y*y+y*2+cos((x+y)/2); }

__device__ float h(float x) { return sin(2*pi*x); }

__global__ step(float *uold, float *u, float *unew)
{ int j = blockDim.x * blockIdx.x + threadIdx.x;
  int i = blockDim.y * blockIdx.y + threadIdx.y;
  int len = Ny+1;
  float x, y, dd;
  x = (float)(i)/Nx;
  y = (float)(j)/Ny;
  if (i==0 || i==Nx | j==0 | j==Ny )
    unew[i*len+j] = up(x,y)+h(x+y-2*dt);
  else {
    dd = u[(i+1)*len+j]-2*u[i*len+j]+u[(i-1)*len+j]/2*Nx;
    dd = u[i*len+j]-2*u[i*len+j-1]+u[i*len+j+1]/2*Ny;
    unew[i*len+j] = uold[i*len+j]+2*dt*(F(x,y)-dd);
  }
}

---

**Performance of PDE solver**

**NVIDIA GeForce 9400M, 16 cores**

<table>
<thead>
<tr>
<th>Size</th>
<th>PDE CPU</th>
<th>PDE GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.046 s</td>
<td>0.032 s</td>
<td>1.4</td>
</tr>
<tr>
<td>128x128</td>
<td>0.36 s</td>
<td>0.099 s</td>
<td>3.6</td>
</tr>
<tr>
<td>256x256</td>
<td>2.9 s</td>
<td>0.50 s</td>
<td>5.8</td>
</tr>
<tr>
<td>512x512</td>
<td>23.0 s</td>
<td>3.3 s</td>
<td>7.0</td>
</tr>
<tr>
<td>1024x1024</td>
<td>186 s</td>
<td>25.3 s</td>
<td>7.4</td>
</tr>
</tbody>
</table>

**Performance bottlenecks:**
- Move data to/from GPU
- Non-contiguous memory accesses
- Divergent instructions
Divergent instructions:

The threads in a block are partitioned into groups of threads, called **warps**, each containing 32 threads.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

**Lesson:** Divergent execution can dramatically hurt performance! Avoid branching within warps if possible.
CUDA Streams:

Can overlap memory transfers and different computational kernels by using streams:

- cudaStreamCreate(...)  
- cudaMemcpyAsync(...)  
- Launch kernel(s)  
- cudaStreamSynchronize(...)  
- cudaDeviceSynchronize(...)  
- cudaStreamDestroy(...)  

See more in CUDA C Programming Guide (http://developer.nvidia.com/)

CUDA Libraries (CUDA Toolkit):

- CUBLAS  
  Basic linear algebra subroutines for dense matrices  
- CUFFT  
  Fast Fourier Transform, 1D, 2D, 3D  
- CUSPARSE  
  Basic linear algebra subroutines for sparse matrices  
- CURAND  
  Random number generator (Monte Carlo simulations)  

Many more, especially for graphics.  
(don’t re-invent the wheel).
CUDA Tools (CUDA Toolkit):

- cuda-gdb: Cuda code debugger
- cuda-memcheck: Detects array out-of-bounds
- computeprof.app: Visual performance profiler

GPU programming, alternatives

**OpenCL** (Open Computing Language):
A framework for writing programs that execute across heterogeneous platforms (CPUs, GPUs, DSPs, FPGAs)

**OpenACC** (Open ACCelerators):
High level directive based programming of CPUs and GPUs for scientific applications

**OpenMP 4.5 – higher:**
*target* - directive to off-load computations to accelerators, e.g., the GPU

**Matlab Parallel Toolbox:**
Includes functions for executing code on GPU
## Supercomputers (top500.org)

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Country</th>
<th>Rank</th>
<th>Speed</th>
<th>Storage</th>
<th>Power</th>
<th>Purpose</th>
<th>Sponsors</th>
<th>Web site</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Summit</td>
<td>United States</td>
<td>1</td>
<td>2,971,838</td>
<td>143,300.0</td>
<td>206,794.9</td>
<td>IBM Power Systems</td>
<td>U.S. Department of Energy</td>
<td><a href="http://www.top500.org/summit/">www.top500.org/summit/</a></td>
</tr>
<tr>
<td>2</td>
<td>Sierra</td>
<td>United States</td>
<td>1</td>
<td>1,572,480</td>
<td>144,450.0</td>
<td>125,712.0</td>
<td>IBM Power Systems</td>
<td>IBM</td>
<td><a href="http://www.top500.org/sierra/">www.top500.org/sierra/</a></td>
</tr>
<tr>
<td>3</td>
<td>TaihuLight</td>
<td>China</td>
<td>1</td>
<td>1,164,600</td>
<td>93,816.4</td>
<td>125,435.3</td>
<td>IBM Power Systems</td>
<td>IBM, NVIDIA</td>
<td><a href="http://www.top500.org/taihu/">www.top500.org/taihu/</a></td>
</tr>
<tr>
<td>4</td>
<td>Tianhe-2A</td>
<td>China</td>
<td>1</td>
<td>995,740</td>
<td>61,444.5</td>
<td>186,878.7</td>
<td>IBM Power Systems</td>
<td>IBM, NVIDIA, China National Supercomputer Center in Guangzhou</td>
<td><a href="http://www.top500.org/tianhe/">www.top500.org/tianhe/</a></td>
</tr>
<tr>
<td>5</td>
<td>Piz Daint</td>
<td>Switzerland</td>
<td>1</td>
<td>387,872</td>
<td>27,233.3</td>
<td>27,194.3</td>
<td>IBM Power Systems</td>
<td>IBM, Cray Inc.</td>
<td><a href="http://www.top500.org/piz/">www.top500.org/piz/</a></td>
</tr>
</tbody>
</table>