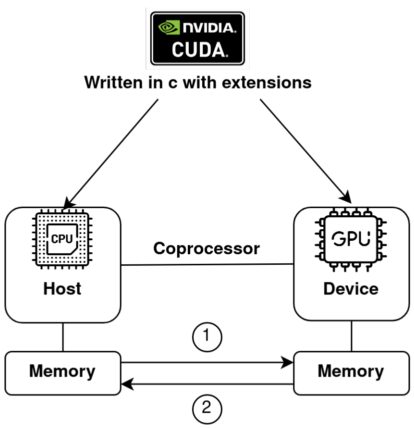
Parallel Programming

Introduction

Three traditional ways how designers make computers run faster:
* More processors
* Faser clocks
* More work / clock cycle

CPU	GPU
Complex control hardware	Simpler control hardware
Flexibility and performance	More hardware for computation
Expensive in terms of power	Potentially more power efficient
	More restrictive programing model

Build a Power-Efficient high performance processor:
- **Minimize latency**. Latency is the amaunt of time to complete a task (seconds). Used in (CPU)
- Throughput Is tasks compledted per unit time (Jobs/hours). Used in (GPU)



A typical GPU program :

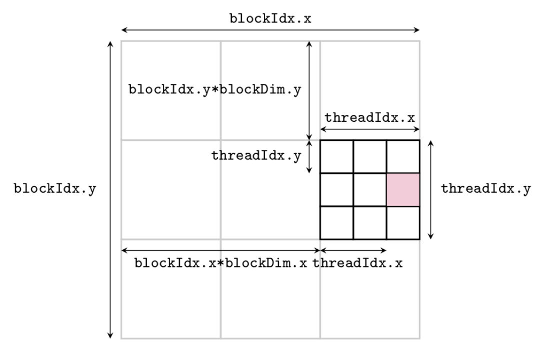
1. CPU allocates storage on GPU (cuda Malloc)
2. CPU copies input data from CPU TO GPU (cuda Memcpy)
3. CPU launches kernels on GPU to process the data (kernel launch)
4. CPU copies results back to CPU from GPU (cuda Memcpy)

kerner

Kernel look like serial programs. write your program as if it will run on one thread [a]. the GPU will run that program on many threads
the kerner Lanch:

```
Kernel<<<Grid of blocks, Block of threades>>>
(d_out,d_in)
```

Maximum number of threads/block: (older gpus:512 / newer gpus 1024)
thread Idx : thread within block (threadIdx.x ,threadIdx.y)
blockDim : size of a block.
blockIdx : block with in grid
gridDim : size of grid
Matrix



- colonne = blockIdx.x*blockDim.x+theadIdx.x
- ligne = blockIdx.y*blockDim.y+theadIdx.y

```
kenel<<<dim3(bx,by,bz),dim3(tx,ty,tz),
shmem>>>(...)
```

shmem : shared memory par block in byets.

[a] one independent path of execution through the code

Communication Patterns

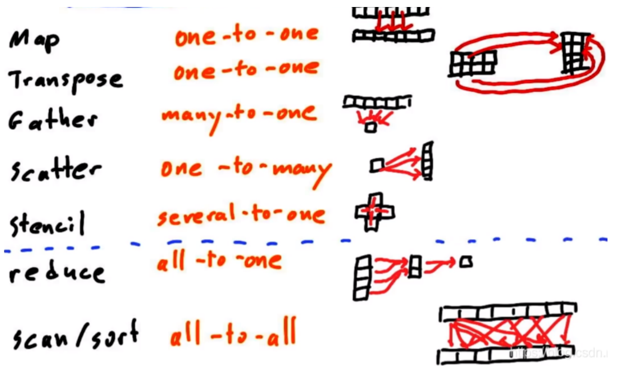
MAP: With Map, you've got many data elements. Such as elements of an array, or entries in a matrix, or pixels in an image. And you're going to do the same function, or computational task, on each piece of data. This means each task is going to read from and write to a specific place in memory. There's a 1 to 1 correspondence between input and output.
GATHER: This operation is called a gather because each calculation gathers input data elements together from different places to compute an output result. suppose that you want each thread to compute and store the average across a range of data elements. Say maybe we want to average each set of 3 elements together. In this case each thread is going to read the values from 3 locations in memory and write them into a single place and so on.
Scatter: Rather than having each thread read three neighboring elements, average their value, and write a single output result, we can have each thread read a single input result and add 1 3rd of its element's value to the three neighboring elements. So, each of these writes, really be a, an increment operation. we call this scatter because the threads are scattering the results over memory.
Stencil: Stencil codes update each element in an array using neighboring array elements in a fixed pattern called the stencil.
TRANSPOSE:the transpose operation is where tasks reorder data elements in memory.exemple: array of structure to structure of array (AoS and SoA).
FOR EXAMPLE:
Map:output[i] = function(input[i])
GATHER:output[i] = input[gatherLoc[i]] i.e.

```
 if(i % 2)
out[i] = (in[i-1] + in[i] + in[i+1]) / 3.0;
```

SCATTER: output[scatterLoc[i]] = input[i] i.e.

```
if(i % 2 {
out[i - 1] = in[i] * pi;
 out[i] = in[i] * pi;
 out[i + 1] = in[i] * pi; }
```

Parallel Communication Patterns Recap



View of Hardware

GPU: N Streaming Processors (SMs) (m simple Processors + Memory)

GPU is responsible for allocating thread blocks to SMs

All SMs run in parallel and independently

A thread block may not run on more than one SM

CUDA Programming Features

CUDA Makes Few Guarantees About when and where Thread Blocks will run

Advantages

- hardware can run things efficiently
- no waiting on slowpokes
- scalability! (same code for different hardwares)

Consequences:

- no assumptions between blocks to SMs
- no communications between blocks, 'dead lock'

Dead lock: 2 computer programs sharing the same resource are effectively preventing each other from accessing the resource

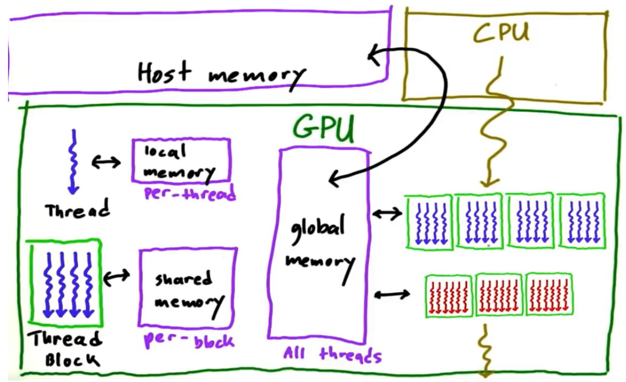
CUDA guarantees that:

- all threads in a block run on the same sm at the same time
- all blocks in a kernel finish before any blocks from the next kernel run

Atomic Memory Operations

e.g. atomicAdd(& g[i], 1);
guarantees only 1 thread can read-modify-write a variable at the same time

GPU Memory Model



Synchronization

Barrier – point in the program where threads stop and wait, all threads reach barrier and they can proceed.

```
int idx = threadIdx.x;
__shared__ int array[128];
array[idx] = threadIdx.x;
__syncthreads();
if (idx < 127) {
    int temp = array[idx+1];
    __syncthreads();
    array[idx] = temp;
    __syncthreads();
```

Writing Efficient Programs

1. Maximize arithmetic intensity (math/intensity)
- maximize compute ops per thread
- minimize time spent on memory per thread

- move frequently-accessed data to fast memory (local > shared >> global >> CPU memory)
- **coalesce** global memory accesses, threads read and write contiguous memory locations (larger stride, more memory transactions -> not so good; random -> bad)

2. Avoid thread divergence

code

Listing 1: Squaring Numbers

```
#include <stdio.h>
__global__ void square(float *d_out,
    float *d_in)
{
    int idx = threadIdx.x;
    float f = d_in[idx];
    d_out[idx] = f*f;
}//end of square kernel

int main(void)
{
    const int ARRAY_SIZE = 64;
    const int ARRAY_BYTES = ARRAY_SIZE *
        sizeof (float);
    //——generate the input array on host
    ——
    //build an array
    float h_in[ARRAY_SIZE];
    //convert each element in array to
        float
    for (int i = 0; i<ARRAY_SIZE ; i++)
    {
        h_in[i] = float(i);
    }//end of for loop
    float h_out[ARRAY_SIZE];
    //——declare GPU memory pointers——
    //To declare variable on GPU, you deal
        with it as pointer
    float * d_in;
    float * d_out;
    //——allocate GPU memory——
    cudaMalloc((void **) &d_in,
        ARRAY_BYTES);
    cudaMalloc((void **) &d_out,
        ARRAY_BYTES);
    //——transfer the array to the GPU
    ——
    cudaMemcpy( d_in, h_in, ARRAY_BYTES,
        cudaMemcpyHostToDevice );
    //——Launch the kernel——
    square<<<1, ARRAY_SIZE>>> (d_out, d_in
        );
    //copy back the result array to CPU
    cudaMemcpy( h_out, d_out, ARRAY_BYTES,
        cudaMemcpyDeviceToHost );
    for (int  i =0;i<ARRAY_SIZE; i++)
    {
        printf("%f", h_out[i]);
    }//end of for loop
    cudaFree(d_in);
    cudaFree(d_out);
    return 0;
}//end of main
```

code

Listing 2: Color to Greyscale Conversion

```

__global__
void rgba_to_greyscale(const uchar4* const rgbaImage, unsigned char* const greyImage, int numRows, int numCols)
{
  int y = threadIdx.y+ blockIdx.y*blockDim.y;
  int x = threadIdx.x+ blockIdx.x*blockDim.x;
  if (y < numCols && x < numRows) {
   int index = numRows*y +x;
  uchar4 color = rgbaImage[index];
  unsigned char grey = (unsigned char)(0.299f*color.x+ 0.587f*color.y + 0.114f*color.z);
  greyImage[index] = grey;
  }
}

void your_rgba_to_greyscale(const uchar4 * const h_rgbaImage, uchar4 * const d_rgbaImage, unsigned char* const d_greyImage, size_t numRows, size_t numCols)
{
  //You must fill in the correct sizes for the blockSize and gridSize
  //currently only one block with one thread is being launched

  int   blockWidth = 32;

  const dim3 blockSize(blockWidth, blockWidth, 1);
  int   blocksX = numRows/blockWidth+1;
  int   blocksY = numCols/blockWidth+1;
  const dim3 gridSize( blocksX, blocksY, 1);
  rgba_to_greyscale<<<gridSize, blockSize>>>(d_rgbaImage, d_greyImage, numRows, numCols);

  cudaDeviceSynchronize(); checkCudaErrors(cudaGetLastError());
}
```

code

Listing 3: hello blockIdx

```
#include <stdio.h>

#define NUM_BLOCKS 16
#define BLOCK_WIDTH 1

__global__ void hello()
{
    printf("Hello world! I'm a thread in block %d\n", blockIdx.x);
}


int main(int argc,char **argv)
{
    // launch the kernel
    hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

    // force the printf()s to flush
    cudaDeviceSynchronize();

    printf("That's all!\n");

    return 0;
}
```

code

Listing 4: Using different memory spaces in CUDA

```
#include <stdio.h>
/* using local memory */
__global__ void use_local_memory_GPU(float in)
{
    float f;    // variable "f" is in local memory and private to each thread
    f = in;     // parameter "in" is in local memory and private to each thread
}
/* using global memory */
__global__ void use_global_memory_GPU(float *array)
{
    // "array" is a pointer into global memory on the device
    array[threadIdx.x] = 2.0f * (float) threadIdx.x;
}
/* using shared memory */
__global__ void use_shared_memory_GPU(float *array)
{
    int i, index = threadIdx.x;
    float average, sum = 0.0f;
    // __shared__ var are visible to all threads in the thread block
    // and have the same lifetime as the thread block
    __shared__ float sh_arr[128];
    // here, each thread is responsible for copying a single element.
    sh_arr[index] = array[index];
    __syncthreads();
    // find the average of all previous elements
    for (i=0; i<index; i++) { sum += sh_arr[i]; }
    average = sum / (index + 1.0f);
    // since array[] is in global memory, this change will be seen by the host (and potentially
    // other thread blocks, if any)
    if (array[index] > average) { array[index] = average; }
    // the following code has NO EFFECT: it modifies shared memory, but
    // the resulting modified data is never copied back to global memory
    // and vanishes when the thread block completes
    sh_arr[index] = 3.14;
}
```

code

Listing 5: Using different memory main

```
int main(int argc, char **argv)
{
    /* First, call a kernel that shows using local memory */
    use_local_memory_GPU<<<1, 128>>>(2.0f);
    /* Next, call a kernel that shows using global memory */
    float h_arr[128];   // convention: h_ variables live on host
    float *d_arr;       // convention: d_ variables live on device (GPU global mem)
    // allocate global memory on the device, place result in "d_arr"
    cudaMalloc((void **) &d_arr, sizeof(float) * 128);
    // now copy data from host memory "h_arr" to device memory "d_arr"
    cudaMemcpy((void *)d_arr, (void *)h_arr, sizeof(float) * 128, cudaMemcpyHostToDevice);
    // launch the kernel (1 block of 128 threads)
    use_global_memory_GPU<<<1, 128>>>(d_arr);  // modifies the contents of array at d_arr
    // copy the modified array back to the host, overwriting contents of h_arr
    cudaMemcpy((void *)h_arr, (void *)d_arr, sizeof(float) * 128, cudaMemcpyDeviceToHost);
    // ... do other stuff ...
    /* Next, call a kernel that shows using shared memory*/
    // as before, pass in a pointer to data in global memory
    use_shared_memory_GPU<<<1, 128>>>(d_arr);
    // copy the modified array back to the host
    cudaMemcpy((void *)h_arr, (void *)d_arr, sizeof(float) * 128, cudaMemcpyHostToDevice);
    // ... do other stuff ...
    return 0;
}
```