CHAPTER

Data parallel computing

2

David Luebke

CHAPTER OUTLINE

Many code examples will be used to illustrate the key concepts in writing scalable parallel programs. For this we need a simple language that supports massive parallelism and heterogeneous computing, and we have chosen CUDA C for our code examples and exercises. CUDA C extends the popular C programming language with minimal new syntax and interfaces to let programmers target heterogeneous computing systems containing both CPU cores and massively parallel GPUs. As the name implies, CUDA C is built on NVIDIA's CUDA platform. CUDA is currently the most mature framework for massively parallel computing. It is broadly used in the high performance computing industry, with sophisticated tools such as compilers, debuggers, and profilers available on the most common operating systems.

An important point: while our examples will mostly use CUDA C for its simplicity and ubiquity, the CUDA platform supports many languages and application programming interfaces (APIs) including C++, Python, Fortran, OpenCL, OpenACC, OpenMP, and more. CUDA is really an architecture that supports a set

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00002-9

of concepts for organizing and expressing massively parallel computation. It is those concepts that we teach. For the benefit of developers working in other languages (C++, FORTRAN, Python, OpenCL, etc.) we provide appendices that show how the concepts can be applied to these languages.

2.1 DATA PARALLELISM

When modern software applications run slowly, the problem is usually having too much data to be processed. Consumer applications manipulate images or videos, with millions to trillions of pixels. Scientific applications model fluid dynamics using billions of grid cells. Molecular dynamics applications must simulate interactions between thousands to millions of atoms. Airline scheduling deals with thousands of flights, crews, and airport gates. Importantly, most of these pixels, particles, cells, interactions, flights, and so on can be dealt with largely independently. Converting a color pixel to a greyscale requires only the data of that pixel. Blurring an image averages each pixel's color with the colors of nearby pixels, requiring only the data of that small neighborhood of pixels. Even a seemingly global operation, such as finding the average brightness of all pixels in an image, can be broken down into many smaller computations that can be executed independently. Such independent evaluation is the basis of *data parallelism*: (re)organize the computation around the data, such that we can execute the resulting independent computations in parallel to complete the overall job faster, often much faster.

TASK PARALLELISM VS. DATA PARALLELISM

Data parallelism is not the only type of parallelism used in parallel programming. Task parallelism has also been used extensively in parallel programming. Task parallelism is typically exposed through task decomposition of applications. For example, a simple application may need to do a vector addition and a matrix-vector multiplication. Each of these would be a task. Task parallelism exists if the two tasks can be done independently. I/O and data transfers are also common sources of tasks.

In large applications, there are usually a larger number of independent tasks and therefore larger amount of task parallelism. For example, in a molecular dynamics simulator, the list of natural tasks include vibrational forces, rotational forces, neighbor identification for nonbonding forces, nonbonding forces, velocity and position, and other physical properties based on velocity and position.

In general, data parallelism is the main source of scalability for parallel programs. With large data sets, one can often find abundant data parallelism to be able to utilize massively parallel processors and allow application performance to grow with each generation of hardware that has more execution resources. Nevertheless, task parallelism can also play an important role in achieving performance goals. We will be covering task parallelism later when we introduce streams.

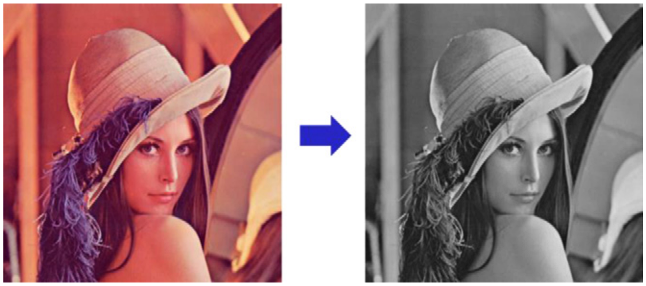


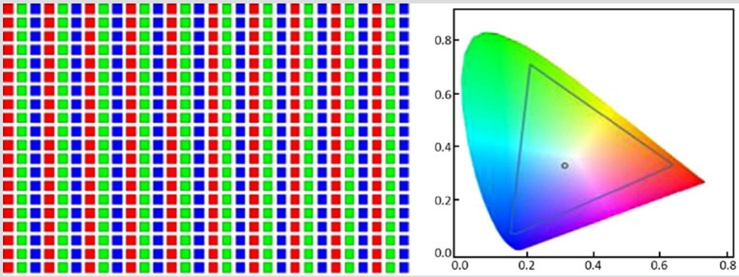
FIGURE 2.1

Conversion of a color image to a greyscale image.

We will use image processing as a source of running examples in the next chapters. Let us illustrate the concept of data parallelism with the color-to-greyscale conversion example mentioned above. Fig. 2.1 shows a color image (left side) consisting of many pixels, each containing a red, green, and blue fractional value (r, g, b) varying from 0 (black) to 1 (full intensity).

RGB COLOR IMAGE REPRESENTATION

In an RGB representation, each pixel in an image is stored as a tuple of (r, g, b) values. The format of an image's row is (r g b) (r g b) … (r g b), as illustrated in the following conceptual picture. Each tuple specifies a mixture of red (R), green (G) and blue (B). That is, for each pixel, the r, g, and b values represent the intensity (0 being dark and 1 being full intensity) of the red, green, and blue light sources when the pixel is rendered.



The actual allowable mixtures of these three colors vary across industry-specified color spaces. Here, the valid combinations of the three colors in the AdobeRGB color space are shown as the interior of the triangle. The vertical coordinate (y value) and horizontal coordinate (x value) of each mixture show the fraction of the pixel intensity that should be G and R. The remaining fraction ($1 - y–x$) of the pixel intensity that should be assigned to B. To render an image, the r, g, b values of each pixel are used to calculate both the total intensity (luminance) of the pixel as well as the mixture coefficients (x, y, $1 - y - x$).

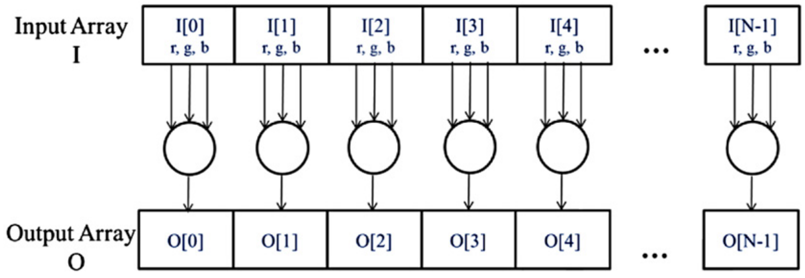


FIGURE 2.2

The pixels can be calculated independently of each other during color to greyscale conversion.

To convert the color image (left side of Fig. 2.1) to greyscale (right side) we compute the luminance value L for each pixel by applying the following weighted sum formula:

$$L = r * 0.21 + g * 0.72 + b * 0.07$$

If we consider the input to be an image organized as an array I of RGB values and the output to be a corresponding array O of luminance values, we get the simple computation structure shown in Fig. 2.2. For example, $O[0]$ is generated by calculating the weighted sum of the RGB values in $I[0]$ according to the formula above; $O[1]$ by calculating the weighted sum of the RGB values in $I[1]$, $O[2]$ by calculating the weighted sum of the RGB values in $I[2]$, and so on. None of these per-pixel computations depends on each other; all of them can be performed independently. Clearly the color-to-greyscale conversion exhibits a rich amount of data parallelism. Of course, data parallelism in complete applications can be more complex and much of this book is devoted to teaching the “parallel thinking” necessary to find and exploit data parallelism.

2.2 CUDA C PROGRAM STRUCTURE

We are now ready to learn to write a CUDA C program to exploit data parallelism for faster execution. The structure of a CUDA C program reflects the coexistence of a *host* (CPU) and one or more *devices* (GPUs) in the computer. Each CUDA source file can have a mixture of both host and device code. By default, any traditional C program is a CUDA program that contains only host code. One can add device functions and data declarations into any source file. The functions or data declarations for device are clearly marked with special CUDA C keywords. These are typically functions that exhibit rich amount of data parallelism.

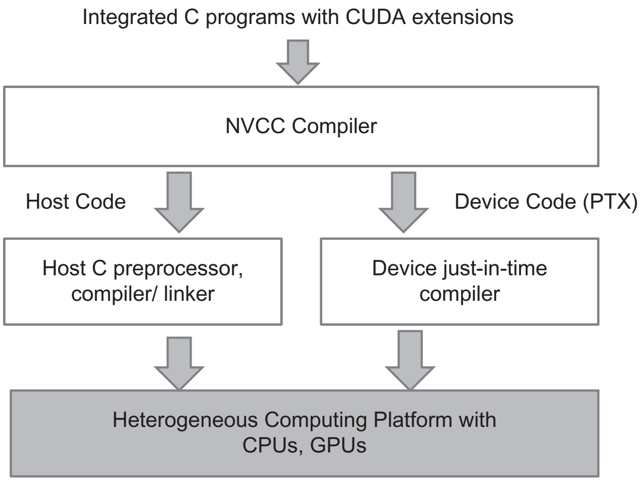

Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C preprocessor, compiler/ linker
Device just-in-time compiler
Heterogeneous Computing Platform with CPUs, GPUs

FIGURE 2.3

Overview of the compilation process of a CUDA C Program.

Once device functions and data declarations are added to a source file, it is no longer acceptable to a traditional C compiler. The code needs to be compiled by a compiler that recognizes and understands these additional declarations. We will be using a CUDA C compiler called NVCC (NVIDIA C Compiler). As shown at the top of Fig. 2.3, the NVCC compiler processes a CUDA C program, using the CUDA keywords to separate the host code and device code. The host code is straight ANSI C code, which is further compiled with the host's standard C/C++ compilers and is run as a traditional CPU process. The device code is marked with CUDA keywords for data parallel functions, called *kernels*, and their associated helper functions and data structures. The device code is further compiled by a run-time component of NVCC and executed on a GPU device. In situations where there is no hardware device available or a kernel can be appropriately executed on a CPU, one can also choose to execute the kernel on a CPU using tools like MCUDA [SSH 2008].

The execution of a CUDA program is illustrated in Fig. 2.4. The execution starts with host code (CPU serial code). When a kernel function (parallel device code) is called, or launched, it is executed by a large number of threads on a device. All the threads that are generated by a kernel launch are collectively called a *grid*. These threads are the primary vehicle of parallel execution in a CUDA platform. Fig. 2.4 shows the execution of two grids of threads. We will discuss how these grids are organized soon. When all threads of a kernel complete their execution, the corresponding grid terminates, the execution continues on the host until another kernel is launched. Note that Fig. 2.4 shows a simplified model where the CPU execution and the GPU execution do not overlap. Many heterogeneous computing applications actually manage overlapped CPU and GPU execution to take advantage of both CPUs and GPUs.

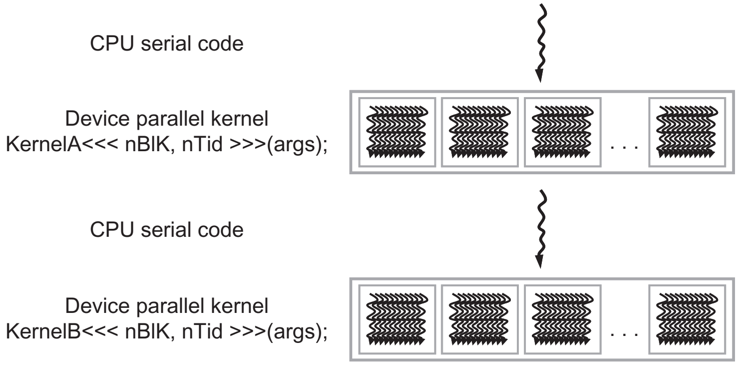


FIGURE 2.4

Execution of a CUDA program.

Launching a kernel typically generates a large number of threads to exploit data parallelism. In the color-to-greyscale conversion example, each thread could be used to compute one pixel of the output array *O*. In this case, the number of threads that will be generated by the kernel is equal to the number of pixels in the image. For large images, a large number of threads will be generated. In practice, each thread may process multiple pixels for efficiency. CUDA programmers can assume that these threads take very few clock cycles to generate and schedule due to efficient hardware support. This is in contrast with traditional CPU threads that typically take thousands of clock cycles to generate and schedule.

THREADS

A thread is a simplified view of how a processor executes a sequential program in modern computers. A thread consists of the code of the program, the particular point in the code that is being executed, and the values of its variables and data structures. The execution of a thread is sequential as far as a user is concerned. One can use a source-level debugger to monitor the progress of a thread by executing one statement at a time, looking at the statement that will be executed next and checking the values of the variables and data structures as the execution progresses.

Threads have been used in programming for many years. If a programmer wants to start parallel execution in an application, he/she creates and manages multiple threads using thread libraries or special languages. In CUDA, the execution of each thread is sequential as well. A CUDA program initiates parallel execution by launching kernel functions, which causes the underlying run-time mechanisms to create many threads that process different parts of the data in parallel.

2.3 A VECTOR ADDITION KERNEL

We now use vector addition to illustrate the CUDA C program structure. Vector addition is arguably the simplest possible data parallel computation, the parallel equivalent of "Hello World" from sequential programming. Before we show the kernel code for vector addition, it is helpful to first review how a conventional vector addition (host code) function works. Fig. 2.5 shows a simple traditional C program that consists of a main function and a vector addition function. In all our examples, whenever there is a need to distinguish between host and device data, we will prefix the names of variables that are processed by the host with "*h*_" and those of variables that are processed by a device "*d*_" to remind ourselves the intended usage of these variables. Since we only have host code in Fig. 2.5, we see only "*h*_" variables.

Assume that the vectors to be added are stored in arrays *A* and *B* that are allocated and initialized in the main program. The output vector is in array *C*, which is also allocated in the main program. For brevity, we do not show the details of how A, B, and C are allocated or initialized in the main function. The pointers (see sidebar below) to these arrays are passed to the vecAdd function, along with the variable *N* that contains the length of the vectors. Note that the formal parameters of the vectorAdd function are prefixed with "*h*_" to emphasize that these are processed by the host. This naming convention will be helpful when we introduce device code in the next few steps.

The vecAdd function in Fig. 2.5 uses a for-loop to iterate through the vector elements. In the *i*th iteration, output element *h*_C[*i*] receives the sum of *h*_A[*i*] and *h*_B[*i*]. The vector length parameter *n* is used to control the loop so that the number of iterations matches the length of the vectors. The formal parameters *h*_A, *h*_B and *h*_C are passed by reference so the function reads the elements of *h*_A, *h*_B and writes the elements of *h*_C through the argument pointers A, B, and C. When the

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (int i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

FIGURE 2.5

A simple traditional vector addition C code example.

```
#include <cuda.h>
…
void vecAdd(float* A, float* B, float* C, int n)
{
   int  size = n* sizeof(float);
   float  *d_A *d_B, *d_C;
   …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```

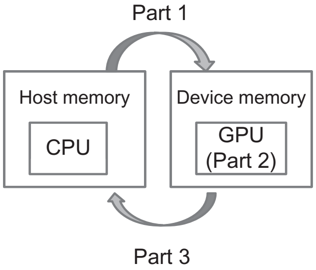


FIGURE 2.6

Outline of a revised vecAdd function that moves the work to a device.

vecAdd function returns, the subsequent statements in the main function can access the new contents of C.

A straightforward way to execute vector addition in parallel is to modify the vecAdd function and move its calculations to a device. The structure of such a modified vecAdd function is shown in Fig. 2.6. At the beginning of the file, we need to add a C preprocessor directive to include the cuda.h header file. This file defines the CUDA API functions and built-in variables (see sidebar below) that we will be introducing soon. Part 1 of the function allocates space in the device (GPU) memory to hold copies of the A, B, and C vectors and copies the vectors from the host memory to the device memory. Part 2 launches parallel execution of the actual vector addition kernel on the device. Part 3 copies the sum vector C from the device memory back to the host memory and frees the vectors in device memory.

POINTERS IN THE C LANGUAGE

The function arguments A, B, and C in Fig. 2.4 are pointers. In the C language, a pointer can be used to access variables and data structures. While a floating-point variable V can be declared with:

```
float V;
```

a pointer variable *P* can be declared with:

```
float *P;
```

By assigning the address of V to P with the statement P=&V, we make P "point to" V. *P becomes a synonym for V. For example U=*P assigns the value of V to U. For another example, *P=3 changes the value of V to 3.

An array in a C program can be accessed through a pointer that points to its 0th element. For example, the statement `P=&(A[0])` makes `P` point to the 0th element of array `A`. `P[i]` becomes a synonym for `A[i]`. In fact, the array name `A` is in itself a pointer to its 0th element.

In Fig. 2.5, passing an array name `A` as the first argument to function call to `vecAdd` makes the function's first parameter `h_A` point to the 0th element of `A`. We say that `A` is passed by reference to `vecAdd`. As a result, `h_A[i]` in the function body can be used to access `A[i]`.

See Patt&Patel [Patt] for an easy-to-follow explanation of the detailed usage of pointers in C.

Note that the revised `vecAdd` function is essentially an outsourcing agent that ships input data to a device, activates the calculation on the device, and collects the results from the device. The agent does so in such a way that the main program does not need to even be aware that the vector addition is now actually done on a device. In practice, such "transparent" outsourcing model can be very inefficient because of all the copying of data back and forth. One would often keep important bulk data structures on the device and simply invocate device functions on them from the host code. For now, we will stay with the simplified transparent model for the purpose of introducing the basic CUDA C program structure. The details of the revised function, as well as the way to compose the kernel function, will be shown in the rest of this chapter.

2.4 DEVICE GLOBAL MEMORY AND DATA TRANSFER

In current CUDA systems, devices are often hardware cards that come with their own dynamic random access memory (DRAM). For example, the NVIDIA GTX1080 comes with up to 8 GB[1] of DRAM, called global memory. We will use the terms global memory and device memory interchangeably. In order to execute a kernel on a device, the programmer needs to allocate global memory on the device and transfer pertinent data from the host memory to the allocated device memory. This corresponds to Part 1 of Fig. 2.6. Similarly, after device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed. This corresponds to Part 3 of Fig. 2.6. The CUDA run-time system provides API functions to perform these activities on behalf of the programmer. From this point on, we will simply say that

[1] There is a trend to integrate the address space of CPUs and GPUs into a unified memory space (Chapter 20). There are new programming frameworks such as GMAC that take advantage of the unified memory space and eliminate data copying cost.

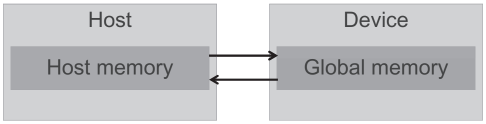


FIGURE 2.7

Host memory and device global memory.

a piece of data is transferred from host to device as shorthand for saying that the data is copied from the host memory to the device memory. The same holds for the opposite direction.

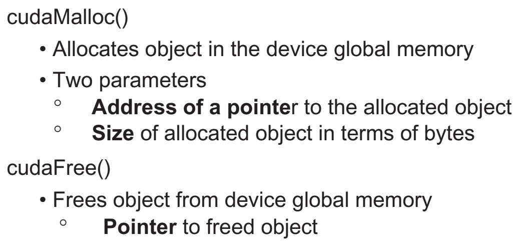
Fig. 2.7 shows a high level picture of the CUDA host memory and device memory model for programmers to reason about the allocation of device memory and movement of data between host and device. The device global memory can be accessed by the host to transfer data to and from the device, as illustrated by the bidirectional arrows between these memories and the host in Fig. 2.7. There are more device memory types than shown in Fig. 2.7. Constant memory can be accessed in a read-only manner by device functions, which will be described in Chapter 7, Parallel patterns: convolution. We will also discuss the use of registers and shared memory in Chapter 4, Memory and data locality. Interested readers can also see the CUDA programming guide for the functionality of texture memory. For now, we will focus on the use of global memory.

BUILT-IN VARIABLES

Many programming languages have built-in variables. These variables have special meaning and purpose. The values of these variables are often preinitialized by the run-time system and are typically read-only in the program. The programmers should refrain from using these variables for any other purposes.

In Fig. 2.6, Part 1 and Part 3 of the vecAdd function need to use the CUDA API functions to allocate device memory for A, B, and C, transfer A and B from host memory to device memory, transfer C from device memory to host memory at the end of the vector addition, and free the device memory for A, B, and C. We will explain the memory allocation and free functions first.

Fig. 2.8 shows two API functions for allocating and freeing device global memory. The cudaMalloc function can be called from the host code to allocate a piece of device global memory for an object. The reader should notice the striking similarity between cudaMalloc and the standard C run-time library malloc function. This is intentional; CUDA is C with minimal extensions. CUDA uses the standard C



cudaMalloc()

- Allocates object in the device global memory
- Two parameters
  - **Address of a pointe**r to the allocated object
  - **Size** of allocated object in terms of bytes

cudaFree()

- Frees object from device global memory
  - **Pointer** to freed object

FIGURE 2.8

CUDA API functions for managing device global memory.

run-time library malloc function to manage the host memory and adds cudaMalloc as an extension to the C run-time library. By keeping the interface as close to the original C run-time libraries as possible, CUDA minimizes the time that a C programmer spends to relearn the use of these extensions.

The first parameter to the cudaMalloc function is the **address** of a pointer variable that will be set to point to the allocated object. The address of the pointer variable should be cast to (void **) because the function expects a generic pointer; the memory allocation function is a generic function that is not restricted to any particular type of objects.[2] This parameter allows the cudaMalloc function to write the address of the allocated memory into the pointer variable.[3] The host code to launch kernels passes this pointer value to the kernels that need to access the allocated memory object. The second parameter to the cudaMalloc function gives the size of the data to be allocated, in number of bytes. The usage of this second parameter is consistent with the size parameter to the C malloc function.

We now use a simple code example to illustrate the use of cudaMalloc. This is a continuation of the example in Fig. 2.6. For clarity, we will start a pointer variable with letter "*d_*" to indicate that it points to an object in the device memory. The program passes the *address* of pointer *d*_A (i.e., &*d*_A) as the first parameter after casting it to a void pointer. That is, *d*_A will point to the device memory region allocated for the A vector. The size of the allocated region will be n times the size of a single-precision floating number, which is 4 bytes in most computers today. After the computation, cudaFree is called with pointer *d*_A as input to free the storage space for the A vector from the device global memory. Note that cudaFree does not need to

[2] The fact that cudaMalloc returns a generic object makes the use of dynamically allocated multidimensional arrays more complex. We will address this issue in Section 3.2.

[3] Note that cudaMalloc has a different format from the C malloc function. The C malloc function returns a pointer to the allocated object. It takes only one parameter that specifies the size of the allocated object. The cudaMalloc function writes to the pointer variable whose address is given as the first parameter. As a result, the cudaMalloc function takes two parameters. The two-parameter format of cudaMalloc allows it to use the return value to report any errors in the same way as other CUDA API functions.

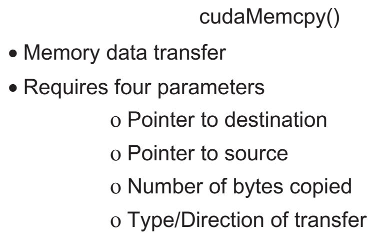
change the content of pointer variable *d*_A; it only needs to use the value of *d*_A to enter the allocated memory back into the available pool. Thus only the value, not the address of *d*_A, is passed as the argument.

```
float *d_A;
int size=n * sizeof(float);
cudaMalloc((void**)&d_A, size);
...
cudaFree(d_A);
```

The addresses in *d*_A, *d*_B, and *d*_C are addresses in the device memory. These addresses should not be dereferenced in the host code for computation. They should be mostly used in calling API functions and kernel functions. Dereferencing a device memory point in host code can cause exceptions or other types of run-time errors during execution.

The reader should complete Part 1 of the vecAdd example in Fig. 2.6 with similar declarations of *d*_B and *d*_C pointer variables as well as their corresponding cudaMalloc calls. Furthermore, Part 3 in Fig. 2.6 can be completed with the cudaFree calls for *d*_B and *d*_C.

Once the host code has allocated device memory for the data objects, it can request that data be transferred from host to device. This is accomplished by calling one of the CUDA API functions. Fig. 2.9 shows such an API function, cudaMemcpy. The cudaMemcpy function takes four parameters. The first parameter is a pointer to the destination location for the data object to be copied. The second parameter points to the source location. The third parameter specifies the number of bytes to be copied. The fourth parameter indicates the types of memory involved in the copy: from host memory to host memory, from host memory to device memory, from device memory to host memory, and from device memory to device memory. For example, the memory copy function can be used to copy data from one location of the device memory to another location of the device memory.[4]



cudaMemcpy()

- Memory data transfer
- Requires four parameters
    - o Pointer to destination
    - o Pointer to source
    - o Number of bytes copied
    - o Type/Direction of transfer

FIGURE 2.9

CUDA API function for data transfer between host and device.

[4] Please note cudaMemcpy currently cannot be used to copy between different GPU's in multi-GPU systems.

The vecAdd function calls the cudaMemcpy function to copy *h*_A and *h*_B vectors from host to device before adding them and to copy the *h*_C vector from the device to host after the addition is done. Assume that the values of *h*_A, *h*_B, *d*_A, *d*_B and size have already been set as we discussed before, the three cudaMemcpy calls are shown below. The two symbolic constants, cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost, are recognized, predefined constants of the CUDA programming environment. Note that the same function can be used to transfer data in both directions by properly ordering the source and destination pointers and using the appropriate constant for the transfer type.

```
cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
```

To summarize, the main program in Fig. 2.5 calls vecAdd, which is also executed on the host. The vecAdd function, outlined in Fig. 2.6, allocates device memory, requests data transfers, and launches the kernel that performs the actual vector addition. We often refer to this type of host code as a *stub function* for launching a kernel. After the kernel finishes execution, vecAdd also copies result data from device to the host. We show a more complete version of the vecAdd function in Fig. 2.10.

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE 2.10

A more complete version of `vecAdd()`.

ERROR CHECKING AND HANDLING IN CUDA

In general, it is very important for a program to check and handle errors. CUDA API functions return flags that indicate whether an error has occurred when they served the request. Most errors are due to inappropriate argument values used in the call.

For brevity, we will not show error checking code in our examples. For example, Fig. 2.10 shows a call to cudaMalloc:

```
cudaMalloc((void **) &d_A, size);
```

In practice, we should surround the call with code that test for error condition and print out error messages so that the user can be aware of the fact that an error has occurred. A simple version of such checking code is as follows:

```
cudaError_t err=cudaMalloc((void **) &d_A, size);
if (error !=cudaSuccess) {
  printf("%s in %s at line %d\n", cudaGetErrorString(err),__
  FILE__,__LINE__);
  exit(EXIT_FAILURE);
}
```

This way, if the system is out of device memory, the user will be informed about the situation. This can save many hours of debugging time.

One could define a C macro to make the checking code more concise in the source.

Compared to Fig. 2.6, the vecAdd function in Fig. 2.10 is complete for Part 1 and Part 3. Part 1 allocates device memory for *d*_A, *d*_B, and *d*_C and transfer *h*_A to *d*_A and *h*_B to *d*_B. This is done by calling the cudaMalloc and cudaMemcpy functions. The readers are encouraged to write their own function calls with the appropriate parameter values and compare their code with that shown in Fig. 2.10. Part 2 invokes the kernel and will be described in the following subsection. Part 3 copies the sum data from device memory to host memory so that their values will be available in the main function. This is accomplished with a call to the cudaMemcpy function. It then frees the memory for *d*_A, *d*_B, and *d*_C from the device memory, which is done by calls to the cudaFree function.

2.5 KERNEL FUNCTIONS AND THREADING

We are now ready to discuss more about the CUDA kernel functions and the effect of launching these kernel functions. In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase. Since all these threads execute the same code, CUDA programming is an instance of the well-known Single-Program

Multiple-Data (SPMD) [Ata 1998] parallel programming style, a popular programming style for massively parallel computing systems.[5]

When a program's host code launches a kernel, the CUDA run-time system generates a grid of threads that are organized into a two-level hierarchy. Each grid is organized as an array of thread blocks, which will be referred to as blocks for brevity. All blocks of a grid are of the same size; each block can contain up to 1024 threads. [6] Fig. 2.11 shows an example where each block consists of 256 threads. Each thread is represented by a curly arrow stemming from a box that is labeled with a number. The total number of threads in each thread block is specified by the host code when a kernel is launched. The same kernel can be launched with different numbers of threads at different parts of the host code. For a given grid, the number of threads in a block is available in a built-in blockDim variable.

The blockDim variable is of struct type with three unsigned integer fields: *x*, *y*, and *z*, which help a programmer to organize the threads into a one-, two-, or three-dimensional array. For a one-dimensional organization, only the *x* field will be used. For a two-dimensional organization, *x* and *y* fields will be used. For a three-dimensional structure, all three fields will be used. The choice of dimensionality for organizing threads usually reflects the dimensionality of the data. This makes sense since the threads are created to process data in parallel. It is only natural that the organization of the threads reflects the organization of the data. In Fig. 2.11, each thread block is organized as a one-dimensional array of threads because the data are one-dimensional vectors. The value of the blockDim.x variable specifies the total number of threads in each block, which is 256 in Fig. 2.11. In general, the number of

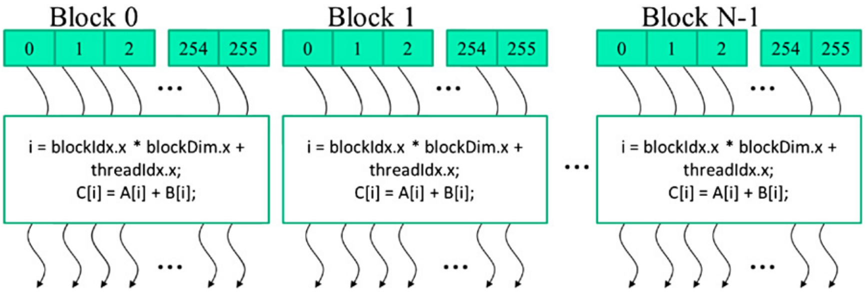

Block 0
0
1
2
254
255
Block 1
0
1
2
254
255
Block N-1
0
1
2
254
255
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
…

FIGURE 2.11

All threads in a grid execute the same kernel code.

[5] Note that SPMD is not the same as SIMD (Single Instruction Multiple-Data) [Flynn 1972]. In an SPMD system, the parallel processing units execute the same program on multiple parts of the data. However, these processing units do not need to be executing the same instruction at the same time. In an SIMD system, all processing units are executing the same instruction at any instant.

[6] Each thread block can have up to 1024 threads in CUDA 3.0 and beyond. Some earlier CUDA versions allow only up to 512 threads in a block.

threads in each dimension of thread blocks should be multiples of 32 due to hardware efficiency reasons. We will revisit this later.

CUDA kernels have access to two more built-in variables (threadIdx, blockIdx) that allow threads to distinguish among themselves and to determine the area of data each thread is to work on. Variable threadIdx gives each thread a unique coordinate within a block. For example, in Fig. 2.11, since we are using a one-dimensional thread organization, only threadIdx.x will be used. The threadIdx.x value for each thread is shown in the small shaded box of each thread in Fig. 2.11. The first thread in each block has value 0 in its threadIdx.x variable, the second thread has value 1, the third thread has value 2, etc.

The blockIdx variable gives all threads in a block a common block coordinate. In Fig. 2.11, all threads in the first block have value 0 in their blockIdx.x variables, those in the second thread block value 1, and so on. Using an analogy with the telephone system, one can think of threadIdx.x as local phone number and blockIdx.x as area code. The two together gives each telephone line a unique phone number in the whole country. Similarly, each thread can combine its threadIdx and blockIdx values to create a unique global index for itself within the entire grid.

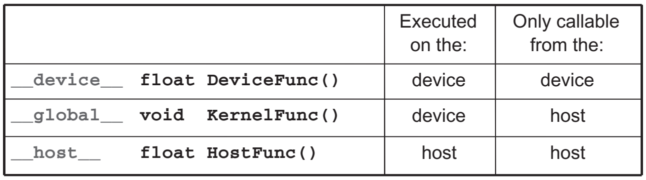
In Fig. 2.11, a unique global index i is calculated as $i =$ blockIdx.x*blockDim.x + threadIdx.x. Recall that blockDim is 256 in our example. The i values of threads in block 0 range from 0 to 255. The i values of threads in block 1 range from 256 to 511. The i values of threads in block 2 range from 512 to 767. That is, the i values of the threads in these three blocks form a continuous coverage of the values from 0 to 767. Since each thread uses i to access A, B, and C, these threads cover the first 768 iterations of the original loop. Note that we do not use the "*h_*" and "*d_*" convention in kernels since there is no potential confusion. We will not have any access to the host memory in our examples. By launching the kernel with a larger number of blocks, one can process larger vectors. By launching a kernel with n or more threads, one can process vectors of length n.

Fig. 2.12 shows a kernel function for vector addition. The syntax is ANSI C with some notable extensions. First, there is a CUDA C specific keyword "__global__" in front of the declaration of the vecAddKernel function. This keyword indicates that the function is a kernel and that it can be called from a host function to generate a grid of threads on a device.

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
   int i = blockDim.x*blockIdx.x + threadIdx.x;
   if(i<n) C[i] = A[i] + B[i];
}
```

FIGURE 2.12

A vector addition kernel function.



	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

FIGURE 2.13

CUDA C keywords for function declaration.

In general, CUDA C extends the C language with three qualifier keywords that can be used in function declarations. The meaning of these keywords is summarized in Fig. 2.13 The “__global__” keyword indicates that the function being declared is a CUDA C kernel function. Note that there are two underscore characters on each side of the word “global.” Such kernel function is to be executed on the device and can only be called from the host code except in CUDA systems that support *dynamic parallelism*, as we will explain in Chapter 13, CUDA dynamic parallelism. The “__device__” keyword indicates that the function being declared is a CUDA device function. A device function executes on a CUDA device and can only be called from a kernel function or another device function.[7]

The “__host__” keyword indicates that the function being declared is a CUDA host function. A host function is simply a traditional C function that executes on host and can only be called from another host function. By default, all functions in a CUDA program are host functions if they do not have any of the CUDA keywords in their declaration. This makes sense since many CUDA applications are ported from CPU-only execution environments. The programmer would add kernel functions and device functions during porting process. The original functions remain as host functions. Having all functions to default into host functions spares the programmer the tedious work to change all original function declarations.

Note that one can use both “__host__” and “__device__” in a function declaration. This combination tells the compilation system to generate two versions of object files for the same function. One is executed on the host and can only be called from a host function. The other is executed on the device and can only be called from a device or kernel function. This supports a common use case when the same function source code can be recompiled to generate a device version. Many user library functions will likely fall into this category.

The second notable extension to ANSI C, in Fig. 2.12, are the built-in variables “threadIdx.x” “blockIdx.x” and “blockDim.x”. Recall that all threads execute the same kernel code. There needs to be a way for them to distinguish among themselves and direct each thread towards a particular part of the data. These built-in variables

[7] We will explain the rules for using indirect function calls and recursions in different generations of CUDA later. In general, one should avoid the use of recursion and indirect function calls in their device functions and kernel functions to allow maximal portability.

are the means for threads to access hardware registers that provide the identifying coordinates to threads. Different threads will see different values in their threadIdx.x, blockIdx.x and blockDim.x variables. For simplicity, we will refer to a thread as thread$_{\text{blockIdx.x, threadIdx.x}}$. Note that the ".x" implies that there should be ".y" and ".z". We will come back to this point soon.

There is an automatic (local) variable *i* in Fig. 2.12. In a CUDA kernel function, automatic variables are private to each thread. That is, a version of i will be generated for every thread. If the kernel is launched with 10,000 threads, there will be 10,000 versions of *i*, one for each thread. The value assigned by a thread to its *i* variable is not visible to other threads. We will discuss these automatic variables in more details in Chapter 4, Memory and data locality.

A quick comparison between Figs. 2.5 and 2.12 reveals an important insight for CUDA kernels and CUDA kernel launch. The kernel function in Fig. 2.12 does not have a loop that corresponds to the one in Fig. 2.5. The readers should ask where the loop went. The answer is that the loop is now replaced with the grid of threads. The entire grid forms the equivalent of the loop. Each thread in the grid corresponds to one iteration of the original loop. This type of data parallelism is sometimes also referred to as *loop parallelism*, where iterations of the original sequential code are executed by threads in parallel.

Note that there is an if ($i < n$) statement in addVecKernel in Fig. 2.12. This is because not all vector lengths can be expressed as multiples of the block size. For example, let's assume that the vector length is 100. The smallest efficient thread block dimension is 32. Assume that we picked 32 as block size. One would need to launch four thread blocks to process all the 100 vector elements. However, the four thread blocks would have 128 threads. We need to disable the last 28 threads in thread block 3 from doing work not expected by the original program. Since all threads are to execute the same code, all will test their *i* values against *n*, which is 100. With the if ($i < n$) statement, the first 100 threads will perform the addition whereas the last 28 will not. This allows the kernel to process vectors of arbitrary lengths.

When the host code launches a kernel, it sets the grid and thread block dimensions via *execution configuration parameters*. This is illustrated in Fig. 2.14. The configuration parameters are given between the " <<<" and ">>>" before the traditional C function arguments. The first configuration parameter gives the number of thread blocks in the grid. The second specifies the number of threads in each thread block. In this example, there are 256 threads in each block. In order to ensure that we

```
int vectAdd(float* A, float* B, float* C, int n)
{
//  d_A, d_B, d_C allocations and copies omitted
// Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

FIGURE 2.14

A vector addition kernel launch statement.

have enough threads to cover all the vector elements, we apply the C ceiling function to *n*/256.0. Using floating-point value 256.0 ensures that we generate a floating value for the division so that the ceiling function can round it up correctly. For example, if we have 1000 threads, we would launch ceil(1000/256.0) = 4 thread blocks. As a result, the statement will launch 4*256 = 1024 threads. With the if ($i < n$) statement in the kernel as shown in Fig. 2.12, the first 1000 threads will perform addition on the 1000 vector elements. The remaining 24 will not.

2.6 KERNEL LAUNCH

Fig. 2.15 shows the final host code in the vecAdd function. This source code completes the skeleton in Fig. 2.6. Figs. 2.12 and 2.15 jointly illustrate a simple CUDA program that consists of both host code and a device kernel. The code is hardwired to use thread blocks of 256 threads each. The number of thread blocks used, however, depends on the length of the vectors (*n*). If *n* is 750, three thread blocks will be used. If *n* is 4000, 16 thread blocks will be used. If *n* is 2,000,000, 7813 blocks will be used. Note that all the thread blocks operate on different parts of the vectors. They can be executed in any arbitrary order. Programmers must not make any assumptions regarding execution order. A small GPU with a small amount of execution resources may execute only one or two of these thread blocks in parallel. A larger GPU may execute 64 or 128 blocks in parallel. This gives CUDA kernels scalability in execution speed with hardware, that is, same code runs at lower speed on small GPUs and

```
void vecAdd(float* A, float* B, float* C, int n)
{

  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc((void **) &d_A, size);
  cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
  cudaMalloc((void **) &d_B, size);
  cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

  cudaMalloc((void **) &d_C, size);

  vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

        // Free device memory for A, B, C
  cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE 2.15

A complete version of the host code in the vecAdd.function.

higher speed on larger GPUs. We will revisit this point later in Chapter 3, Scalable parallel execution.

It is important to point out again that the vector addition example is used for its simplicity. In practice, the overhead of allocating device memory, input data transfer from host to device, output data transfer from device to host, and de-allocating device memory will likely make the resulting code slower than the original sequential code in Fig. 2.5. This is because the amount of calculation done by the kernel is small relative to the amount of data processed. Only one addition is performed for two floating-point input operands and one floating-point output operand. Real applications typically have kernels where much more work is needed relative to the amount of data processed, which makes the additional overhead worthwhile. They also tend to keep the data in the device memory across multiple kernel invocations so that the overhead can be amortized. We will present several examples of such applications.

2.7 SUMMARY

This chapter provided a quick, simplified overview of the CUDA C programming model. CUDA C extends the C language to support parallel computing. We discussed an essential subset of these extensions in this chapter. For your convenience, we summarize the extensions that we have discussed in this chapter as follows:

FUNCTION DECLARATIONS

CUDA C extends the C function declaration syntax to support heterogeneous parallel computing. The extensions are summarized in Fig. 2.13. Using one of “__global__”, “__device__”, or “__host__”, a CUDA C programmer can instruct the compiler to generate a kernel function, a device function, or a host function. All function declarations without any of these keywords default to host functions. If both “__host__” and “_device__” are used in a function declaration, the compiler generates two versions of the function, one for the device and one for the host. If a function declaration does not have any CUDA C extension keyword, the function defaults into a host function.

KERNEL LAUNCH

CUDA C extends C function call syntax with kernel execution configuration parameters surrounded by <<< and >>>. These execution configuration parameters are only used during a call to a kernel function, or a kernel launch. We discussed the execution configuration parameters that define the dimensions of the grid and the dimensions of each block. The reader should refer to the CUDA Programming Guide [NVIDIA 2016] for more details of the kernel launch extensions as well as other types of execution configuration parameters.

BUILT-IN (PREDEFINED) VARIABLES

CUDA kernels can access a set of built-in, predefined read-only variables that allow each thread to distinguish among themselves and to determine the area of data each thread is to work on. We discussed the threadIdx, blockDim, and blockIdx variables in this chapter. In Chapter 3, Scalable parallel execution, we will discuss more details of using these variables.

RUN-TIME API

CUDA supports a set of API functions to provide services to CUDA C programs. The services that we discussed in this chapter are cudaMalloc(), cudaFree(), and cudaMemcpy() functions. These functions allocate device memory and transfer data between host and device on behalf of the calling program respectively. The reader is referred to the CUDA C Programming Guide for other CUDA API functions.

Our goal for this chapter is to introduce the core concepts of CUDA C and the essential CUDA C extensions to C for writing a simple CUDA C program. The chapter is by no means a comprehensive account of all CUDA features. Some of these features will be covered in the remainder of the book. However, our emphasis will be on the key parallel computing concepts supported by these features. We will only introduce enough CUDA C features that are needed in our code examples for parallel programming techniques. In general, we would like to encourage the reader to always consult the CUDA C Programming Guide for more details of the CUDA C features.

2.8 EXERCISES

1. If we want to use each thread to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to data index?

A. `i=threadIdx.x + threadIdx.y;`
B. `i=blockIdx.x + threadIdx.x;`
C. `i=blockIdx.x*blockDim.x + threadIdx.x;`
D. `i=blockIdx.x * threadIdx.x;`

2. Assume that we want to use each thread to calculate two (adjacent) elements of a vector addition. What would be the expression for mapping the thread/block indices to i, the data index of the first element to be processed by a thread?

A. `i=blockIdx.x*blockDim.x + threadIdx.x +2;`
B. `i=blockIdx.x*threadIdx.x*2;`
C. `i=(blockIdx.x*blockDim.x + threadIdx.x)*2;`
D. `i=blockIdx.x*blockDim.x*2 + threadIdx.x;`

3. We want to use each thread to calculate two elements of a vector addition. Each thread block processes 2*blockDim.x consecutive elements that form two sections. All threads in each block will first process a section first, each processing one element. They will then all move to the next section, each processing one element. Assume that variable *i* should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

A. `i=blockIdx.x*blockDim.x + threadIdx.x +2;`
B. `i=blockIdx.x*threadIdx.x*2;`
C. `i=(blockIdx.x*blockDim.x + threadIdx.x)*2;`
D. `i=blockIdx.x*blockDim.x*2 + threadIdx.x;`

4. For a vector addition, assume that the vector length is 8000, each thread calculates one output element, and the thread block size is 1024 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?

A. 8000
B. 8196
C. 8192
D. 8200

5. If we want to allocate an array of *v* integer elements in CUDA device global memory, what would be an appropriate expression for the second argument of the cudaMalloc call?

A. `n`
B. `v`
C. `n * sizeof(int)`
D. `v * sizeof(int)`

6. If we want to allocate an array of *n* floating-point elements and have a floating-point pointer variable *d*_A to point to the allocated memory, what would be an appropriate expression for the first argument of the cudaMalloc() call?

A. `n`
B. `(void *) d_A`
C. `*d_A`
D. `(void **) &d_A`

7. If we want to copy 3000 bytes of data from host array *h*_A (*h*_A is a pointer to element 0 of the source array) to device array *d*_A (*d*_A is a pointer to element 0 of the destination array), what would be an appropriate API call for this data copy in CUDA?

A. `cudaMemcpy(3000, h_A, d_A, cudaMemcpyHostToDevice);`
B. `cudaMemcpy(h_A, d_A, 3000, cudaMemcpyDeviceTHost);`
C. `cudaMemcpy(d_A, h_A, 3000, cudaMemcpyHostToDevice);`
D. `cudaMemcpy(3000, d_A, h_A, cudaMemcpyHostToDevice);`

8. How would one declare a variable err that can appropriately receive returned value of a CUDA API call?
   - **A.** `int err;`
   - **B.** `cudaError err;`
   - **C.** `cudaError_t err;`
   - **D.** `cudaSuccess_t err;`

9. A new summer intern was frustrated with CUDA. He has been complaining that CUDA is very tedious: he had to declare many functions that he plans to execute on both the host and the device twice, once as a host function and once as a device function. What is your response?

REFERENCES

Atallah, M. J. (Ed.). (1998). *Algorithms and theory of computation handbook*. Boca Raton: CRC Press.

Flynn, M. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, *C-21*, 948-–960.

NVIDIA Corporation. (2016). NVIDIA CUDA C Programming Guide, version 7.

Patt, Y. N., & Patel, S. J. (2003). *Introduction to computing systems: from bits and gates to C and beyond*. McGraw Hill Publisher.

Stratton, J. A., Stone, S. S., & Hwu, W. W. (2008). MCUDA: an Efficient Implementation of CUDA Kernels for Multi-Core CPUs. *The 21st international workshop on languages and compilers for parallel computing*, July 30–31, Canada, 2008. Also available as Lecture Notes in Computer Science.