GPUs and GPGPU ● In the late 1990s and early 2000s, the computer industry responded to the demand for highly realistic computer video games and video animations by developing extremely powerful graphics processing units or GPUs. ● These processors are designed to improve the performance of programs that need to render many detailed images. ● The existence of this computational power was a temptation to programmers who didn't specialize in computer graphics, and by the early 2000s they were trying to apply the power of GPUs to solving general computational problems, ● problems such as searching and sorting, rather than graphics. This became known as General Purpose computing on GPUs or GPGPU. ● So, One of the biggest difficulties faced by the early developers of GPGPU was that the GPUs of the time could only be programmed using computer graphics APIs, such as Direct3D and OpenGL. ● used graphics concepts, such as vertices, triangles, and pixels to reformulate algorithms for general computational problems added considerable complexity to the development of early GPGPU programs ● Then, languages and compilers were developed to implement general algorithms for GPUs ● Currently the most widely used APIs are CUDA and OpenCL. SIMD architectures ● We often think of a conventional CPU as a SISD device in Flynn's Taxonomy. ● The processor fetches an instruction from memory and executes the instruction on a small number of data items. ● The instruction is an element of the Single Instruction stream—the "SI" in SISD. ● The data items are elements of the Single Data stream—the "SD" in SISD ● We can think of a SIMD processor as being composed of a single control unit and multiple datapaths. ● The control unit fetches an instruction from memory and broadcasts it to the datapaths. ● Each datapath either executes the instruction on its data or is dle.SIMD architectures ● In a typical SIMD system, each datapath carries out the test x[i] >=0. Then the datapaths for which the test is true execute x[i] +=1, while those for which x[i]0 are idle. ● Then the roles of the datapaths are reversed: those for which x[i] >=0 are idle while the other datapaths execute x[i] −= 2. GPU architectures ● A typical GPU can be thought of as being composed of one or more SIMD processors. ● Nvidia GPUs are composed of Streaming Multiprocessors or SMs. ● One SM can have several control units and many more datapaths. ● So an SM can be thought of as consisting of one or more SIMD processors. ● The SMs, however, operate asynchronously: ● there is no penalty if one branch of an if-else executes on one SM, and the other executes on another SM. ● So in our preceding example, if all the threads with x[i] >=0 were executing on one SM, and all the threads with x[i]0 were executing on another, the execution of our if-else example would require only tw.GPU architectures ● Each SM has a relatively small block of memory that is shared among its SPs. ● This memory can be accessed very quickly by the SPs. ● All of the SMs on a single chip also have access to a much larger block of memory that is shared among all the SPs. Accessing this memory is relatively slow. GPU architectures ● The GPU and its associated memory are usually physically separate from the CPU and its associated memory. ● Host: CPU together with its associated memory. ● Device: GPU together with its memory. ● In earlier systems the physical separation of host and device memories required that data was usually explicitly transferred between CPU memory and GPU memory. ● However, in more recent Nvidia systems (those with compute capability ≥ 3.0), the explicit transfers in the source code aren't needed. Heterogeneous computing ● Up to now we've implicitly assumed that our parallel programs will be run on systems in which the individual processors have identical architectures. ● Writing a program that runs on a GPU is

an example of heterogeneous computing. ● The reason is that the programs make use of both a host processor—a conventional CPU—and a device processor—a GPU—the two processors have different architectures. ● We'll still write a single program using the SPMD. BUT now there will be: 1. functions for conventional CPUs 2. and functions for GPUs. So, effectively, we'll be writing two programs.CUDA ● CUDA will be used to program heterogeneous CPU–GPU systems. ● CUDA is a software platform that can be used to write GPGPU programs for heterogeneous systems equipped with an Nvidia GPU. ● CUDA was originally an acronym for "Compute Unified Device Architecture," which was meant to suggest that it provided a single interface for programming both CPU and GPU. ● More recently, however, Nvidia has decided that CUDA is not an acronym; it's simply the name of an API for GPGPU programming. ● There is a language-specific CUDA API for several languages; for example, there ● are CUDA APIs for C, C++, Fortran, Python, and Java. ● We will use CUDA C/C++ ● We'll write a program in which each CUDA thread prints a greeting. ● Since the program is heterogeneous, we will write two programs 1. A host or CPU program and 2. A device or GPU program.Heterogeneous Computing 1 __global__ void Vec_add (2 const f l o a t x [ ] / i n / , 3 const f l o a t y [ ] / i n / , 4 f l o a t z [ ] / out / , 5 const int n / i n /) { 6 i n t my_elt = blockDim . x blockIdx . x + threadIdx . x ; 7 8 / t o t a l threads = b l k _ c t t h _ p e r _ b l k may ben / 9 i f (my_eltn) 10 z [my_elt] = x [my_elt] + y [my_elt] ; 11 } / Vec_add / 12 13 i n t main (i n t argc , char argv [ ]) { 14 i n t n , th_per_blk , blk_ct ; 15 char i_g ; / Are x and y user i n p u t or random? / 16 f l o a t x , y , z , cz ; 17 double diff_norm ; 18 19 / Get the command l i n e arguments , and s e t up v e c t o r s / 20 Get_args (argc , argv , &n, &blk_ct , &th_per_blk , &i_g) ; 21 Allocate_vectors (&x, &y, &z, &cz , n) ; 22 Init_vectors (x , y , n , i_g) ; 23 24 / Invoke k e rne l and wait fo r i t to complete / 25 Vec_add (x , y , z , n) ; 26 cudaDeviceSynchronize ( ) ; 27 28 / Check for c o r r e c t n e s s / 29 Serial_vec_add (x , y , cz , n) ; 30 diff_norm = Two_norm_diff (z , cz , n) ; 31 printf ("Two – norm of difference between host and ") ; 32 printf (" device = %e\n" , diff_norm) ; 33 34 / Free s t o r a g e and q u i t / 35 Free_vectors (x , y , z , cz) ; 36 return 0 ; 37 } / main / serial code parallel code .serial code Simple Processing Flow 1. Copy input data from CPU memory to GPU memory PCI = peripheral component interconnect PCI Bus NVIDIA 2013 14 Simple Processing Flow 1. Copy input data from CPU memory to GPU memory 2. Load GPU program and execute, caching data on chip for performance NVIDIA 2013 PCI Bus 15 Simple Processing Flow 1. Copy input data from CPU memory to GPU memory 2. Load GPU program and execute, caching data on chip for performance 3. Copy results from GPU memory to CPU memory. CUDA Hello ● Note that even though our programs are written in CUDA C, CUDA programs cannot be compiled with an ordinary C compiler. ● So unlike MPI and OpenMP, CUDA is not just a library that can be linked into an ordinary C program: CUDA requires a special compiler. ● For example, an ordinary C compiler (such as gcc) generates a machine language executable for a single CPU (e.g., an x86 processor), but the CUDA compiler must generate machine language for two different processors: the host processor and the device processor.CUDA Hello : Compile and run ● A CUDA program file that contains both host code and device code should be stored in a file with a ".cu" suffix. ● For example, our hello program is in a file called cuda_hello.cu. ● We can compile it using the CUDA compiler nvcc. ● The command should look something like this: $ nvcc –o cuda_hello cuda_hello.cu Device functions (e.g.

Hello()) processed by NVIDIA compiler Host functions (e.g. main()) processed by standard host compiler e.g. gcc ● If we want to run one thread on the GPU, we can type $ ./cuda_hello 1 ● The output will be: Hello from thread 0! If we want to run 5 threads on the GPU, we can type ● $ ./cuda_hello 5 and the output of will be Hello from thread 0! Hello from thread 1! Hello from thread 2! Hello from thread 3! Hello from thread 4! Hello from thread 5! 19 CUDA Hello : Closer Look ● The execution begins on the host in the main function. It gets the number of threads from the command line by calling the C library strtol function. ● the call from the host to the device (kernel) in Line 18 tell the system how many threads to start on the GPU by enclosing the pair 1,thread_count in triple angle brackets >. ● If there were any arguments to the Hello function, we would enclose them in the following parentheses. ● The kernel specifies the code that each thread will execute. ● So, each of our threads will print a message: " Hello from thread %d\n“ ● The decimal int format specifier (%d) refers to the variable threadIdx.x. ● The struct threadIdx is one of several variables defined by CUDA when a kernel is started. ● In our example, the field x gives the relative index or rank of the executing thread. ● After a thread has printed its message, it terminates execution. ● Our kernel code uses the Single-Program Multiple-Data or SPMD paradigm. 20 CUDA Hello : Closer Look ● One very important difference between the execution of an ordinary C function and a CUDA kernel is that kernel execution is asynchronous. ● This means that the call to the kernel on the host returns as soon as the host has notified the system that it should start running the kernel, and even though the call in main has returned, the threads executing the kernel may not have finished executing. ● The call to cudaDeviceSynchronize in Line 21 forces the main function to wait until all the threads executing the kernel have completed. ● If we omitted the call to cudaDeviceSynchronize, our program could terminate before the threads produced any output, and it might appear that the kernel was never called. 21 Threads, blocks, and grids ● What does 1 mean in the function call? Hello (); ● Recall that an Nvidia GPU consists of a collection of streaming multiprocessors (SMs), and each streaming multiprocessor consists of a collection of streaming processors (SPs). ● When a CUDA kernel runs, each individual thread will execute its code on an SP. ● With “1” as the first value in angle brackets, all of the threads that are started by the kernel call will run on a single SM. ● If our GPU has two SMs, we can try to use both of them with the kernel call Hello (); ● If thread_count is even, this kernel call will start a total of thread_count threads, and the threads will be divided between the two SMs: thread_count/2 threads will run on each SM. (What happens if thread_count is odd?) 22 Threads, blocks, and grids ● CUDA organizes threads into blocks and grids. ● A thread block (or just a block) : is a collection of threads that run on a single SM. 1. In a kernel call the first value in the angle brackets specifies the number of thread blocks. 2. The second value is the number of threads in each thread block. ● So when we started the kernel with Hello (); we were using one thread block, which consisted of thread_count threads, and, as a consequence, we only used one SM. ● We can modify our greetings program so that it uses a user-specified number of blocks, each consisting of a user-specified number of threads.Threads, blocks, and grids ● There are several built-in variables that a thread can use to get information on the grid started by the kernel. ● The following four variables are structs that are initialized in each thread’s memory when a kernel begins execution: 1. threadIdx: the rank or index of the thread in its thread block. 2. blockDim: the dimensions, shape, or size of the thread blocks. 3. blockIdx: the rank

or index of the block within the grid. 4. gridDim: the dimensions, shape, or size of the grid. ● All of these structs have three fields, x, y, and z, and the fields all have unsigned integer types. The fields are often convenient for applications. ● For example, an application that uses graphics may find it convenient to assign a thread to a point in two- or three-dimensional space, and the fields in threadIdx can be used to indicate the point's position. ● An application that makes extensive use of matrices may find it convenient to assign a thread to an element of a matrix, and the fields in threadIdx can be used to indicate the column and row of the element. 25 Threads, blocks, and grids ● When we call a kernel with something like: ● The three-element structures gridDim and blockDim are initialized by assigning the values in angle brackets to the x fields. So, effectively, the following assignments are made: ● The y and z fields are initialized to 1. If we want to use values other than 1 for the y and z fields, we should declare two variables of type dim3, and pass them into the call to the kernel. For example, ● This should start a grid with 2×3×1 = 6 blocks, each of which has 4^3 = 64 threads. Iint blk_ct , th_per_blk ; . . . Hello (); gridDim.x = blk_ct ; blockDim.x = th_per_blk ; dim3 grid_dims , block_dims ; grid_dims.x = 2; grid_dims.y = 3; grid_dims.z = 1; block_dims.x = 4; block_dims.y = 4; block_dims.z = 4; . . . Kernel (. . .); ● Note that all the blocks must have the same dimensions. ● CUDA requires that thread blocks be independent. So one thread block must be able to complete its execution, regardless of the states of the other thread blocks. 26 Nvidia compute capabilities and device architectures ● There are limits on the number of threads and the number of blocks. ● The limits depend on what Nvidia calls the compute capability of the GPU. ● The compute capability is a number having the form a.b. It describes the device architecture, e.g., Number of registers, Sizes of memories, and Features & capabilities ● Currently the a-value or major revision number can be 1, 2, 3, 5, 6, 7, 8. The possible b-values or minor revision numbers depend on the major revision value, but currently they fall in the range 0–7. For devices with compute capability1, the maximum number of threads per block is 1024. For devices with compute capability 2.b, the maximum number of threads that can be assigned to a single SM is 1536, and for devices with compute capability2, the maximum is currently 2048. ● There are also limits on the sizes of the dimensions in both blocks and grids. For example, for compute capability1, the maximum x- or y-dimension is 1024, and the maximum z-dimension is 64