HW 4

```
#include <omp.h>
#include  <stdlib.h> //atoi
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cmath>
//timing routines
#include  <time.h>
#include <sys/time.h>

#define SIZE 256

using namespace std;

//function for elapsed time
double getTimeElapsed(struct timeval end, struct timeval start)
   {
      return (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1000000.00;
   }
int main(int argc, char* argv[])
{
//  timing info variables in main
   struct timeval t0, t1;
   double htime;
   bool print = false;
```

HW4-2

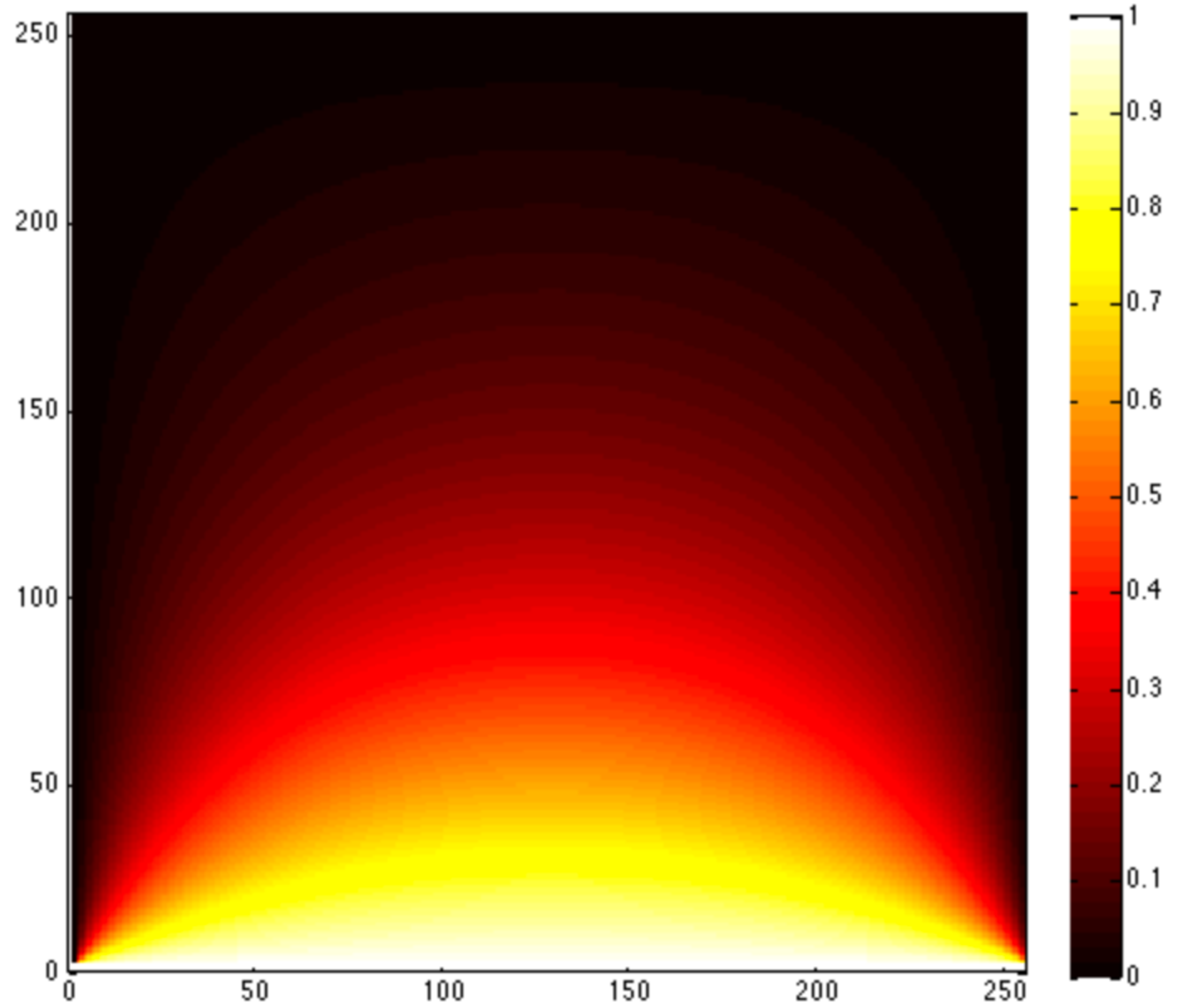
```
float phi[SIZE][SIZE]={};
  float phinew[SIZE][SIZE] = {};
  float dx;
  int testx = SIZE/2;
  int testy = SIZE/2;
  int np = 8;   //default
  float converge;
  int i,j;
  //set up boundary conditions;  y=0
  for (i=0; i<SIZE; i++)
    {
      phi[i][0] = 1.000000;  //all other sides already defined as zero
    }
     if (argc >1)
   {
     np= atoi(argv[1]);  //parse command line for number of threads

     printf("Number of threads: %d\n",np);
   }
  else
   {
   printf("be sure you had an arg on command line for np!\n");
   exit(1);
   }
    // omp_set_dynamic(false);
    omp_set_num_threads(np);
```

HW4-3

```
// set up iterations
gettimeofday(&t0, NULL);
for (int iter= 0; iter<100000; iter++)
  {
   cout<<"Iteration: "<<iter<<", ";
   converge=0.0f;
    #pragma omp parallel for private(i) reduction(+:converge)
     for (j=1; j <(SIZE-1); j++)
      {
       for (i= 1; i< (SIZE-1); i++)
         {
          phinew[i][j]=0.25*(phi[i-1][j] + phi[i+1][j] + phi[i][j+1]+phi[i][j-1]);
          converge+=abs(phinew[i][j]-phi[i][j]);
         }
      } //end omp for



        if(iter > SIZE)
         {
           cout <<converge/(SIZE*SIZE)<<endl;
         }

      // update
     #pragma omp parallel for private(i)
      for (j=1; j <(SIZE-1); j++)
        {
         for (i=1; i<(SIZE-1); i++)
            phi[i][j]=phinew[i][j];
        }  //end omp for
```

HW4-4

```
  if (converge/(SIZE*SIZE) < 0.00001)
    {
       gettimeofday(&t1, NULL);
       cout<<"Converged: "<<converge/(SIZE*SIZE)<<endl;
      htime=getTimeElapsed(t1,t0);
              printf("time for computation: %f \n", htime);
               ofstream myfile;
                  myfile.open("Laplace.txt");
               if(myfile.is_open())
                 {
                   cout << "file open for writing\n";
                   for (j=0; j<SIZE; j++)
                        { for (i=0; i<SIZE; i++)
                            {myfile <<phi[i][j] <<" " ;}
                       myfile<<endl;
                        }
               }
               else
                {cout << "file open failure\n";
                 }
    myfile.close();
     return iter;
    }
 }  //end iteration loop
              printf("Did not converge\n")
}
```



Using the GPU

Start serial work on CPU

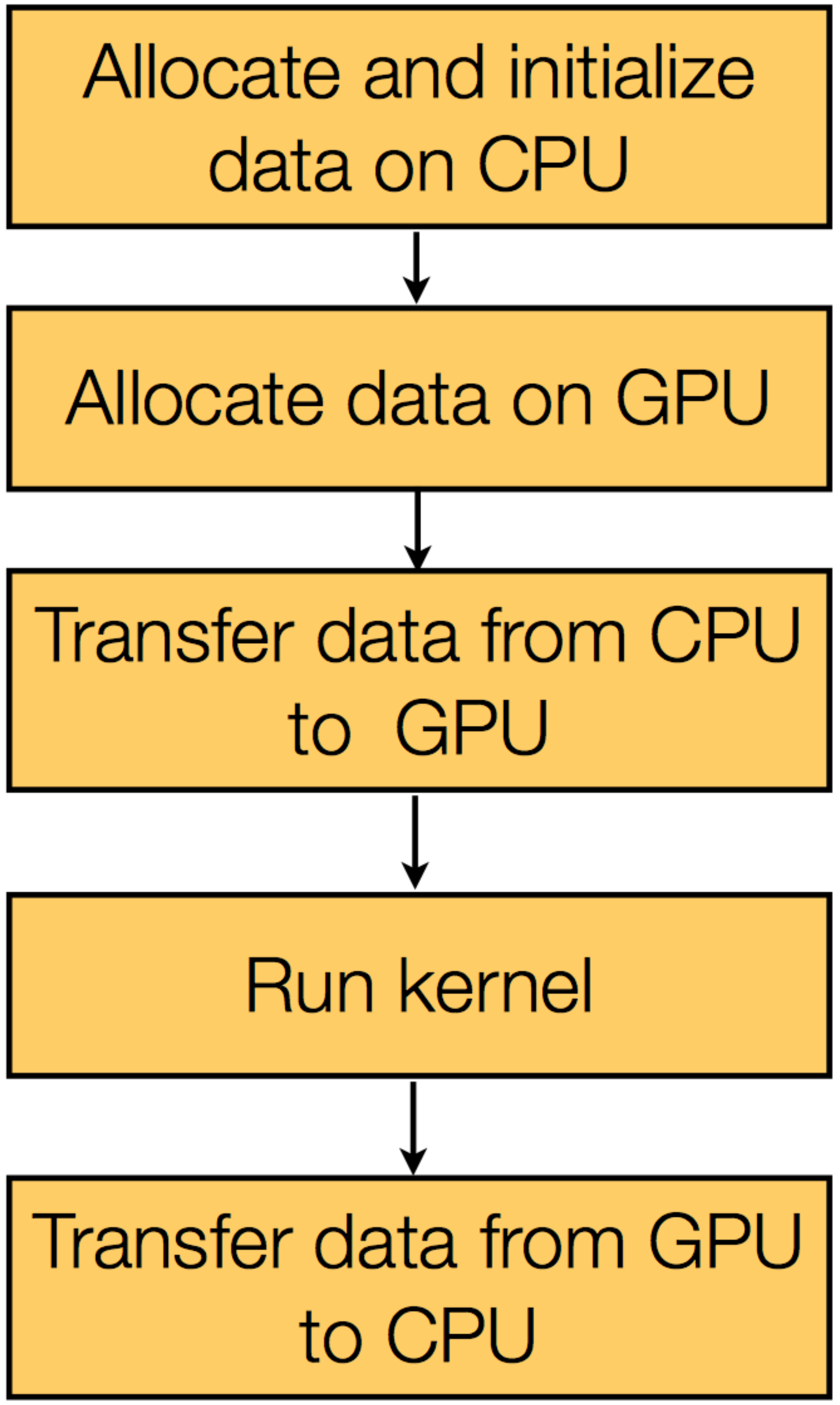
Send data to the GPU

Do computational work on the GPU (kernel)

Pass back answers only when necessary;
that is, do as much work on GPU as possible

This avoids the PCI-e bus bandwidth issues

Remember: kernels are like functions, but massively parallel.


Allocate and initialize data on CPU
Allocate data on GPU
Transfer data from CPU to GPU
Run kernel
Transfer data from GPU to CPU

Data Parallelism

Data parallelism is a program property where many arithmetic
operations can be performed on the data simultaneously.

For CUDA programs: part done on CPU; part on the GPU. Parts with
no/little data parallelism, do on the CPU; parts with data parallelism, use the GPU.

HOST (CPU)

CUDA routines for:

memory allocation

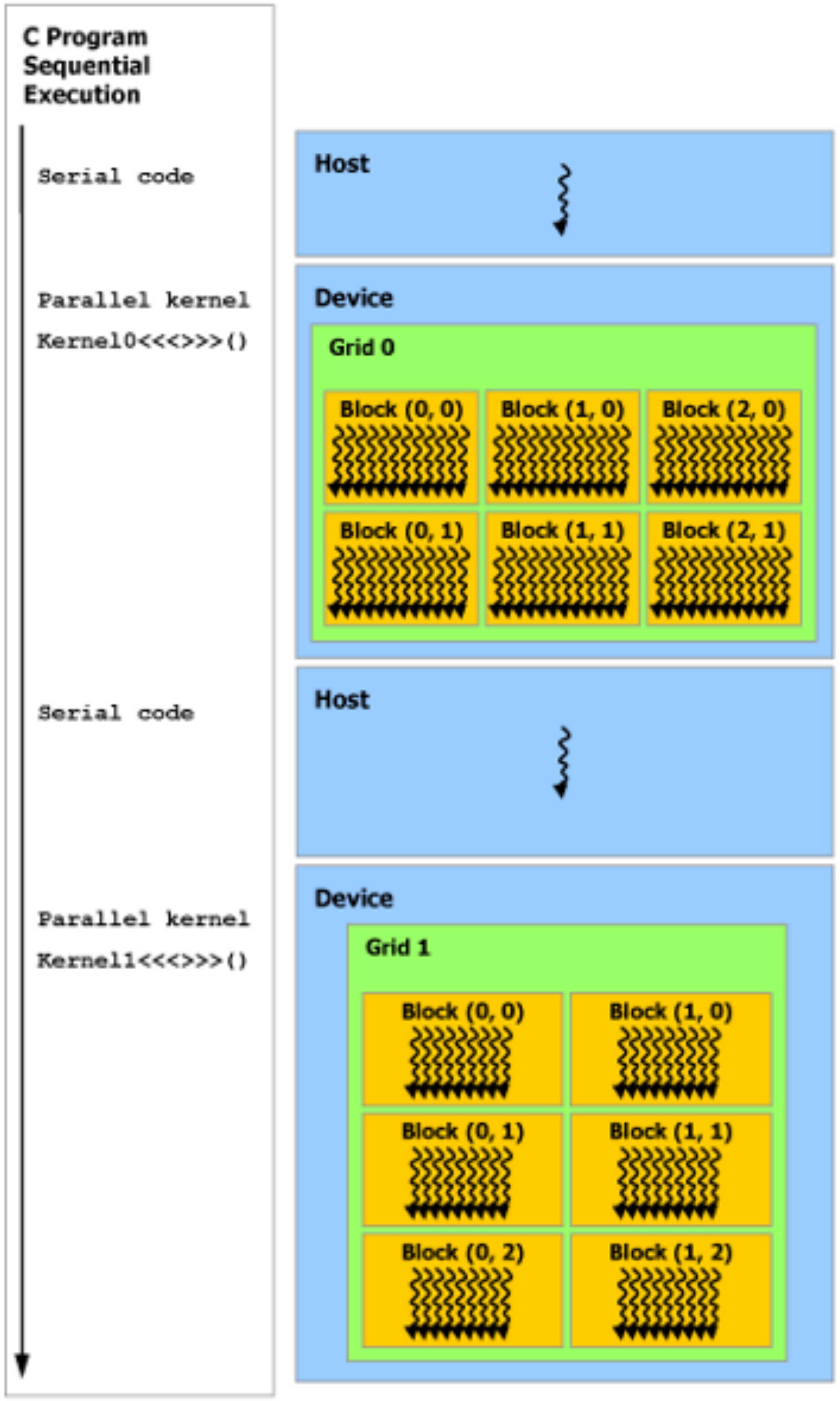
data transfer to and from DEVICE

- ordinary data
- constants
- texture arrays (2D read only)

error checking

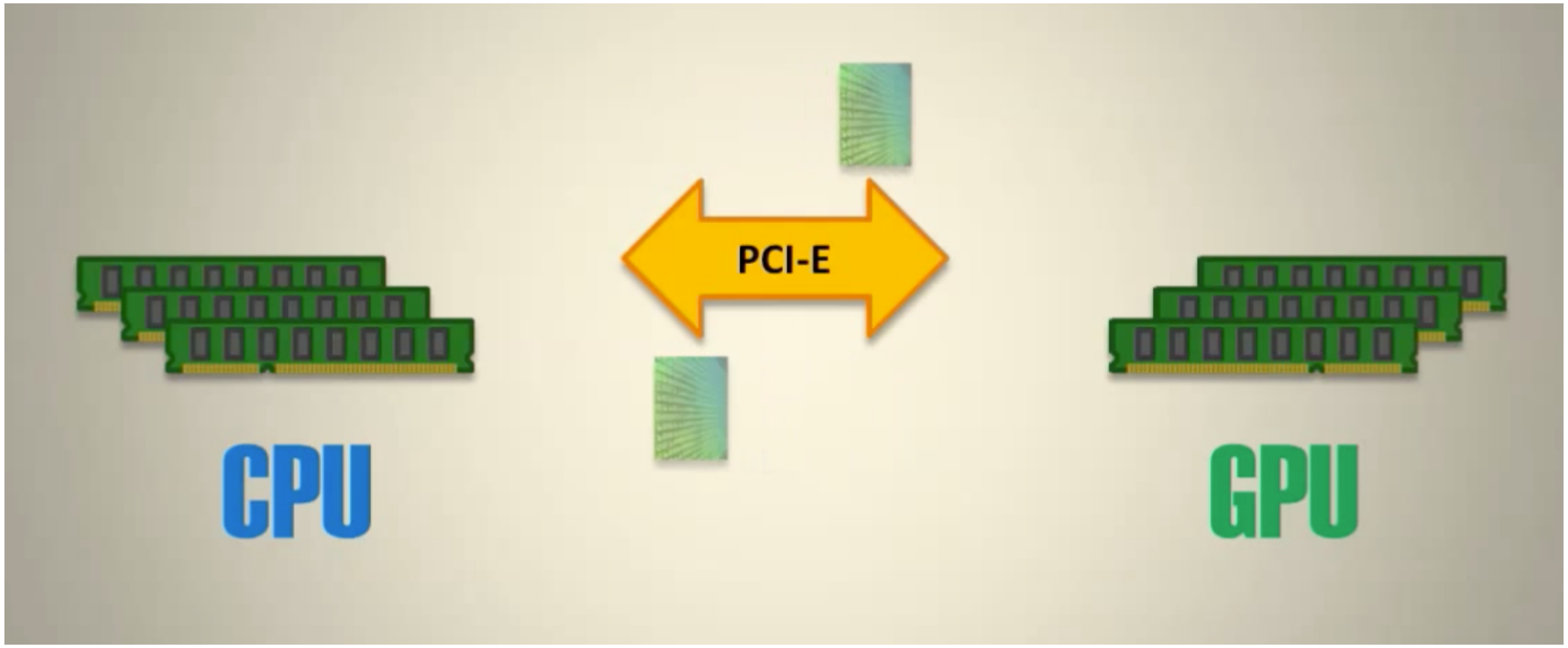
timing

Heterogeneous Programming



Why GPU code can be slower than CPU

Data transfers between devices



Best Practices

Keep the GPU cores busy. It is often faster to just crunch numbers rather than be clever.

Example: Calculating forces between objects.

For N objects, calculate the force on object (i) due to object(j). Since force on object(j) due to object(i) is equal and opposite, you could neglect half the computations.

But the logic programming/calculations necessary to do this is not worth the time. Compute everything, even if redundant.

Memory

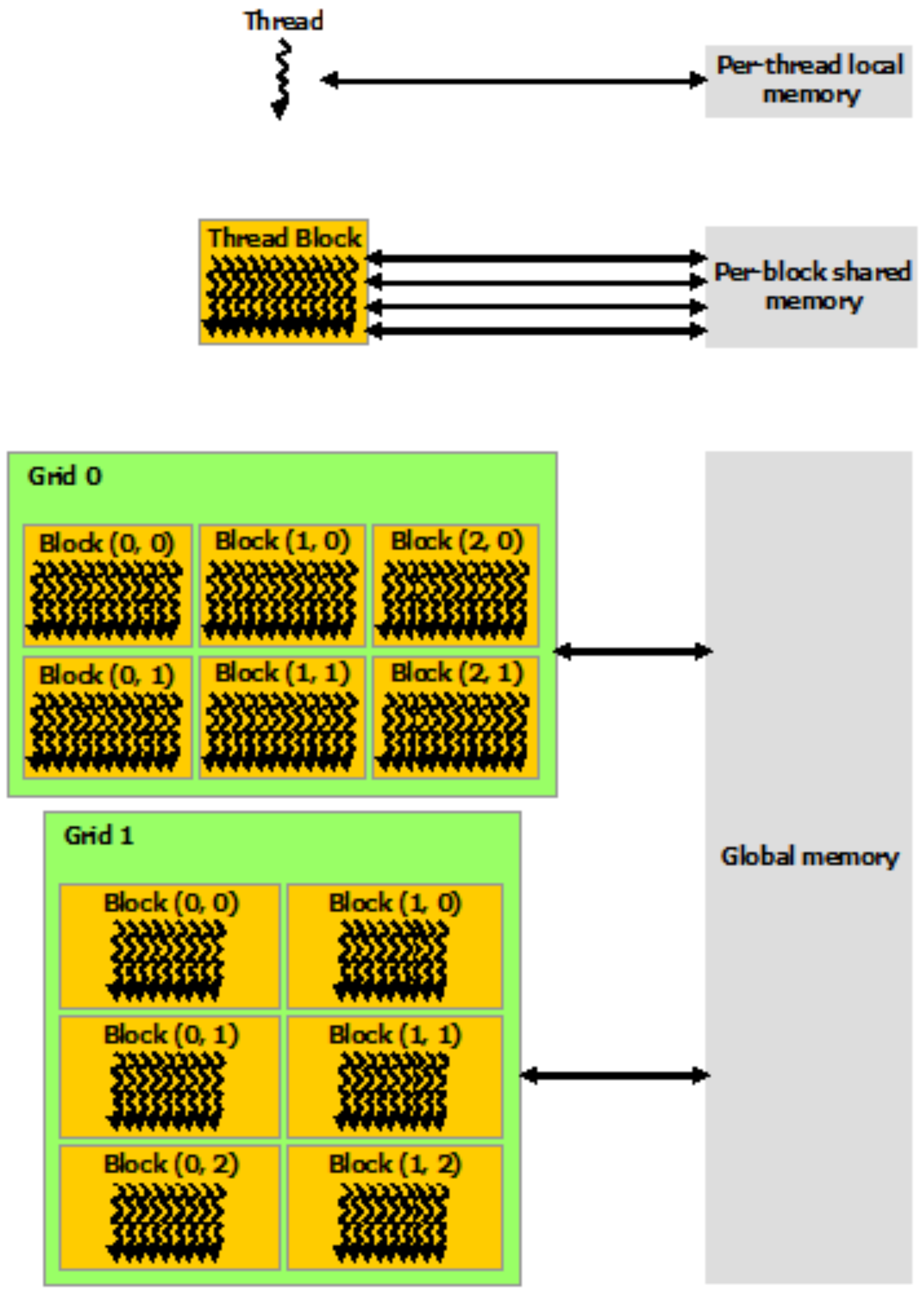
CPU and GPU have separate memory (DRAM-dynamic random access memory); with the CPU generally having more.

For this reason, doing things on the GPU requires sending data from CPU to GPU, or, in the lingo, Host to Device. And then usually back again.

(Note for compute capable 3.0 devices, can write from GPU.)

Several kinds of Device memory:
Per thread (registers), constant memory, shared (SM), texture, global (slowest).

Different types of Memory on GPU



cudaMalloc(&d_array,num_bytes)

Allocate (global) memory on device

cudaMalloc needs address of pointer for the allocated memory
and the size of the memory required.

Pass-by-pointer (C-style) (see Class3.pdf)
  Returns the address of a GPU pointer to the memory

In C,

```
cudaMalloc( (void **) &d_array, num_bytes)  //need void ** cast
```

Use cudaFree(d_array) to free memory when done

Error Handling

Every CUDA call returns an error code of type cudaError_t

cudaMalloc(….) returns cudaSucess, cudaErrorInvalidValue,
        cudaErrorInvalidDevicePointer, cudaInvalidMemcpyDirection

cudaError_t cudaGetLastError(void);

Readable error messages can be obtained by

char* cudaGetErrorString (cudaError_t error)

Error Checking CUDA calls

```
/* CUDA error checking. Usage:  after every cuda command, such as
   kernel launch, type cudaCheckErrors("kernel launch failed")
   replacing message with the type of cuda command
      #include <stdio.h>     */

#define cudaCheckError(msg) \
    do { \
        cudaError_t __err = cudaGetLastError(); \
        if (__err != cudaSuccess) { \
            fprintf(stderr, "Fatal error: %s (%s at %s:%d)\n", \
                msg, cudaGetErrorString(__err), \
                __FILE__, __LINE__); \
            fprintf(stderr, "*** FAILED - ABORTING\n"); \
            exit(1); \
        } \
    } while (0)                         // from Robert Crovella, stackoverflow
```

Example:

```
  cudaMalloc( (void **) &d_part_sum, N*sizeof(double));
  cudaCheckError("cudaMalloc");
```

Executing kernels

```
// kernel definition:

__global__ void vectorAdd(int *a, int *b, int *c, int n)
 {
    int i=threadIdx.x;  //replaces for loop as whole function is parallel
    if (i<n)           //only do as many threads as you have data
       c[i] = a[i] + b[i];
  }
```

In main:

```
  vectorAdd<<<1,SIZE>>>(d_a,d_b,d_c, SIZE);   //CUDA kernel call

   kernel <<< num_blocks, threads_per_block>> (args);
```

so one 1D block of length SIZE, so SIZE threads are used in vectorAdd (i = 0, i< SIZE)

MyPinC —> MyPiGPU

Convert MyPinC to GPU code

```
__device__ __constant__  double PI25D= 3.141592653589793238462643;
;
```

From DEVICE (to save copying data back):

```
printf(“Pi = %.10g, error = %.3g \n", sum*interval_size, interval_size*sum-PI25D);
```

```
__syncthreads():
```

Steps:

1. create a kernel function for integrals

2. create space on device for integrands

3. obtain pi on device

4. print out the answer (from device)

```
nvcc  myPiGPU.cu -arch compute_30
```

Kernel

```
__global__ void term (double* d_part_sum, int n)
{
   double interval_size = 1.0/(double)n;
   int tid = blockIdx.x*blockDim.x + threadIdx.x;
   if (tid <n)
    {
      double x = interval_size * ((double) tid -0.5);
      d_part_sum[tid] = 4.0 / (1.0 + x * x);
   }
   __syncthreads();
   if (tid ==0)
     {
     double sum = 0.0;
     for (int i=0; i<n; i++)
        {
          sum += d_part_sum[i];
        }
     printf("Pi = %.10g, error = %.3g \n", sum*interval_size, interval_size*sum-PI25D);
     }
}
```

Using nvcc

```
tcsh

setenv PATH /usr/local/cuda/bin:$PATH

setenv DYLD_LIBRARY_PATH  /usr/local/cuda/lib (or location of
                                                                  cuda/lib)

nvcc myPiGPU.cu -o myPiGPU
```

Compiling CUDA programs

Any program with CUDA programming should be named: xxxx**.cu**
This file can contain both HOST and DEVICE code.

The compiler is **nvcc,** which can handle both C++ and CUDA code. It is capable of compiling all C++ commands for the HOST and most for the DEVICE.

```
    nvcc xxxx.cu       -> a.out

   nvcc xxxx.cu -o xxxxx     yields xxxxx as executable file
```

Compute capability: newer cards have better capabilities

```
nvcc -code=sm_30  xxxx.cu          means compute_capability of 3.0

nvcc -m32   xxxx,cu    means compile 32bit code, default is 64 bit (-m64)
```