COMP 322: Fundamentals of Parallel Programming
Module 3: Locality and Distribution

April 18, 2016

DRAFT VERSION – PLEASE DO NOT DISTRIBUTE

Contents

1 Task Affinity with Places

1.1 Hardware Memory Hierarchies

In today's lecture, we will revisit a basic question that arose when we discussed algorithmic complexity metrics *viz.,* what cost should we assume for a read/write access to a memory location? Historically, the answer from the Computer Science theory community has been that a memory read/write operation can be assumed to take constant time, but this theoretical position is increasingly at odds with real-world system trends.

Figure 1 shows an example memory hierarchy from the viewpoint of a single processor core. It is organized as a pyramid because the upper levels are smaller, faster, and more expensive per byte, where as the lower levels are larger, slower, and cheaper per byte. This hierarchy has 6 levels, and is typical of today's systems including your laptop and server processors such as the Rice STIC system. The number of hierarchy levels is expect to increase further in future systems, easily reaching 9 or 10 levels by the year, 2020. The word, *cache*, in "L1 cache" and "L2 cache" refers to a smaller and faster storage device that captures a subset of the data in the lower levels of the memory hierarchy.

Table 1 shows the latency parameters for a typical memory hierarchy. The fundamental idea of a memory hierarchy is to ensure that, for each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1. The reason why this approach works well in practice is because of the *principle of locality viz.,* programs tend to access the data at level k more often than they access the data at level k+1. Thus, the storage at level k+1 can be slower, larger and cheaper per bit, than level k. The ultimate goal is to create a large pool of storage with average cost per byte that approaches that of the cheap storage near the bottom of the hierarchy, and average latency that approaches that of fast storage near the top of the hierarchy.

The memory hierarchy becomes more complicated in the presence of parallelism. Figure 2 shows the structure of a single Intel Xeon Quad-core E5440 HarperTown processor chip; a single node in the STIC system contains 8 cores obtained by connecting two similar chips together to a shared memory. From this figure, we can see that not only is it beneficial to reuse data in upper levels of the hierarchy more frequently, but it can also be advantageous when tasks that share data execute on nearby cores. For example, if two tasks that share data run on core A and core B, then the data only needs to be fetched once into their shared L2 cache, instead of being fetched twice into two different L2 caches if the tasks ran on core A and core C (say). Such tasks that share data are said to have *affinity* with each other.

1.2 HJ's place construct

In Lecture 11, you were introduced to work-sharing and work-stealing schedulers that created one worker thread per core, and coordinated the scheduling of `async` tasks among worker threads. This model treats all workers as being interchangeable *i.e.,* any task could be executed on any thread. However, as discussed in Section 1.1 above, all cores are not interchangeable since it is preferable to execute tasks with mutual affinity on nearby cores. In fact, there is a natural trade-off between parallelism and affinity. From the parallelism viewpoint, it is desirable to run a task on the first available core (as in the "greedy" schedulers discussed in Lecture 11). From the affinity viewpoint, it is desirable to run a task T_i on a core close to cores where tasks with affinity to T_i have executed or are executing. Ideally, the runtime scheduler should migrate tasks and workers so as to co-locate tasks with affinity on nearby cores. But, this migration is also accompanied by its own overheads, and can be challenging for a runtime system to perform automatically.

The *place* construct in HJ provides a way for the programmer to specify affinity among async tasks. A place is an abstraction for a set of worker threads. When an HJ program is launched with the command, "`hj -places p:w`", a total of $p \times w$ worker threads are created with w workers per place. The places are numbered in the range $0 \ldots p-1$ and can be referenced in an HJ program, as described below. The number of places remains fixed during program execution; there is no construct to create a new place after the program is launched. This is consistent with other programming models, such as OpenMP, CUDA, and MPI, that require the number of threads/processes to be specified when an application is launched. However, the

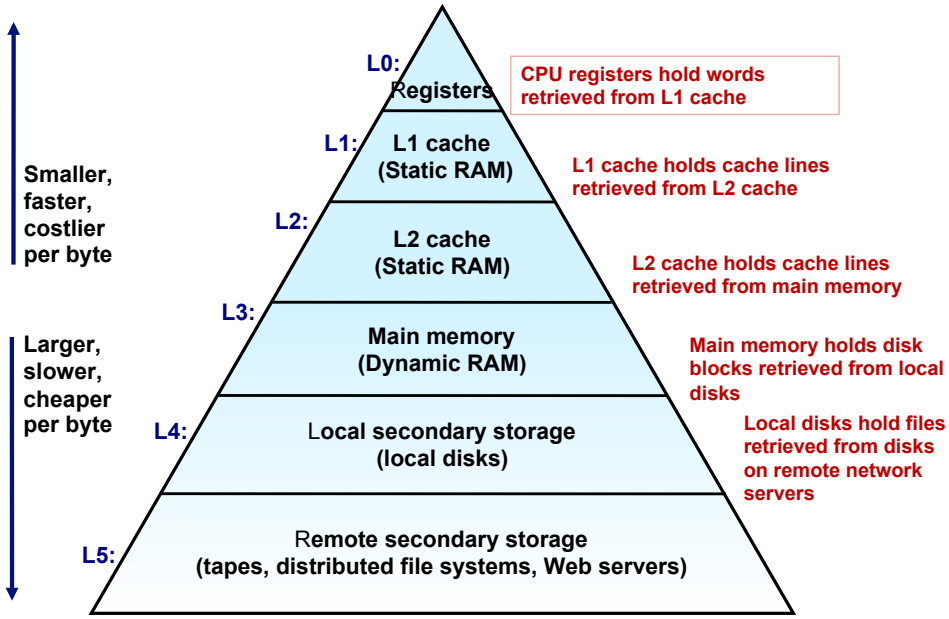

Figure 1: An example memory hierarchy from the viewpoint of a single processor core (source: [4])

Hierarchy Level	What is cached?	Where is it cached?	Latency (cycles)	Managed by
Registers	4-32 bytes (words)	CPU core	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware
L1 cache	64-bytes block	On-Chip L1	$O(10^0)$	Hardware
L2 cache	64-bytes block	On/Off-Chip L2	$O(10^1)$	Hardware
Virtual Memory	4 KB page	Main memory	$O(10^2)$	Hardware & OS
Buffer cache	Parts of files	Main memory	$O(10^2)$	OS
Disk cache	Disk sectors	Disk controller	$O(10^5)$	Disk firmware
Network buffer cache	Parts of files	Local disk	$O(10^7)$	AFS/NFS client
Browser cache	Web pages	Local disk	$O(10^7)$	Web browser
Web cache	Web pages	Remote server disks	$O(10^9)$	Web proxy server

Table 1: Examples of Caching in the Memory Hierarchy (source: [4])

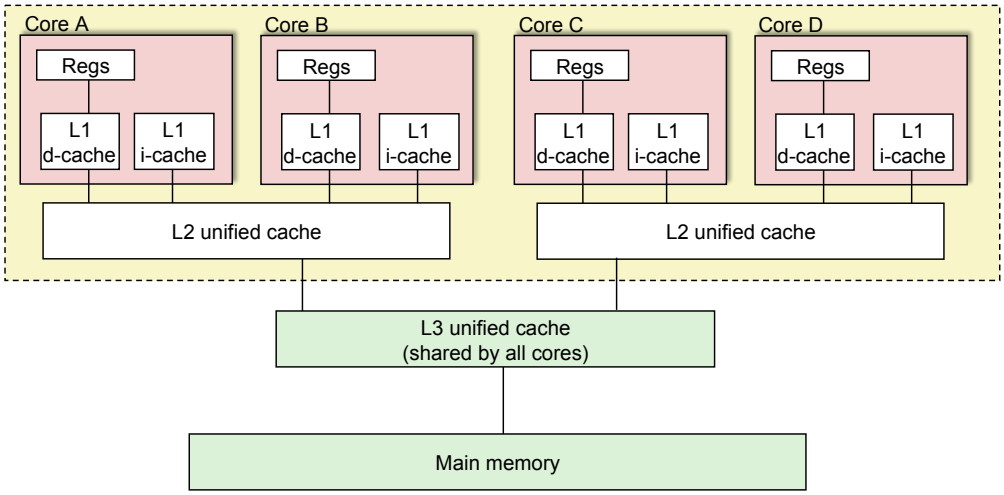

Figure 2: Memory hierarchy for a single Intel Xeon Quad-core E5440 HarperTown processor chip

management of individual worker threads within a place is not visible to an HJ program, giving the runtime system the freedom to create additional worker threads in a place, if needed, after starting with w workers per place.

The main benefit of use $p > 1$ places is that an optional `at` clause can be specified on an async statement or expression of the form, "`async at(`*place-expr*`) ...`", where *place-expr* is a place-valued expression. This clause dictates that the child async task can only be executed by a worker thread at the specified place. Data locality can be controlled by assigning two tasks with the same data affinity to execute in the same place. If the `at` clause is omitted, then the child task is scheduled by default to execute at the same place as its parent task. The main program task is assumed to start in place 0.

Thus, each task has a designated place. The value of a task's place can be retrieved by using the keyword, `here`. Since all your HJ programming thus far has been for a single place, all `async` tasks just ran at place 0 by default and there was no need for you to specify an `at` clause for any of them.

The `place` type is part of the `hj.lang.*` package, and can be used in any HJ program. The constant, `place.MAX_PLACES`, returns the number of places in the current program execution *i.e.,* the p value in the `-places p:w` option. A place-valued expression (*place-expr*) can be generated in one of the following ways:

1. The keyword, `here`, returns the place assigned to the currently executing task.
2. The method call, `place.factory.place(i)`, returns a reference to the place corresponding to integer expression `i`, which is expected to be in the range $0 \ldots$ `place.MAX_PLACES` $- 1$.
3. Any variable or field of type `place` returns a reference to a place.

Listing 1 shows an HJ program for which it could be beneficial to use places. Lines 1–7 define a class, `T1`, containing a field, `affinity`, of type `place`. The constructor, `T1()`, in line 5 initializes the `affinity` field to the place where the object was allocated by using the keyword, `here`. Later, the `for` loop in lines 11–16 iterates through a collection of objects of type `T1`. For each such object, `a`, a child `async` task is created to run at the place specified by `a.affinity`. Ideally the `a` object would still be accessible in L1 or L2 cache for the worker thread in place `a.affinity` that executes the child `async` task. Note that the `finish` construct works transparently across different places *i.e.,* `async` tasks in the same `finish` scope can be created at different places.

The print statement in line 10 prints the value of the designated place for the parent task. A predefined *place-expr*`.toString()` method generates a string representation for the place of the form, "`place(id=1)`".

The print statement in line 14 prints the designated place for the child task, which may be different from that of the parent task. For convenience, a predefined *place-expr*.`id` field is also available that returns the place identifier for a *place-expr* as an `int` value in the range 0...`place.MAX_PLACES` − 1.

```
1  class T1 {
2    final place affinity;
3    . . .
4    // T1's constructor sets affinity to place where instance was created
5    T1() { affinity = here; ... }
6    . . .
7  }
8  . . .
9  finish { // Inter-place parallelism
10   System.out.println("Parent place = ", here); // Parent task s place
11   for (T1 a = . . .) {
12     async at (a.affinity) { // Execute async at place with affinity to a
13       a.foo();
14       System.out.println("Child place = ", here); // Child task's place
15     } // async
16   } // for
17 } // finish
18 . . .
```

Listing 1: Example HJ program with places

The `-places p:w` option in the `hj` command is called a *place configuration*. The same program can be executed with different place configurations, even on the same hardware. For example, since a node in the STIC cluster contains 8 cores, it is reasonable to experiment with the following configurations on that machine — 1:8, 2:4, 4:2, and 8:1. A program such as the one outlined in Listing 1 would work correctly in all cases, but may exhibit different trade-offs between affinity and load balance for different configurations. The 1:8 configuration is best suited for load balance, since it contains a single place and it enables any of the 8 workers to execute any task. The 8:1 configuration is best suited for affinity, since it contains 8 places with 1 worker per place. A judicious choice of affinity values in the `at` clause could lead to better performance than the 1:8 case due to improved locality. However, a poor load imbalance (*e.g.,* if the longest-running tasks are all assigned to place 0) could lead to poor performance because a worker in a lightly-loaded place is unable to assist a worker in heavily-loaded place.

2 Distributions

There are many parallel programming languages that support the concept of mapping elements in a rectangular index space to places or "nodes" in a parallel computer. This mapping is referred to as a *distribution*. In HJ, distributions can be used to control the placement of *computations*. In some other languages (*e.g.,* Unified Parallel C [5]) distributions are used to control the placement of data. However, in both cases, the fundamental idea behind distributions as mappings is essentially the same.

Formally, an HJ distribution is a function from a *point* in a k-dimensional space to a place (with id in the range 0...`place.MAX_PLACES` − 1). While the programmer is free to create any data structure they choose to store these mappings, the HJ language provides a predefined type, `hj.lang.dist`, to simplify working with distributions. The following methods are available for any instance `d` of type `hj.lang.dist`:

- `d.rank` = number of dimensions in the input domain for distribution `d`.
- `d.places()` = set of places in the range of distribution `d`.
- `d.get(p)` = place for point `p` mapped by distribution `d`. It is an error to call `d.get(p)` if `p.rank` ≠ `d.rank`.

2.1 Block Distributions

The method call, `dist.factory.block([lo:hi])`, creates a *block distribution* over the one-dimensional region, `lo...hi`. A block distribution splits the region into contiguous subregions, one per place, while trying to keep the subregions as close to equal in size as possible. As an example, Table 2 shows the distribution created by the call, `dist.factory.block([0:15])`. If the input region is multidimensional, then a block distribution is computed over the linearized one-dimensional version of the multidimensional region as shown in Table 3. Block distributions can improve the performance of parallel loops that exhibit *spatial locality* across contiguous iterations.

Listing 2 shows the typical pattern used to iterate over an input region `r`, while creating one `async` task for each iteration `p` at the place dictated by distribution `d` *i.e.,* at place `d.get(p)`. An elegant property of this pattern that it works correctly regardless of the rank and contents of input region `r` and input distribution `d`.

```
1  finish {
2    region r = ... ; // e.g., [0:15] or [0:7,0:1]
3    dist d = dist.factory.block(r);
4    for (point p:r)
5      async at(d.get(p)) {
6        // Execute iteration p at place specified by distribution d
7        . . .
8      }
9  } // finish
10 . . .
```

Listing 2: Example HJ program with block distribution `d` on region `r`

2.2 Cyclic Distributions

The method call, `dist.factory.cyclic([lo:hi])`, creates a *cyclic distribution* over the one-dimensional region, `lo...hi`. A cyclic distribution "cycles" through places `0...place.MAX_PLACES` − 1 when spanning the input region, as shown in Tables 4 and Table 5. Cyclic distributions can improve the performance of parallel loops that exhibit load imbalance, as in Listing 3. In this example, the `async` task created at line 5 takes $O(i)$ time. If a `block` distribution was used, place 0 would be assigned the tasks with the shortest running times and place `place.MAX_PLACES-1` would be assigned the tasks with the largest running times. Instead, the `cyclic` distribution in Listing 3 ensures that the shortest and longest `async` tasks are evenly distributed across all the places.

```
1  finish {
2    region r = [0:n-1];
3    dist d = dist.factory.cyclic(r);
4    for (point[i]:r)
5      async at(d.get(i)) { // Takes O(i) time
6        for (int j = 0; j <= i; j++ {
7          . . .
8        }
9      }
10 } // finish
11 . . .
```

Listing 3: Example HJ program with cyclic distribution

The multidimensional distribution examples in Tables 3 and 5 repeated evenly across dimension boundaries. However, this need not be the case in general. As an example, Figure 3 shows a cyclic distribution for a 8×8 sized region (*e.g.,* `[1:8,1:8]`) mapped on to 5 places.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Place id	0				1				2				3			

Table 2: Block distribution mapping input region, `[0:15]`, to four place ids, 0 . . . 3

Index	[0,0]	[0,1]	[1,0]	[1,1]	[2,0]	[2,1]	[3,0]	[3,1]	[4,0]	[4,1]	[5,0]	[5,1]	[6,0]	[6,1]	[7,0]	[7,1]
Place id	0				1				2				3			

Table 3: Block distribution mapping input region, `[0:7,0:1]`, to four place ids, 0 . . . 3

2.3 Block-Cyclic Distributions

The method call, `dist.factory.blockCyclic([lo:hi],b)`, creates a *block-cyclic distribution* over the one-dimensional region, `lo...hi`. A block-cyclic distribution combines the locality benefits of the `block` distribution with the load-balancing benefits of the `cyclic` distribution by introducing a *block size* parameter, `b`. The linearized region is first decomposed into contiguous blocks of size `b`, and then the blocks are distributed in a cyclic manner across the places as shown in Table 6.

3 GPU Computing

3.1 Introduction to GPGPUs

Thus far, your experience with parallel programming has been with *general-purpose computing* cores as in SUG@R. These cores are referred to as CPUs (Central Processing Unit) cores because each core's CPU can perform any kind of computation, data access, and synchronization that you have learned in class. CPU cores are designed to maximize the performance of a single thread executing on a single CPU core. As shown in the left side of Figure 4, a single CPU core contains multiple Arithmetic Logic Units (ALUs) as well as a substantial Control unit (to enable "out-of-order" execution of sequential instructions) and a substantial *Cache* memory. Current mainstream quad-core processors have 4 such CPU cores in a single chip.

This design approach for CPUs is ill suited to executing computations with massive amounts of parallelism, as needed currently for video games and graphics applications and is expected to be needed by other applications in the future. To counter this problem, manufacturers such as NVIDIA and ATI have produced special-purpose processors called Graphics Processing Units (GPUs) to execute such computations. These processors are also referred to as "devices" because they were historically used to execute device driver software that controlled graphic display peripheral devices.

A GPU is designed with the assumption that the software can provide an abundant number of parallel threads that perform identical computations on different data operands. As shown on the right side of Figure 4, the space occupied by a single CPU core can be replaced by a much larger number of GPU processors. Each green box (labeled "A") on the GPU side corresponds to a simple ALU. Further a collection of ALU's in a single row share a single Control unit (labeled "Co") and a single cache memory (labeled "Ca"). As we will see, there are a number of restrictions on the capabilities of a GPU control unit and GPU cache that enable these components to be much smaller in a GPU than in a CPU core.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Place id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

Table 4: Cyclic distribution mapping input region, `[0:15]`, to four place ids, 0 . . . 3

Index	[0,0]	[0,1]	[1,0]	[1,1]	[2,0]	[2,1]	[3,0]	[3,1]	[4,0]	[4,1]	[5,0]	[5,1]	[6,0]	[6,1]	[7,0]	[7,1]
Place id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

Table 5: Cyclic distribution mapping input region, `[0:7,0:1]`, to four place ids, 0 . . . 3

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Place id	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3

Table 6: Block-cyclic distribution mapping input region, `[0:15]`, to four place ids, 0 . . . 3, with block size `b` = 2

The collection of ALU's, control unit, and cache memory in a single row in Figure 4 is referred to as a *Streaming Multiprocessor* (SM). It is also complicated to determine what should be called a "core" in a GPU. The common convention is to refer to a single ALU in an SM as a core. The number of cores in an SM, and the number of SM's in a GPU vary from model to model. For the NVIDIA Tesla C870 processor that you will work with in the lab (badlands.rcsg.rice.edu), there are 8 cores per SM and 16 SMs per device, resulting in 128 cores per device. The more recent NVIDIA Fermi GPGPU processor has 16 SMs per device with 32 cores per SM, resulting in 512 cores per device!

Until 2006, the software tool chains available for graphics chips only supported computer graphics APIs such as OpenGL and Direct3D. Since 2007, GPU vendors have enabled their processors to run more general classes of programs than graphics applications (albeit more restricted than the applications that can run on multicore CPUs). This led to the acronym, GPGPU, which stands for "General Purpose computing on a Graphics Processing Unit". A number of programming models have been developed for GPGPU programming including NVIDIA's CUDA model model [2, 9], AMD's Brook+ streaming model [3], and Khronos Group's OpenCL framework [1]. We will study CUDA in this course, but the principles you learn for CUDA should be applicable to other GPGPU programming models as well.

3.2 Introduction to CUDA

Acknowledgment: the text in this section was derived from references [10] and [6].

CUDA[1] is a highly popular language developed by NVIDIA for GPGPU programming. The CUDA programming model is an extension of the C language. Programmers write an application with two portions of code — functions to be executed on the CPU *host* and functions to be executed on the GPU *device*. The entry functions for the device code are tagged with a `__global__` keyword, and are referred to as *kernels*. A kernel executes in parallel across a set of parallel threads in a Single Instruction Multiple Thread (SIMT) model [9]. Since the host and device codes execute in two different memory spaces, the host code must include special calls for host-to-device and device-to-host data transfers.

Figure 5 shows the sequence of steps involved in a typical CUDA kernel invocation. The CUDA runtime library provides a collection of device memory management, host-device stream synchronization, and execution control functions (among others).

In the simplest case, the execution of a CUDA program can be viewed as a series of alternating phases of host (CPU) execution and device (GPGPU) execution as shown in Figure 6. Program execution starts in the host (CPU), moves to the device (GPGPU) when a kernel is invoked, and returns to the host (CPU) when the kernel completes. This structure can become more complicated when a single host is connected to multiple GPGPU devices and when host and device computations are overlapped with each other.

[1]The CUDA acronym originally stood for "Compute Unified Device Architecture" but now CUDA is used to refer to the programming model for such devices.

3.3 Logical Structure of threads in a CUDA kernel

As indicated earlier, a single execution of a CUDA kernel can result in the creation of a large number of threads on the device. These threads are logically equivalent to the HJ nested-forall loop shown below in Listing 4. Note that HJ does *not* currently execute on GPUs. (In fact, Java does not run on GPUs either.) However, we will use the code in Listing 4 to understand the logical parallelism in a CUDA kernel invocation before discussing lower-level syntax issues for using CUDA.

```
1  finish async at(GPU) {
2    // Parallel execution of blocks in grid
3    forall (point[blockIdx.x,blockIdx.y] : [0:gridDim.x-1,0:gridDim.y-1]) {
4      // Parallel execution of threads in block (blockIdx.x,blockIdx.y)
5      forall (point[threadIdx.x,threadIdx.y,threadIdx.z]
6                    : [0:blockDim.x-1,0:blockDim.y-1,0:blockDim.z-1]) {
7        // Perform kernel computation as function of blockIdx.x,blockIdx.y
8        // and threadIdx.x,threadIdx.y,threadIdx.z
9        . . .
10       next; // barrier synchronizes inner forall only (__syncthreads)
11       . . .
12     } // forall threadIdx.x,threadIdx.y,threadIdx.z
13   } // forall blockIdx.x, blockIdx.y
14 } // finish async (GPU)
```

Listing 4: Logical structure of a CUDA kernel invocation

Line 1 in Listing 4 uses the HJ `async at` construct to indicate that the CUDA kernel is executing at a different place form the parent task; specifically, it executes at a designated place called "GPU" which is intended to map to a GPU device. The use of `finish` in Line 1 indicates that the parent CPU task will wait for the GPU kernel to complete execution before moving to its next statement. The two levels of `forall`'s in Listing 4 mirror the two levels of inter-SM and intra-SM parallelism in current GPU's.

As shown in Figure 7, each kernel invocation is an instance of the `async at(GPU)` statement in line 1. A kernel is organized as a two-dimensional *grid* of blocks of size `gridDim.x` × `gridDim.y`. The `forall` loop in line 3 iterates through blocks with index variables (blockIdx.x,blockIdx.y). For each block, the `forall` loop in line 5 iterates through the individual "threads" with index variables (threadIdx.x,threadIdx.y,threadIdx.z). The computation performed by a single CUDA thread occurs in lines 6–10 of Listing 4. This code can reference individual components of the block and thread indices by accessing any of the fields, blockIdx.x ,blockIdx.y, threadIdx.x, threadIdx.y, and threadIdx.z.

Note that a `next` operation performed in the body of the inner `forall` loop (as in line 9) only synchronizes iterations of the inner forall. This matches the barrier functionality available in CUDA (called `__syncthreads`) that only synchronizes threads within the same block, but does not provide any inter-block synchronization. Note that if fewer dimensions are desired at the grid or block level, that can be accomplished by setting the appropriate dimension sizes to 1. For example if `gridDim.y` = 1 and `blockDim.y` = 1 and `blockDim.z` =1, then we effectively get a one-dimensional grid of blocks, each of which is a one-dimensional block of threads.

3.4 Matrix Multiply Example

Following [7], we will use a simple example to illustrate GPU programming with CUDA. Consider the sequential CPU version of the matrix multiply code shown in Figure 8. Though this code is written in C, it should be understandable to anyone familiar with Java. The only difference with Java is that, for the purpose of this example, a declaration like `float* M` in C should be considered equivalent to `float[] M` in Java.

The matrix multiply code in Figure 8 computes the product of two input matrices of size *Width*×*Width*, `M` and `N`, and stores the result in matrix `P` also of size *Width*×*Width*. It does so by using the following

mathematical identity:

$$P[i,j] = \sum_{0 \le k < Width} M[i,k] \times N[k,j]$$

Note that all three matrices are represented as one-dimensional arrays in the code. This is done to ensure that all rows within a matrix are stored contiguously in memory. However, as a result it becomes the programmer's responsibility to correctly map between matrix indices and array subscripts. For example, matrix element $M[i,k]$ is mapped to element `M[i*Width+k]` in the code. To convert this code to CUDA, we start by observing that the `i` and `j` loops in Figure 8 can be executed with `forall` semantics since they are logically parallel. There are multiple possible grid and block structures that can be created for this loop nest. For simplicity, we will just set `gridDim.x` = `gridDim.y` = 1 to obtain a single block with `blockDim.x` = `blockDim.y` = `Width`.

Figure 9 shows the computation performed by a single thread (kernel) for this grid/block structure. Since there is only one block, the code starts by extracting the `x` and `y` fields of `threadIdx` to obtain the `i` and `j` values for the given thread (stored in variables `tx` and `ty` in Figure 9). Array variables `Md`, `Nd`, and `Pd` represent copies of these arrays on the GPU device.

The kernel in Figure 9 only accounts for step 3 in Figure 5. The remaining steps are as follows:

- *Step 1: copy data from CPU memory to GPU memory.* This also includes allocation of GPU memory which can be accomplished on the CPU side by issuing a `cudaMalloc()` call. The actual data copy is accomplished by a call to `cudaMemCpy()` with a `cudaMemcpyHostToDevice` parameter.
- *Step 2: CPU instructs GPU to start Kernel.* CUDA uses a special syntax with `<<<...>>>` angle brackets, as shown in Figure 10, to launch a kernel. The first parameter, `dimGrid`, specifies the number of blocks and the shape of the grid. The second parameter, `dimBlock`, specifies a block with `blockDim.x` = `blockDim.y` = `Width`. The kernel launch corresponds to the `finish–async` invocation in line 1 of Listing 4. In addition, the `<<<...>>>` statement implicitly captures the two `forall` loops shown in Listing 4.
- *Step 4: Copy the results from GPU memory to CPU memory.* This is accomplished by a call to `cudaMemCpy()` with a `cudaMemcpyDeviceToHost` parameter.

References

[1] OpenCL. URL `http://www.khronos.org/opencl/`.

[2] NVIDIA CUDA Programming Guide, Version 3.0, February 2010. URL `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf`.

[3] AMD. ATI Stream Computing - Technical Overview. Technical report, AMD, 2008.

[4] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective, Second Edition*. Prentice Hall, 2010. URL `http://csapp.cs.cmu.edu/`.

[5] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003. ISBN 0471220485. URL `http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471220485,descCd-description.html`.

[6] Max Grossman, Alina Simion Sbîrlea, Zoran Budimlić, and Vivek Sarkar. Cnc-cuda: Declarative programming for gpus. In *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, October 2010.

[7] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010. ISBN 0123814723, 9780123814722.

[8] Calvin Lin and Lawrence Snyder. *Principles of Parallel Programming*. Addison-Wesley, 2009.

[9] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6:40–53, March 2008. ISSN 1542-7730. URL `http://doi.acm.org/10.1145/1365490.1365500`.

[10] Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 887–899, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03868-6. URL `http://dx.doi.org/10.1007/978-3-642-03869-3_82`.

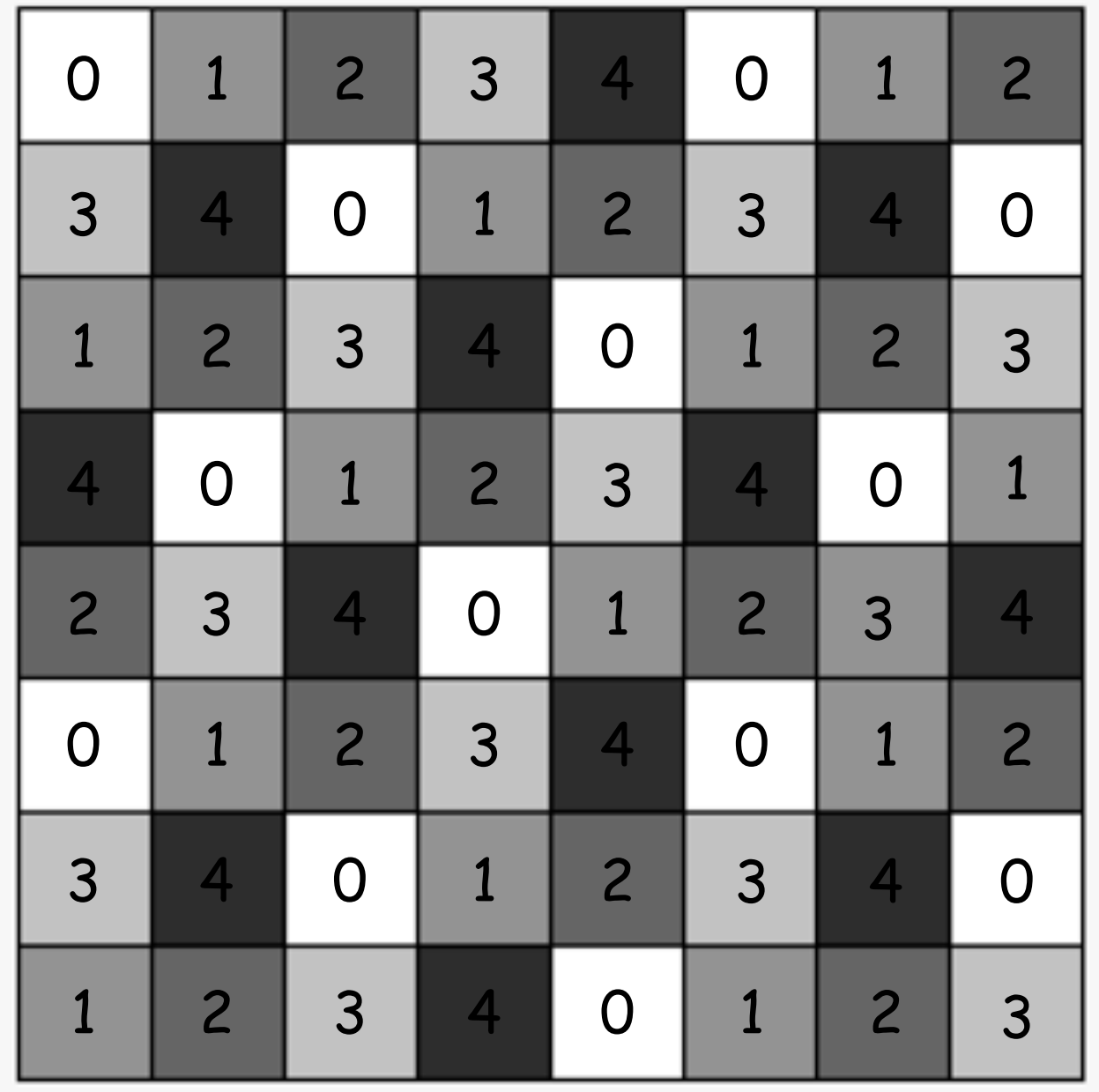

Figure 3: Illustration of a cyclic distribution for a 8×8 sized region (*e.g.,* `[1:8,1:8]`) mapped on to 5 places (source: Figure 5.12 in [8])

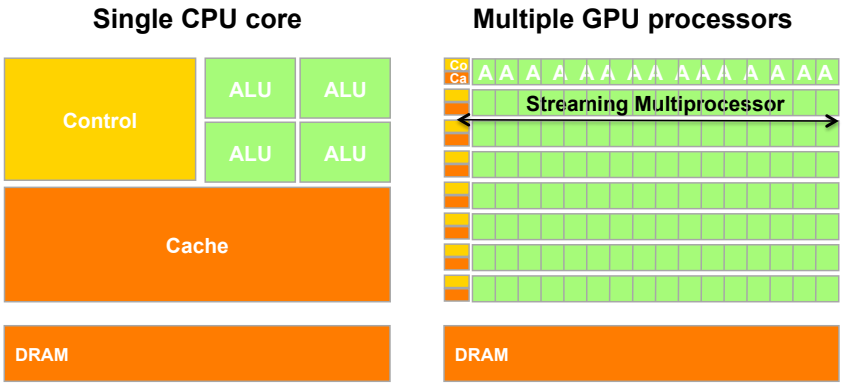

Figure 4: CPUs and GPUs have fundamentally different design philosophies (Source: Figure 1.2 in [7])

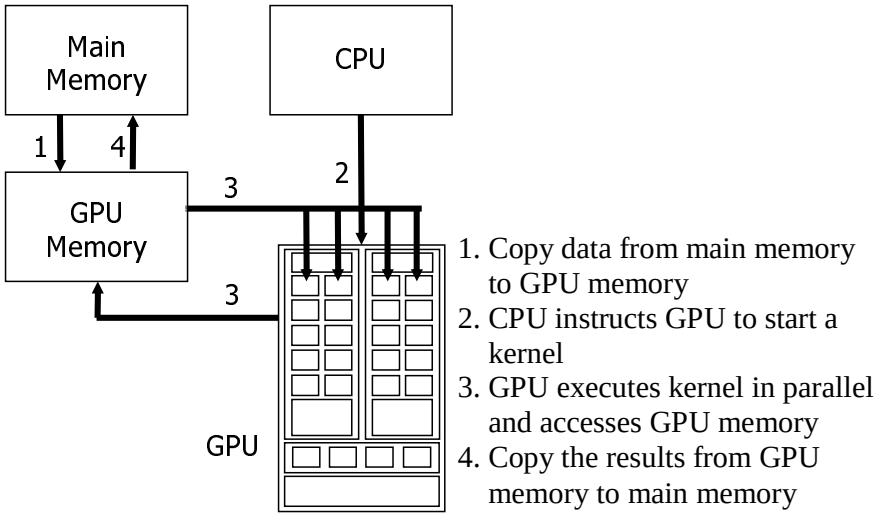

Figure 5: Process Flow of a CUDA Kernel Call (Source: Figure 1 in [10])

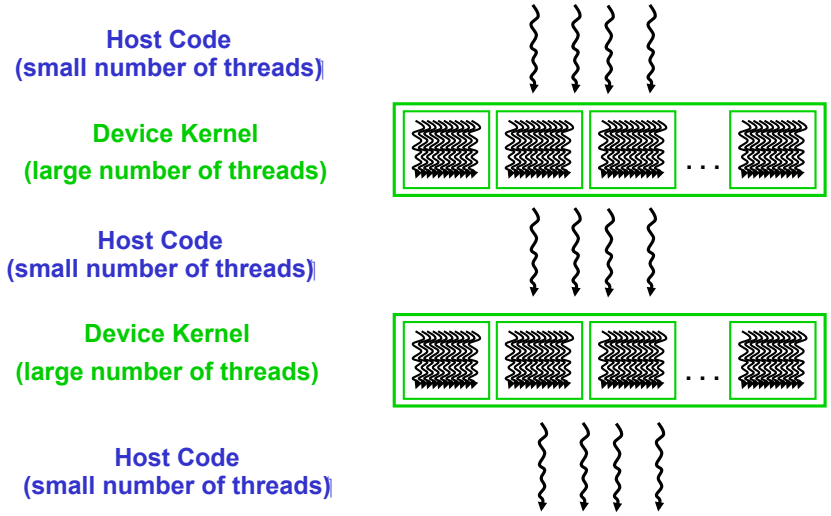

Figure 6: Execution of a CUDA program (Source: Figure 3.2 in [7])

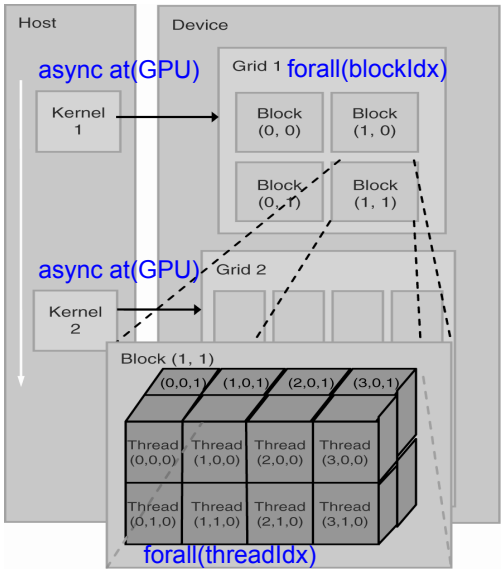

Figure 7: Organization of CUDA grid (Source: Figure 4.2 in [7])

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k  = 0; k < Width; ++k) {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

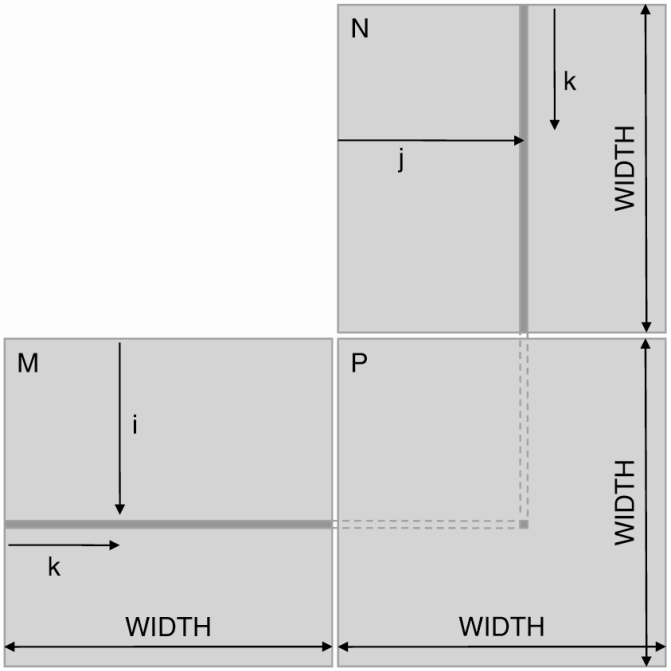

Figure 8: Sequential CPU version of matrix multiply written in C (Source: Figure 3.4 in [7])

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;

  for (int k = 0; k < Width; ++k)
  {
     float Mdelement = Md[ty * Width + k];
     float Ndelement = Nd[k * Width + tx];
     Pvalue += Mdelement * Ndelement;
  }

  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
```

Figure 9: The matrix multiplication kernel function (Source: Figure 3.11 in [7])

```
// Setup the execution configuration
dim3 dimBlock(Width, Width);
dim3 dimGrid(1, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Figure 10: CUDA code to launch an instance of the kernel in Figure 10