A Unified Compiler Backend for Distributed, Cooperative Heterogeneous Execution

by

Jessica Morgan Ray

B.S., University of Massachusetts Amherst (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

Author .....
Department of Electrical Engineering and Computer Science
January 31, 2018

Certified by .....
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chairwoman, Department Committee on Graduate Students

A Unified Compiler Backend for Distributed, Cooperative Heterogeneous Execution

by

Jessica Morgan Ray

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

In many of today's applications, achieving high-performance is critical. Numerous architectures, such as shared memory systems, distributed memory systems, and heteregeneous systems, are all used in high-performance computing. However, all of these have different optimizations and programming models. Additionally, performance is not always portable across architectures–one optimization that provides significant improvement in one system may have little to no effect in another system.

Writing high-performance code is a notoriously difficult task that requires significant trial-and-error. Writing high-performance code for multiple architectures is even harder, particularly when these architectural components are all together in a single heterogeneous system and the programmer has to make them all cooperate in order to achieve the highest performance. Hand-optimization only goes so far; it is infeasible to try many compositions of optimizations by hand, so the resulting performance will likely be sub-optimal. This thesis employs a scheduling language approach to abstract optimizations and code generation for shared memory with NUMA multicore, distributed memory, and GPU systems. To the best of our knowledge, we provide the first scheduling language approach that lets a programmer schedule cooperative execution on distributed, heterogeneous systems, all from a single algorithm. Our work extends an existing mid-level compiler, Tiramisu, with several primitives and functions that present the programmer with a unified interface for generating code for several backends and execution configurations.

Our results show that we are able to generate efficient MPI code and CUDA code for distributed memory and heterogeneous GPU systems from a single algorithm. From our unified scheduling abstraction, we are able to generate distributed, heterogeneous cooperative code, giving us OpenMP+MPI+CUDA capability without the extra complexities that come with using multiple programming models.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

In 2015, an MIT professor took a gamble on an incoming grad student who was interested in performance engineering and compilers. That student was worried that her lack of experience in those areas would be cause for concern, but that professor reassured her that she would learn everything she needed to along the way, and he took her as one his students. Of course, that student was me and that professor was (and is!) my fantastic advisor Saman Amarasinghe. There is no way I could've learned all I did over the past few years and do the work for this thesis without Saman's insight and guidance. Every step of the way, throughout the numerous project changes, dead ends, frustrations, and thankfully, the achievements, Saman has been there. I'm extremely grateful to have an advisor that genuinely cares about the progress of his graduate students. Thank you Saman–I'm very excited for the PhD adventure over the next *few* years!

Secondly, I want to thank the postdoc in my group, Riyadh Baghdadi. He is the author of Tiramisu, the compiler that this thesis builds from. He provided endless guidance and knowledge to me on Tiramisu, ISL, and Halide, and was there to answer all of my questions. This work would not have been possible without him!

I have a lot of individuals from MIT Lincoln Laboratory that I owe many thanks to as well. First off, I'm humbled by my acceptance into the Lincoln Scholars Program (LSP), and I want to thank the entire LSP committee for giving me this opportunity. It was difficult for me originally to leave a great place like MITLL when I went to grad school, so being able to come back through LSP is amazing. I'm looking forward to what the future brings me at MITLL! I also owe thanks to my group leaders at MITLL, Dr. Nick Malyska and Dr. Joe Campbell, for their help in preparing my LSP application, as well as for the opportunities they have provided for me during my time at MITLL. Additionally, thanks to Dr. Doug Sturim, Dr. Elizabeth Godoy, and Elizabeth Salesky from my group at MITLL for all their help in preparing my application (for both LSP and also grad school), and just helping me keep my sanity in general!

Of course, I can't forget to thank one of my professors from UMass, Professor Hanna Wallach. She has helped me with everything from getting me into MITLL to writing recommendations for my grad school applications, and even my LSP application.

Thanks to my amazing friend Whitney for answering my numerous questions about LSP, sympathizing with me over thesis struggles, cooking me dinner when I started getting stressed, and keeping me laughing throughout the whole process.

Obviously, thanks to all my friends for keeping me sane throughout! You all are the best friends I could have asked for!

And thanks to my wonderful boyfriend David. Your constant motivation and patience was more helpful in getting me through this process than I could ever put into words. I'm looking forward to endless hours of watching Family Guy and Game of Thrones with you once this thesis is done :)

Finally, how do I even begin to thank my parents? My parents have always been incredibly supportive through all of my endeavors, and I'm so lucky to have them. They've always been there for me to keep me positive even when I felt like giving up, and I often look to them as the voice of reason when it seems like everything is getting out of hand. Thank you for the unconditional love and support you've provided to me my whole life. Also thank you for letting me bring my laundry home almost every weekend.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, the tech industry has seen a proliferation of different computing architectures, each touting different benefits such as performance, memory availability, programming simplicity, cost, etc. For example, in 2007, NVIDIA released their CUDA [30] software package, which made programming GPUs feasible for applications other than graphics, thus bringing their GPU architecture into the high performance computing world. GPUs offer the potential for high throughput of massively parallel applications thanks to their high number of cores. Later on in 2012, Intel released the Xeon Phi co-processor [8], followed by the Knights Landing processor [50] in 2013. These CPU chips contain many more cores than previous Intel multi-core chips, providing greater throughput for operations at the cost of reduced speeds of individual cores (similar to the GPU). Combining different architectures, such as these, results in large, interconnected heterogeneous systems with distributed multi-socket CPUs, GPUs, and possibly even other compute devices like FPGAs. One of the world's fastest heterogeneous supercomputers, the TSUBAME 3.0 from Japan, is equipped with over 2,160 interconnected Tesla P100 GPUs and 15,120 Intel Xeon E5-2680 v4 cores [52]. Trying to write code that utilizes even a small subset of the available compute devices on a machine such as this is a significant undertaking.

Individual instances of these architectures are hard enough to generate code for, let alone optimize them. As we will see in section 1.0.1, combining the systems only increases the difficulty of writing the code. There are numerous factors to consider

when optimizing code, as everything from the application itself to the architecture to the programming language can impact performance. It is not even guaranteed that an optimization will help; sometimes, an optimization that improves performance on its own may end up hurting performance when composed with other optimizations. Optimization is largely a trial-and-error process, and it is infeasibile to manually try a large number of different compositions of optimizations, particularly across different architectures where code portability is very limited. This often leads to code that does not reach its full performance potential.

In this thesis, we focus on abstracting multi-socket/distributed CPU and heterogeneous architectures and present programmers with a simpler interface for programming and optimizing these systems. In particular, we look at the following architectures and configurations:

Shared memory: A shared memory architecture consists of processors and memories sharing an interconnect network. The processors all share a global memory space and communicate and synchronize with each other via load and store operations to the global memory [44]. The physical memory can be separate, giving a non-uniform memory access (NUMA) system, where the load and store operations' performance varies based on the processor's location to the memory.

Distributed memory: A distributed memory architecture is a collection of local memories and processors connected with an interconnect network. Unlike shared memory, there is no global memory, so explicit data movement is needed between the individual local memories [44].

Heterogeneous: A heterogeneous architecture contains different types of processors with different execution models, such as a combination of CPUs and GPUs. This is opposed to a homogeneous system that contains only one type of processor, such as just CPUs [60]. In a heterogeneous programming model, computations run on one type of processor at a time. For example, in a CPU-GPU heterogeneous model, a computation can run in parallel across multiple

CPUs or multiple GPUs, but it would not execute on both the CPUs and GPUs simultaneously.

Cooperative: Cooperative computing is a programming model for a heterogeneous system. In this model, a given part of an application is partitioned to execute on the various processor types simultaneously, thus maximizing the set of computing resources utilized [56]. This is opposed to the heterogeneous model described above, where part of the application runs on either the CPU or GPU, but not both.

Beyond actual code optimizations, these large interconnected systems present other difficulties such as load balancing and communication, to name a few. Unbalanced workloads lead to poor performance since not all the resources are being utilized; communication is required for individual nodes to synchronize with each other, but is very costly, so it needs to be carefully implemented. Achieving the full compute potential of these large heterogeneous systems should be a priority, especially in cooperative configurations; as problem sizes increase, we will need as much compute power as possible. We cannot rely much longer on Moore's Law to provide us with faster processors, so improving performance is going to rely heavily on the power of cooperative execution on distributed and heterogeneous systems. We utilize the idea of separating the *algorithm* from the actual *implementation* in our work, abstracting the underlying hardware and looking at optimizations as transformations on top of the algorithm. Taking advantage of prior work on TIRAMISU, a mid-level compiler for shared memory optimizations, we implement primitives and functions that expose distributed, heterogeneous, and cooperative code generation and optimization to the programmer through a high-level scheduling language. These optimizations describe transformations on the loop nests, so the programmer does not have to change the algorithm when trying a new optimization, they just have to select a different scheduling function. Our system can generate code for shared memory, distributed memory, and heterogeneous systems, as well as code for cooperative execution on heterogeneous systems. The type of code generated is based on the schedule applied

by the programmer.

Our work makes it simpler for a programmer to explore the tradeoff space between each of these systems and optimize code appropriately. The trial-and-error process is still there–it is inherent to optimization; however, our work keeps the programmer from having to make significant changes to the structure of their code just to try out an optimization.

How to design a completely unified representation for programing and optimizing these systems is still very much an open research question. However, this thesis provides building blocks for achieving that representation. We look at similarities and differences between architectures and programming models with the goal of understanding what set of features should be exposed to the programmer. In particular, we focus on the tradeoff between computation and communication, where communication means explicitly transferring data between different memories. The cost of communication may, for example, outweigh the benefits of GPU execution, so this tradeoff space needs to be explored. This work exposes communication directly to the programmer using a set of abstractions atop the underlying communication paradigm. This allows programmers to optimize communication along side the computations and maximize the overall performance of their application code.

1.0.1 A Motivating Example

To demonstrate how much an implementation can change based on the architecture and optimizations, we provide a few handcoded examples. The application here is a 2-dimensional box blur stencil. This is a simple stencil commonly used in image processing applications. The stencil computation for the blur is shown in 1-1. It computes an average of all the pixels within the stencil and then slides one column over and does the same computation until the whole image has been processed.

The basic single-threaded code for a box blur looks like the code in figure 1-2. This code is a direct translation of the blur algorithm in figure 1-1. For high-performance code, we would not want to write the code this way, however. It would be too slow.

Figure 1-3 shows the resulting code if we apply parallelization and vectorization

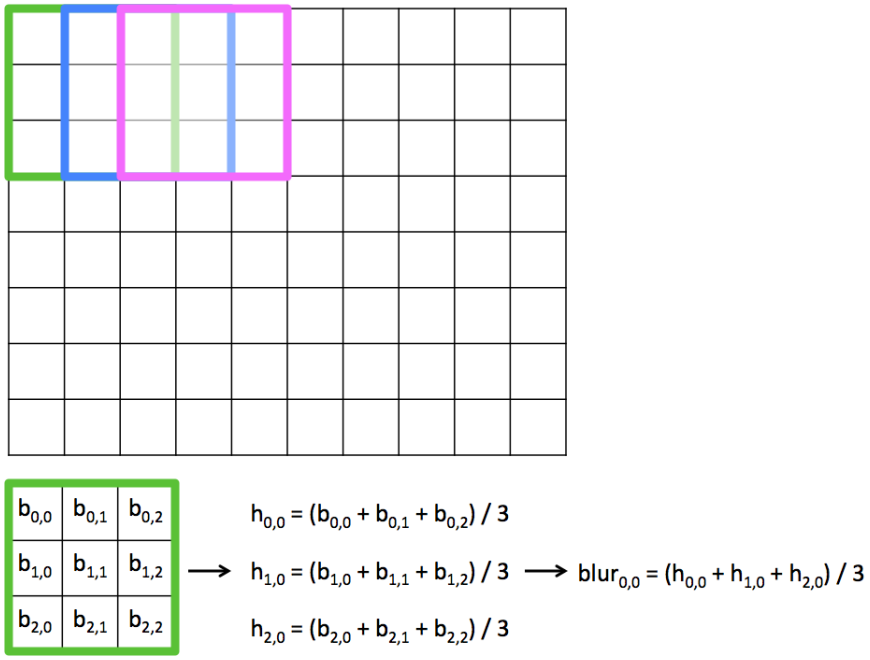


Figure 1-1: 2D blur example showing three consecutive applications of the stencil, as well as the computation for the blur of the upper left pixel.

```
1  ...
2  void blur() {
3    for (unsigned long r = 0; r < rows+2; r++) {
4      for (unsigned long c = 0; c < cols; c++) {
5        temp[r*cols+c] = (input[r*cols+c] + input[r*cols+c+1] +
             input[r*cols+c+2]) / 3.0f;
6      }
7    }
8    for (unsigned long r = 0; r < rows; r++) {
9      for (unsigned long c = 0; c < cols; c++) {
10       output[r*cols+c] = (temp[r*cols+c] + temp[(r+1)*cols+c] +
             temp[(r+2)*cols+c]) / 3.0f;
11     }
12   }
13 }
14 ...
```

Figure 1-2: Simple single-threaded CPU-only version of a 2D blur.

```
1  void blur_par_vec() {
2    omp_set_num_threads(20);
3    __m128 div_by = _mm_set_ps1(3.0f);
4  #pragma omp parallel for schedule(static, std::max((int)(rows/20), 1))
5    for (ulong r = 0; r < rows+2; r++) {
6      for (ulong c = 0; c < cols; c+=4) {
7        __m128 input_load_0 = _mm_loadu_ps(&input[r*cols+c]);
8        __m128 input_load_1 = _mm_loadu_ps(&input[r*cols+c+1]);
9        __m128 input_load_2 = _mm_loadu_ps(&input[r*cols+c+2]);
10       __m128 sum = _mm_add_ps(_mm_add_ps(input_load_0, input_load_1), input_load_2);
11       __m128 res = _mm_div_ps(sum, div_by);
12       _mm_storeu_ps(&temp[r*cols+c], res);
13     }
14   }
15 #pragma omp parallel for schedule(static, std::max((int)(rows/20), 1))
16   for (ulong r = 0; r < rows; r++) {
17     for (ulong c = 0; c < cols; c+=4) {
18       __m128 temp_load_0 = _mm_loadu_ps(&temp[r*cols+c]);
19       __m128 temp_load_1 = _mm_loadu_ps(&temp[(r+1)*cols+c]);
20       __m128 temp_load_2 = _mm_loadu_ps(&temp[(r+2)*cols+c]);
21       __m128 sum = _mm_add_ps(_mm_add_ps(temp_load_0, temp_load_1), temp_load_2);
22       __m128 res = _mm_div_ps(sum, div_by);
23       _mm_storeu_ps(&output[r*cols+c], res);
24     }
25   }
26 }
```

Figure 1-3: Multi-threaded and vectorized 2D blur.

(discussed in section 2.1.2). These are two common shared memory optimizations. In this code, the algorithm is still the same–we use two loop nests to compute the two parts of the blur, but the implementation has changed. There are more levels in the loop nests and the loop bodies now use vector instructions.

Next, we implement the blur to execute on the GPU. Figure 1-4[1] shows code for running the blur on a single GPU. It is still the same algorithm, but the implementation has completely changed from the previous versions. The two loop nests become the two GPU functions shown at the top, and then calling those functions adds additional code. If we switch to a cooperative setting, the code becomes even more complicated. Figure 1-5 shows code that runs cooperatively on the CPU and GPU for the blur. We need multiple versions of the code depending on the architecture, and if any change is made to the algorithm, such as adding a third dimension to the blur, each version of the code would need to be changed independently, unnecessarily adding additional burden the programmer.

Using our scheduling language approach described throughout this thesis, we can generate the same code with much less effort. Figure 1-6 shows our syntax for creating the initial blur algorithm. We then apply scheduling to that algorithm to describe the optimizations and guide the code generation. Figure 1-7 shows the schedule

[1]Error checking and some linking functions for the CUDA code excluded for brevity.

```
1  __global__
2  void blur_cuda_async_kernel_bx(float *input, float *bx, ulong *_cols, ulong *
       _current_row) {
3    ulong cols = *_cols;
4    ulong current_row = *_current_row;
5    int block_size = 32;
6    bx[current_row*cols+blockIdx.x*block_size+threadIdx.x] = (input[current_row*(cols
         +2)+blockIdx.x*block_size+threadIdx.x] + input[current_row*(cols+2)+blockIdx.x*
         block_size+threadIdx.x+1] + input[current_row*(cols+2)+blockIdx.x*block_size+
         threadIdx.x+2]) / 3.0f;
7  }
8  __global__
9  void blur_cuda_async_kernel_by(float *bx, float *by, ulong *_cols, ulong *
       _current_row) {
10   ulong cols = *_cols;
11   ulong current_row = *_current_row;
12   int block_size = 32;
13   by[current_row*cols+blockIdx.x*block_size+threadIdx.x] = (bx[current_row*cols+
         blockIdx.x*block_size+threadIdx.x] + bx[current_row*cols+blockIdx.x*block_size+
         threadIdx.x+cols] + bx[current_row*cols+blockIdx.x*block_size+threadIdx.x+2*
         cols]) / 3.0f;
14 }
15 void blur_gpu() {
16 ...
17   CUdeviceptr d_input, d_by, d_bx;
18   cuMemAlloc(&d_input, (this->rows+2) * (this->cols+2) * sizeof(float));
19   cuMemAlloc(&d_bx, this->rows * this->cols * sizeof(float));
20   cuMemAlloc(&d_by, this->rows * this->cols * sizeof(float));
21   CUdeviceptr d_kernel_vals_0, d_kernel_vals_1;
22   cuMemAlloc(&d_kernel_vals_0, this->rows * sizeof(ulong));
23   cuMemAlloc(&d_kernel_vals_1, this->rows * sizeof(ulong));
24   CUstream h2d_stream, d2h_stream, kernel_stream;
25   cuStreamCreate(&h2d_stream, CU_STREAM_DEFAULT);
26   cuStreamCreate(&d2h_stream, CU_STREAM_DEFAULT);
27   cuStreamCreate(&kernel_stream, CU_STREAM_DEFAULT);
28   for (ulong r = 0; r < this->rows; r++) {
29     cuMemcpyHtoDAsync(d_input + r * (this->cols+2) * sizeof(float), &input[r * (this
           ->cols+2)], this->cols * sizeof(float), h2d_stream);
30     cuMemcpyHtoDAsync(d_kernel_vals_0+r*sizeof(ulong), &cols, sizeof(ulong),
           kernel_stream);
31     cuMemcpyHtoDAsync(d_kernel_vals_1+r*sizeof(ulong), &r, sizeof(ulong),
           kernel_stream);
32     CUevent h2d_event;
33     cuEventCreate(&h2d_event, 0);
34     cuEventRecord(h2d_event, h2d_stream);
35     cuStreamWaitEvent(kernel_stream, h2d_event, 0);
36     CUdeviceptr k0 = d_kernel_vals_0+r*sizeof(ulong);
37     CUdeviceptr k1 = d_kernel_vals_1+r*sizeof(ulong);
38     void *kernel_args_bx[] = {&d_input, &d_bx, &k0, &k1};
39     cuLaunchKernel(bx_kernel, this->cols / 5, 1, 1, 5, 1, 1, 0, kernel_stream,
           kernel_args_bx, 0);
40   }
41   for (ulong r = 0; r < this->rows; r++) {
42     cuMemcpyHtoDAsync(d_kernel_vals_0+r*sizeof(ulong), &cols, sizeof(ulong),
           kernel_stream);
43     cuMemcpyHtoDAsync(d_kernel_vals_1+r*sizeof(ulong), &r, sizeof(ulong),
           kernel_stream);
44     CUdeviceptr k0 = d_kernel_vals_0+r*sizeof(ulong);
45     CUdeviceptr k1 = d_kernel_vals_1+r*sizeof(ulong);
46     void *kernel_args_by[] = {&d_bx, &d_by, &k0, &k1};
47     cuLaunchKernel(by_kernel, this->cols / 5, 1, 1, 5, 1, 1, 0, kernel_stream,
           kernel_args_by, 0);
48     CUevent kernel_event;
49     cuEventCreate(&kernel_event, 0);
50     cuEventRecord(kernel_event, kernel_stream);
51     cuStreamWaitEvent(d2h_stream, kernel_event, 0);
52     CUdeviceptr kd_by = d_by + r * (this->cols) * sizeof(float);
53     cuMemcpyDtoHAsync(&by[r * this->cols], kd_by, this->cols * sizeof(float),
           d2h_stream);
54   }
55   cuCtxSynchronize();
56 ...
57 }
```

Figure 1-4: Executing the blur on the GPU.

```
__global__
void blur_cuda_async_kernel_bx(float *input, float *bx, ulong *_cols, ulong *_current_row) {
  ulong cols = *_cols;
  ulong current_row = *_current_row;
  int block_size = 32;
  bx[current_row*cols+blockIdx.x*block_size+threadIdx.x] = (input[current_row*(cols+2)+blockIdx.x*block_size+
       threadIdx.x] + input[current_row*(cols+2)+blockIdx.x*block_size+threadIdx.x+1] + input[current_row*(cols
       +2)+blockIdx.x*block_size+threadIdx.x+2]) / 3.0f;
}
__global__
void blur_cuda_async_kernel_by(float *bx, float *by, ulong *_cols, ulong *_current_row) {
  ulong cols = *_cols;
  ulong current_row = *_current_row;
  int block_size = 32;
  by[current_row*cols+blockIdx.x*block_size+threadIdx.x] = (bx[current_row*cols+blockIdx.x*block_size+threadIdx.x
       ] + bx[current_row*cols+blockIdx.x*block_size+threadIdx.x+cols] + bx[current_row*cols+blockIdx.x*
       block_size+threadIdx.x+2*cols]) / 3.0f;
}
void blur_cooperative {
...
  int provided = -1;
  MPI_Init_thread(NULL, NULL, MPI_THREAD_FUNNELED, &provided);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int total_procs = NUM_CPU_PROCS + NUM_GPU_PROCS;
  if (rank <= NUM_CPU_PROCS) {
    omp_set_num_threads(20);
    __m128 div_by = _mm_set_ps1(3.0f);
#pragma omp parallel for schedule(static, std::max((int)((rows/total_procs)/20), 1))
    for (ulong r = 0; r < (rows/total_procs)+2; r++) {
      for (ulong c = 0; c < cols; c+=4) {
        __m128 input_load_0 = _mm_loadu_ps(&input[r * cols + c]);
        __m128 input_load_1 = _mm_loadu_ps(&input[r * cols + c + 1]);
        __m128 input_load_2 = _mm_loadu_ps(&input[r * cols + c + 2]);
        __m128 sum = _mm_add_ps(_mm_add_ps(input_load_0, input_load_1), input_load_2);
        __m128 res = _mm_div_ps(sum, div_by);
        _mm_storeu_ps(&temp[r * cols + c], res);
      }
    }
#pragma omp parallel for schedule(static, std::max((int)((rows/total_procs)/20), 1))
    for (ulong r = 0; r < (rows/total_procs); r++) {
      for (ulong c = 0; c < cols; c+=4) {
        __m128 temp_load_0 = _mm_loadu_ps(&temp[r * cols + c]);
        __m128 temp_load_1 = _mm_loadu_ps(&temp[(r + 1) * cols + c]);
        __m128 temp_load_2 = _mm_loadu_ps(&temp[(r + 2) * cols + c]);
        __m128 sum = _mm_add_ps(_mm_add_ps(temp_load_0, temp_load_1), temp_load_2);
        __m128 res = _mm_div_ps(sum, div_by);
        _mm_storeu_ps(&output[r * cols + c], res);
      }
    }
  } else { // GPU
    CUdeviceptr d_input, d_by, d_bx;
    cuMemAlloc(&d_input, (rows/total_procs + 2) * (cols + 2) * sizeof(float));
    cuMemAlloc(&d_bx, rows/total_procs * cols * sizeof(float));
    cuMemAlloc(&d_by, rows/total_procs * cols * sizeof(float));
    CUdeviceptr d_kernel_vals_0, d_kernel_vals_1;
    cuMemAlloc(&d_kernel_vals_0, rows/total_procs * sizeof(ulong));
    cuMemAlloc(&d_kernel_vals_1, rows/total_procs * sizeof(ulong));
    CUstream h2d_stream, d2h_stream, kernel_stream;
    cuStreamCreate(&h2d_stream, CU_STREAM_DEFAULT);
    cuStreamCreate(&d2h_stream, CU_STREAM_DEFAULT);
    cuStreamCreate(&kernel_stream, CU_STREAM_DEFAULT);
    for (ulong r = 0; r < this->rows; r++) {
      cuMemcpyHtoDAsync(d_input + r * (this->cols+2) * sizeof(float), &input[r * (this->cols+2)], this->cols *
           sizeof(float), h2d_stream);
      cuMemcpyHtoDAsync(d_kernel_vals_0+r*sizeof(ulong), &cols, sizeof(ulong), kernel_stream);
      cuMemcpyHtoDAsync(d_kernel_vals_1+r*sizeof(ulong), &r, sizeof(ulong), kernel_stream);
      CUevent h2d_event;
      cuEventCreate(&h2d_event, 0);
      cuEventRecord(h2d_event, h2d_stream);
      cuStreamWaitEvent(kernel_stream, h2d_event, 0);
      CUdeviceptr k0 = d_kernel_vals_0+r*sizeof(ulong);
      CUdeviceptr k1 = d_kernel_vals_1+r*sizeof(ulong);
      void *kernel_args_bx[] = {&d_input, &d_bx, &k0, &k1};
      cuLaunchKernel(bx_kernel, this->cols / 5, 1, 1, 5, 1, 1, 0, kernel_stream, kernel_args_bx, 0);
    }
    for (ulong r = 0; r < this->rows; r++) {
      cuMemcpyHtoDAsync(d_kernel_vals_0+r*sizeof(ulong), &cols, sizeof(ulong), kernel_stream);
      cuMemcpyHtoDAsync(d_kernel_vals_1+r*sizeof(ulong), &r, sizeof(ulong), kernel_stream);
      CUdeviceptr k0 = d_kernel_vals_0+r*sizeof(ulong);
      CUdeviceptr k1 = d_kernel_vals_1+r*sizeof(ulong);
      void *kernel_args_by[] = {&d_bx, &d_by, &k0, &k1};
      cuLaunchKernel(by_kernel, this->cols / 5, 1, 1, 5, 1, 1, 0, kernel_stream, kernel_args_by, 0);
      CUevent kernel_event;
      cuEventCreate(&kernel_event, 0);
      cuEventRecord(kernel_event, kernel_stream);
      cuStreamWaitEvent(d2h_stream, kernel_event, 0);
      CUdeviceptr kd_by = d_by + r * (this->cols) * sizeof(float);
      cuMemcpyDtoHAsync(&by[r * this->cols], kd_by, this->cols * sizeof(float), d2h_stream);
    }
    cuCtxSynchronize();
    cuMemFree(d_input);
    cuMemFree(d_by);
    cuMemFree(d_bx);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
}
```

Figure 1-5: Executing the blur cooperatively on the CPUs and GPUs.

for generating parallelized and vectorized code. In figure 1-8, we see the code for generating GPU code. Finally, we have the cooperative code. These schedules largely remain the same; we only need a few additional scheduling commands, shown in figure 1-9. With these minor modifications, we get OpenMP+CUDA+MPI code with much less code than the hand-written counterpart.

```
void blur_tiramisu() {
  function f;
  var r("r"), c("c");
  computation blur_input("{blur_input[r,c]: 0<=r<" + srows + " and 0<=c<" + scols + "}", expr(), false, p_float32
      , &f);
  expr bx_expr = (blur_input(r, c) + blur_input(r, c + 1) + blur_input(r, c + 2)) / 3.0f;
  computation bx("{bx[r,c]: 0<=r<" + srows + " and 0<=c<" + scols + "}", bx_expr, true, p_float32, &f);
  expr by_expr = (bx(r, c) + bx(r + 1, c) + bx(r + 2, c)) / 3.0f;
  computation by("{by[r,c]: 0<=r<" + srows + " and 0<=c<" + scols + "}", by_expr, true, p_float32, &f);
}
```

Figure 1-6: Describing the blur algorithm with our approach.

```
var r0("r0"), r1("r1"), c0("c0"), c1("c1");
bx.split(r, rows/20, r0, r1); by.split(r, rows/20, r0, r1);
by.tag_parallel_level(r0); bx.tag_parallel_level(r0);
bx.split(c, 8, c0, c1); by.split(c, 8, c0, c1);
by.tag_vector_level(c1, 8); bx.tag_vector_level(c1, 8);
bx.before(by, root);
```

Figure 1-7: Vectorizing and parallelizing code with our approach.

```
xfer_prop h2d_cuda_async(p_float32, {ASYNC, CUDA, CPU2GPU}, 1);
xfer_prop d2h_cuda_async(p_float32, {ASYNC, CUDA, GPU2CPU}, 2);
xfer_prop d2h_cuda_async_kernel_stream(p_float32, {ASYNC, CUDA, GPU2CPU}, 0);
xfer_prop h2d_cuda_async_kernel_stream(p_float32, {ASYNC, CUDA, CPU2GPU}, 0);
xfer h2d = computation::create_xfer("{h2d[r,c]: 0<=r<" + srows + " and 0<=c<2+" + scols + "}", h2d_cuda_async,
     blur_input(r,c), &blur);
xfer d2h = computation::create_xfer("{d2h[r,c]: 0<=r<" + srows + " and 0<=c<" + scols + "}", d2h_cuda_async, by(r
     ,0), &blur);
tiramisu::wait h2d_wait("{h2d_wait[r,c]: 0<=r<" + srows + " and 0<=c<1}", h2d.os(r,0),
     h2d_cuda_async_kernel_stream, true, &blur);
tiramisu::wait d2h_wait_for_by("{d2h_wait_for_by[r,c]: 0<=r<" + srows + " and 0<=c<1}", by(r, 0), d2h_cuda_async,
      true, &blur);
tiramisu::wait d2h_wait("{d2h_wait[r,c]: 0<=r<" + srows + " and 0<=c<1}", d2h.os(r,0), h2d_cuda_async, true, &
     blur);
generator::update_producer_expr_name(bx, "blur_input", "h2d", false);
h2d.os->collapse_many({collapser(1, 0, COLS+2)});
d2h.os->collapse_many({collapser(1, 0, COLS)});
bx.split(c, BLOCK_SIZE, c0, c1); by.split(c, BLOCK_SIZE, c0, c1);
d2h_wait.before(*h2d.os, r);
h2d.os->before(h2d_wait, r);
h2d_wait.before(bx, r);
bx.before(by, root);
by.before(d2h_wait_for_by, r);
d2h_wait_for_by.before(*d2h.os, r);
bx.tag_gpu_level2(c0, c1, 0); by.tag_gpu_level2(c0, c1, 0);
```

Figure 1-8: Generating GPU code with necessary communication with our approach.

```
// Apply to the CPU and GPU
bx.split(r, rows/PROCS, q1, q2); by.split(r, rows/PROCS, q1, q2);
bx.tag_distribute_level(q1); by.tag_distribute_level(q1);
// Apply to the GPU only
blur.rank_offset = -1*PROCS;
```

Figure 1-9: Additional scheduling commands needed to generate cooperative heterogeneous code.

Chapter 2

System Architecture Background

2.1 Shared Memory Systems

A *shared memory* system is made up of multiple processors and memories that are connected by an interconnect network. The individual processors in the network communicate with each other implicitly via load and store operations between the processor's memories. These systems, while easier to prgram than their distributed and heterogeneous counterparts, have a myriad of optimization opportunities available, and optimizing shared memory code is still an active area of research.

2.1.1 Architecture and Execution Overview

In a shared memory system, all the processors share the same address space, meaning all the processors have access to the same memory. Any processor can access the main memory, as well as the memory in the private caches of other processors, using *load* and *store* operations. The underlying hardware handles all communication implicitly between the different memories and the cores, which is in contrast to *distributed memory systems* discussed in section 2.2. However, programmer intervention is needed to achieve high-performance, as the hardware can only optimize so much. An example configuration of a shared memory multicore system is depicted in figure 2-1. This figure is based off of the Intel Skylake architecture [49].

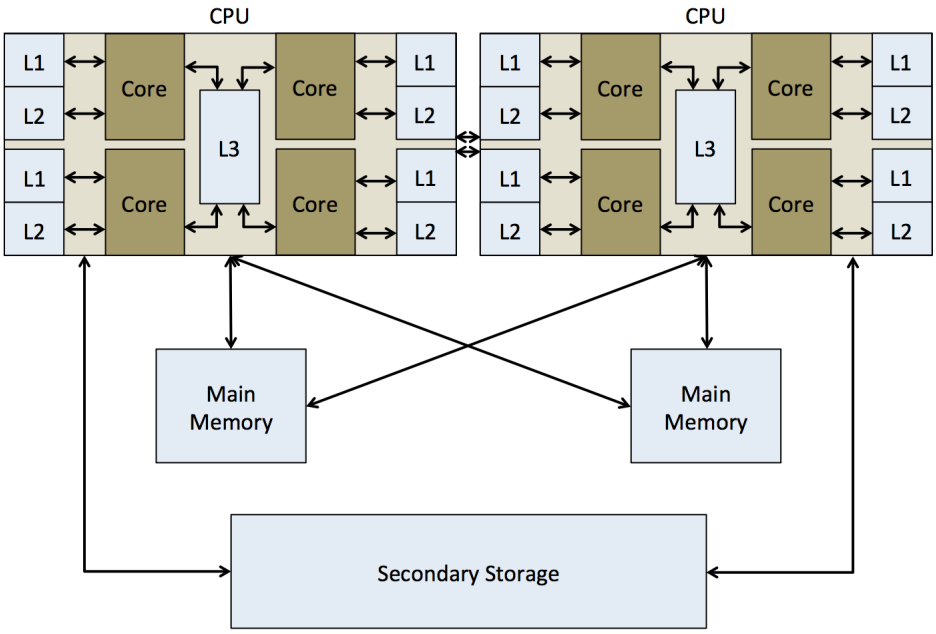


Figure 2-1: Shared memory multicore system.

This architecture shows a dual-socket configuration with four *cores* per *socket* (each socket takes one *CPU*, and each CPU contains four cores). Each socket has the same configuration. Within a socket, each core has its own *L1 cache* (both instruction and data), as well as its own *L2 cache*. The *L3 cache*, also known as the *last-level cache*, is shared by each of the cores in a socket. The caches and cores on an individual socket communicate across a bus. The two sockets are connected by some other high-speed interconnect. For example, in the case of the Intel Skylake architecture, up to 8 different sockets can be connected with Intel Ultra Path Interconnect (UPI) [27]. UPI is a low-latency, power-efficient interconnect for connecting Intel CPUs. Each socket connects to the main memory and secondary memories through another interconnect.

Memory is configured as a hierarchy, and each level presents a tradeoff between memory size and access *latency*. Latency is the time required to complete an event, such as a memory access [19]. Table 2.1 shows a typical memory hierarchy for a personal shared memory system, along with the average memory sizes and access latencies. At the top of this hierarchy, there are the registers, which have the lowest latency. As you move down the hierarchy, the latency increases, with secondary memory having the highest latency. However, the size of the memory increases as

Memory	Avg. Size	Avg. Access Latency
Registers	1000B	300ps
L1	64KB	1ns
L2	256KB	10ns
L3	4MB	20ns
RAM	16GB	100ns
Hard Drive	16TB	10ms

Table 2.1: Average sizes and access latencies for a typical shared memory system [48].

you move down the hierarchy. Ideally, all the necessary data for a process would be located in caches, but that is not always possible, particularly for programs with large working set sizes.

In figure 2-1, notice that the main memory is separated into two individual chunks. Even though a shared memory system has a single address space, the underlying physical memory may be separate across the CPUs. This is type of configuration is referred to *non-uniform memory access* (NUMA), or *distributed shared memory*. A simpler view of a NUMA system is in figure 2-2. In this setup, cores can access addresses in their part of the distributed memory faster than the other parts, so they have non-uniform access latency. This is in contrast to a *uniform memory access* (UMA) configuration where the physical memory is in one centralized location. With an UMA system, all cores have the same memory access latency for a given address, hence the uniform access. UMA can be used with single socket systems that have a small number of cores. As the number of cores increase, there is too much contention on the centralized memory, severely impacting performance [19]. A simple UMA configuration is depicted in figure 2-3.

Accessing Data in a Shared Memory System

When a load or a store is issued by a core in a shared memory system, the hardware performs several steps to access the data and get it to the requesting core. Though this is handled automatically by the hardware, a programmer can influence the performance of these steps based on how they write their program. The first step the hardware takes is locating the data, which begins by checking the core's caches. If the

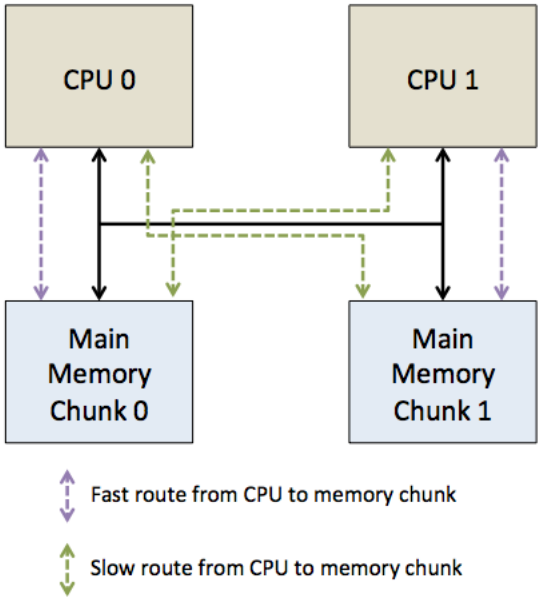


Figure 2-2: NUMA (non-uniform memory access) configuration.

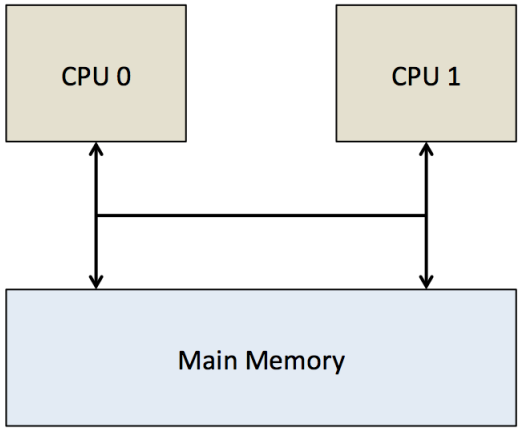


Figure 2-3: UMA (uniform memory access) configuration.

data is not found in the cache, the hardware will check the caches of the other cores. If the address is not found in any of the caches, then the main memory is checked. If the address is not found in main memory, then secondary memory is checked, and so on. Each access is progressively more expensive, so the further into the memory hierarchy the hardware searches, the higher the access latency.

Once the data is located, it needs to be moved to the appropriate location for the requesting core. For loads, this typically means moving the data to the requesting core's L1 cache. For stores, the data may be first moved to the L1 cache (if it is a *write-back* cache), or it may be left in the current location (for a *write-through cache*) [44]. If the data is already in the L1 cache, then there is nothing to be done (this is the ideal case). If the data is located somewhere else, then it is moved into the L1 cache, and possibly also the L2 and L3 cache, depending on their configuration.

Data is not moved as a single byte. When data is moved from secondary memory, the data is moved in whole *pages*, where a page is a chunk of data typically around 4KB in size, though there are large pages as well. This means accessing a single address in secondary memory will cause an entire page worth of data to be moved into main memory. Accessing a page involves converting a *virtual address* to a *physical address*, which is done with a *page table lookup*. This translation introduces extra overhead, increasing the latency of accessing secondary memory.

A cache operates similarly, but data is moved from from main memory as a *cache line* instead, which is usually either 32B or 64B. When the data is finally read from cache, it is moved as a word, which is 4B or 8B.

In a shared memory system, it is possible for the same piece of data to be located in multiple memory locations, such as the L1 caches of different cores. When a core issues a store, any caches containing that address need to be signaled that their data is now stale. If the data was kept in cache without being marked as stale, cores would end up using old copies of the data, possibly resulting in incorrect computations. More precisely, any reads done by another core to that same address need to return the most recent value for that address. This is known as *cache coherence* [46]. The hardware is responsible for maintaining coherency.

The hardware is also responsible for maintaining *cache consistency*. Cache consistency deals with the relative order that cores see stores completed to different addresses (i.e. when does a read see written data) [47]. A programming language should abstract the memory consistency model and provide primitives such as locks, semaphores, and atomics to the programmer.

2.1.2 High Performance Shared Memory Code

The average programmer has more than likely written code for shared memory systems. While the hardware automatically handles much of the data transfer and synchronization in a shared memory system, there are numerous optimizations that can be made on a software level to improve on what the hardware does, so a high performance programming language needs to support these optimizations. We will look at some examples of loop transformations here, namely *caching*, *vectorization*, and *parallelization*, as well as *NUMA-awareness*, and discuss their performance benefits. These optimizations are integral parts of the TIRAMISU framework that this thesis utilizes.

Caching

Caching can have a tremendous impact on the performance of shared memory code. Ineffective or naive use of the cache can cause unnecessary latency due to events such as *eviction*, *ping-ponging* , and/or *false sharing*. Eviction is when a particular piece of data is removed from cache. If data is evicted, but still needed, it will have to be brought back into cache. If the cache constantly evicts a piece of data, and then brings it back in over and over again, this is ping-ponging. False sharing occurs when different cores access data that shares a cache line. If the cores keep writing to those different addresses, even though they are separate, the whole cache line will keep getting bounced back and forth between their two caches since the whole line keeps getting evicted. This can pose a severe bottleneck.

Related to caching is *data locality*. There are two types of data locality, *spatial*

and *temporal*. Spatial locality refers to multiple addresses. It means that if a memory address is accessed, other memory addresses near it are likely to be accessed in the near future. Since a cache brings in data as a whole cache line, we want our code to access the data in that cache line before it gets evicted from the cache, if possible. Temporal locality refers to a single address. If an address is accessed, temporal locality says that same address will be accessed again in the near future. This means we want to keep the data in cache–if we evict it too soon, we will just have to bring the cache line back in, which takes additional time.

One of the loop transformations used to improve locality is loop tiling. Loop tiling splits and reorganizes a loop nest so that the outer loops iterate over blocks of tiles. Within the tiles, the individual data points are iterated over. Note that tiles do not have to be rectangular, they can theoretically be any shape, but some tile shapes work better than others depending on the loop nest.

Vectorization

Vectorization is a hardware feature that provides *data-level parallelism* though vector instructions in the ISA. It is a SIMD (single instruction, multiple data) operation where the same instruction is applied to different pieces of data in parallel. Current vector instructions sets in x86 ISA today include AVX-512 [42] and SSE4 [28].

Vector instructions execute on special vector hardware, which are similar to ALUs, but operate on vector registers. These registers are larger than general purpose registers, with current vector registers being up to 512 bytes in size. Multiple pieces of data of the same type can be placed into a single vector register. For example, with a 256 byte vector register, you can fit 4 64-bit doubles, 8 32-bit floats, and so on. The vector units execute a vector instruction on all the data in the vector register in parallel. So, with 4 64-bit floats, you theoretically get 4x speedup for that instruction as compared to a SISD (single instruction, single data) instruction.

The code in figure 2-4a shows some simple C code for doing an elementwise addition between two arrays with 8 elements each. Figure 2-4b shows the equivalent vectorized code. The `__mm256` values are vector float types, so they represent 8 float

```
1 float a[8], b[8], c[8];
2 ...
3 for (int i=0; i<8; i++) {
4   c[i] = a[i] + b[i];
5 }
```

(a) An unvectorized elementwise add of two 8-element float arrays.

```
1 float a[8], b[8], c[8];
2 ...
3 __m256 vector_a = _m256_load_ps(a);
4 __m256 vector_b = _m256_load_ps(b);
5 __m256 vector_c = _m256_add_ps(vector_a, vector_b);
6 _m256_store_ps(c, vector_c);
```

(b) The vectorized version using AVX intrinsics.

Figure 2-4: Applying vectorization to an elementwise array addition.

values packed together. `_mm256_load_ps`, `_mm_add_ps`, and `_mm_store_ps` are called *vector intrinsics*. While a compiler typically applies auto-vectorization to code, these intrinsics allow programmers to manually vectorize code.

Parallelization

Shared memory parallelization, also known as *symmetric multiprocessing* (SMP) is an optimization where operations in a program are partitioned and run concurrently on the cores. There are two primary types of parallelism: *data parallelism* and *task parallelism*

1. Data parallelism: The data is partitioned across the cores and each core executes concurrently. Usually, each core is running the same set of operations, just on a different chunk of data. *Loop parallelism* is an example of this.

2. Task parallelism: The operations of a program are grouped together into tasks, which are run concurrently across the cores. This typically entails running different operations on the same chunk of data.

These two types of parallelism can be combined together, which is useful as many programs benefit from both types of parallelism [38].

Parallelism applied naively to a program does not always lead to better results though, as not all programs necessarily benefit from parallel execution. *Amdahl's Law* provides an upper limit to the speedup for a program if sequential parts are parallelized (assuming the problem size is fixed). Amdahl's Law is as follows:

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

where S is the possible speedup, p is the portion of the overall program that can be parallelized, and n is the number of processors to parallelize across. While Amdahl's Law does not take into account factors such as processor type, memory issues, etc, it is great starting point when trying to decide if a program will benefit from parallelization.

Other factors such as *granularity* and *load balancing* can also impact performance. Granularity measures the ratio of the amount of computation done in a parallel region to how often a parallel operation is spawned. *Fine-grained* parallelization means there is a small amount of computation done by each parallel thread, while *coarse-grained* parallelization has a large amount of computation per thread. The correct type of parallelism to apply depends on the application. For example, a loop nest with dependencies between each iteration may not be able to be parallelized at the loop level (coarse-grained), but it may be possible to parallelize operations within the loop body itself (fine-grained).

Load balancing is an important factor in applications where threads are assigned different amounts of work. If the work is unbalanced across threads, then some might finish before others, leaving them idle while waiting for the others to complete. This results in decreased performance as not all threads are being used to their full potential.

While hardware can provide some degree of parallelism, the majority of it is defined through software. Many programming languages have APIs and libraries that can be used for parallelization. Some popular libraries for parallelizing code include OpenMP

```
1 #pragma omp parallel for
2 for (i=0; i<I; i++) {
3   for (j=0; j<J; j++) {
4     A[i*J+j] = A[i*J+j] + 4;
5   }
6 }
```

Figure 2-5: Simple usage of OpenMP compiler directive for parallelizing a loop.

[9], Pthreads [29], Thread Building Blocks [43], Cilk Plus [6], to name a few. We look at OpenMP and Pthreads in more detail as they provide a good look at the differences in these libraries.

OpenMP OpenMP provides an API and set of compiler directives for parallelizing a program. It is implemented in C, C++, and Fortran and can be used for both data and task parallelism. With OpenMP, a programmer simply declares that some code, such as a loop nest, should be executed in parallel. These parallel regions are specified using compiler directives. In C/C++, these are preprocessor instructions specified by the `pragma` keyword. A simple loop parallelization with OpenMP example is shown in figure 2-5. By placing the directive at the top level of the loop nest (line 1), the compiler knows that the outer loop should be parallelized [38]. The `parallel for` directive is a simple parallel directive in OpenMP, but OpenMP offers numerous other directives to fine tune parallelism.

In OpenMP, a programmer specifies what needs to be parallelized, but not how the code is parallelized. The job of determining how to parallelize the code, such as managing the threads, is left to the compiler and the runtime system.

Pthreads Pthreads, which stands for POSIX threads, is another standard for SMP in C/C++ that provides an API for creating and managing threads in a program. The standard, maintained by the IEEE, provides an API for creating and managing threads in a program. Pthreads operates at a lower level than OpenMP, so programmers are responsible for picking out the regions to parallelize and how to parallelize them. The API provides functions for thread management, mutexes, condition variables, and synchronization [29]. This offers more direct control to the programmer,

```
1  void *loop_body(void *_t_idx) {
2    int t_idx = (int)_t_idx;
3    A[t_idx] = A[t_idx] + 4;
4    pthread_exit((void*)_t_idx);
5    return NULL;
6  }
7  ...
8  pthread_t threads[I*J];
9  int *t_idxs[I*J];
10 for (t_idx=0; t_idx<I*J; t_idx++) {
11   t_idxs[t_idx] = (int*)malloc(sizeof(int));
12   *t_idxs[t_idx] = t_idx;
13   pthread_create(&threads[t_idx], NULL, &loop_body, (void*)
       t_idxs[t_idx]);
14 }
15 void *status;
16 for (t_idx=0; t_idx<I*J; t_idx++) {
17   // wait for threads to complete
18   pthread_join(threads[t_idx], &status);
19 }
20 pthread_exit(NULL);
21 ...
```

Figure 2-6: Basic usage of the Pthreads API for parallelizing a loop.

but requires more coding than other frameworks like OpenMP.

Figure 2-6 shows the loop nest in figure 2-5 parallelized with Pthreads instead of OpenMP. First, the loop nest body needs to be converted into a function. This is the function on lines 1 through 6. Then, we need a fair amount of code to manage the threads. Here, we create one thread for every element in `A` (lines 8 through 14). When the actual threads are created on line 13 with the `pthread_create` function, the thread begins execution (or waits until resources are available). A *join* operation is needed to check when the thread has actually completed. This is done on lines 16 through 19 using the `pthread_join` function.

NUMA-Awareness

If a system is set up in NUMA configuration, ignoring the placement of data relative to the processors can result in subpar performance. NUMA-awareness refers to code that optimizes performance based on the NUMA setup. In particular, the code must

deal with *data placement* and *processor affinity*. Data placement in NUMA refers to where in physical memory data resides. For good NUMA performance, data should be placed in the physical memory local to the core using it, as the access time to local memory is faster compared to remote memory. Processor affinity is used to assign a thread or process to a specific core. Usually, the task of assigning threads to cores is left to a scheduler, but most systems have an API that can be used to direct the scheduler [37].

Both data placement and processor affinity need to be dealt with to get good NUMA performance. If a thread is assigned to a core on socket 0, but the data that core needs is placed in socket 1, then it is likely that the core will experience additional latency when accessing that data. Whenever possible, the data should be placed in the memory that is local to the core using it. One tool used to direct processor affinity and data placement is the `numactl` [31] command on Linux. On NUMA-capable machines, a programmer can launch their application using this command and say which cores to execute on, as well as how to bind data to memory.

2.1.3 Limitations of Shared Memory Systems

For many datasets and computations, shared memory systems provide enough resources. However, as the size of datasets increase, shared memory systems begin to perform poorly. Datasets may be too big to fit into the system's main memory, or even into secondary memory. When data cannot fit into main memory, swapping in from secondary memory is expensive and negatively impacts the performance of systems. Distributed memory becomes necessary in these situations. Additionally, only so many cores can be put into a shared memory system. As the number of cores increase, the amount of contention on the system bus and other interconnects from events like data access, coherence, etc. increases to a point where the system becomes unusable. Scaling up to more computing power requires distributing the computation, and/or using other resources such as GPUs. Beyond performance issues, optimized shared memory code is trickier to debug, particularly if it is parallelized, as bugs such as race conditions are easy to introduce and can cause errors or give incorrect results.

2.2 Distributed Memory Systems

A *distributed memory system* is a system made up of independent local memories and processors connected with an interconnect network. Opposed to shared memory systems, there is no single global memory, so explicit communication is required to transfer data between the local memories [57, 22]. These systems are known *multiple-instruction, multiple data* (MIMD) systems. The individual processors (which we refer to as *nodes*) could be anything from single CPUs, to full shared memory systems (called *hybrid distributed memory systems* [57]). Typically, it is up to the programmer to implement a communication scheme for a program. Other compute devices, such as GPUs can also be part of the system, giving a *heterogeneous distributed system*. Distributed memory systems scale better than shared memory systems thanks to more compute resources and more memory. However, factors such as communication, data locality, available parallelism in a program, etc, all need to be considered when writing a distributed memory program.

2.2.1 Architecture and Execution Overview

Figure 2-7 shows an example of distributed memory system made up of four shared memory nodes, with each shared memory node having an associated GPU. One of the most important components of distributed memory systems is the communication interconnect between each of the systems. In the literature, this interconnect may be referred to as *network fabric*. Many distributed systems will require some form of communication or data transfer between the individual systems (for example, if all the data starts on one of the nodes, that node will need to transfer the data to other nodes). If the communication link has high latency or low bandwidth, communication will quickly become the bottleneck, particularly in memory-bound programs.

There are two primary networks used to connect a distributed memory system: *bus-based* and *switch-based*. In a bus-based network, all the nodes share the same communication interconnect (bus). If there are a lot of nodes, the bus will get very congested because every piece of data from every node travels on the same bus, re-

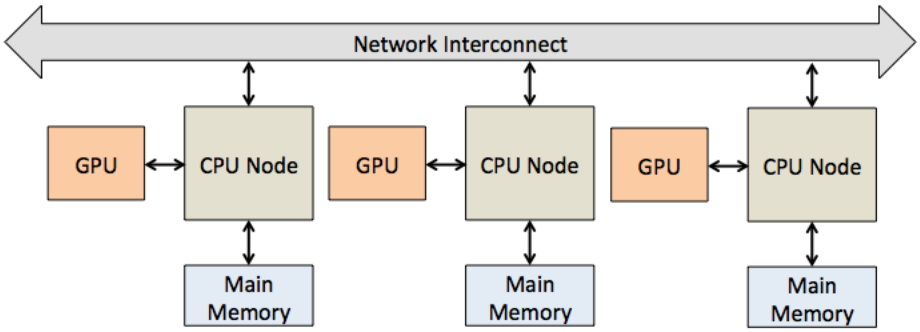


Figure 2-7: Distributed system overview.

gardless of its destination. In a switch-based system, nodes communicate through switches, which sends data to the destination node. This significantly reduces congestion since data only goes through interconnects leading to the destination. It is also much more scalable since new nodes just mean adding a new switch [22]. Figure 2-8 shows a bus-based network configuration and figure 2-9 shows a switch-based network configuration.

There are several interconnects used in distributed memory systems, with some of the main ones being Ethernet [25], InfiniBand [2], and Intel Omni-Path [20]. These interconnects vary in design, latency, bandwidth, price, manufacturer, etc.

2.2.2 High Performance Distributed Memory Code

The main difference between programming shared memory systems and distributed memory systems is the explicit communication required in distributed systems. Communication is used for both synchronization and data transfer, so it is highly likely that a distributed program will have some form of communication. We focus on the data transfer component in this work.

Models for Managing Distributed Memory

Several models exist for programming on distributed memory systems. A big difference between many of the models is how they present the memory address space to the programmer. Some models presented a unified view of the memory, making

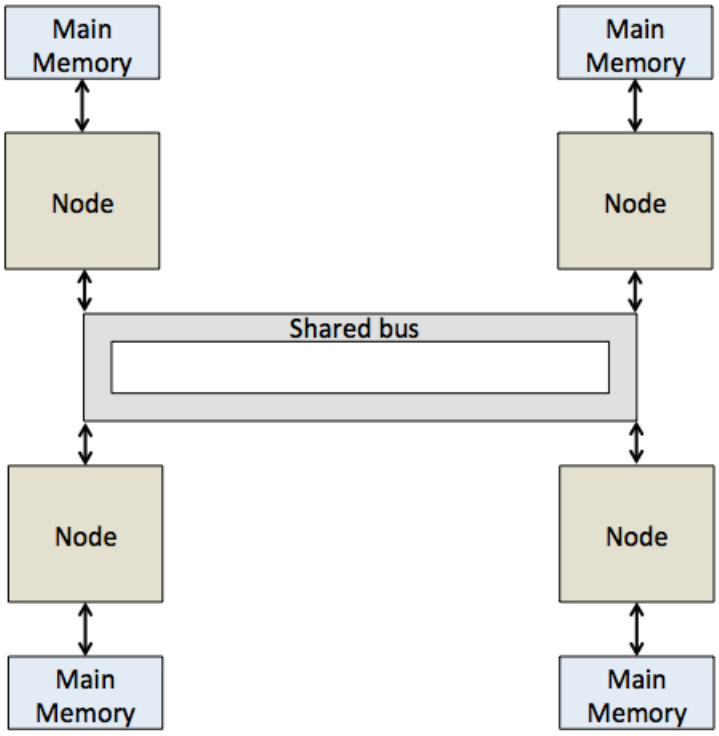


Figure 2-8: Bus-based distributed system.

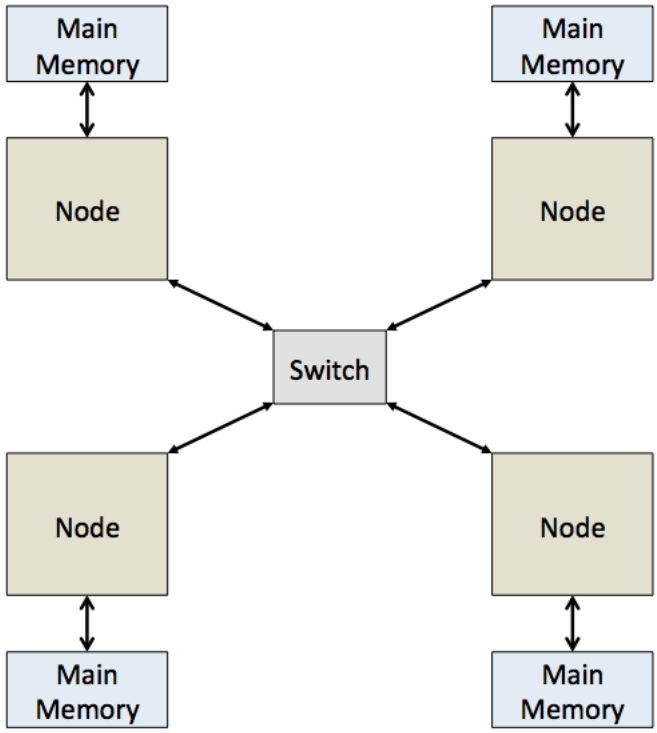


Figure 2-9: Switch-based distributed system.

it look like one big shared memory system. Often, these systems will transparently handle data transfers by adding in the appropriate functions where data is needed. Other models, such as *PGAS*, provide a unified view, but with special constraints on the memory access. This gives more control to the programmer in managing the memory. At the lowest level, there are models like *message passing*, which present the address space as is and programmers are in charge of managing all the data transfers.

PGAS Model PGAS, which stands for partitioned global address space, is a model that looks at distributed memory as a global address space and breaks it into private and shared addresses [7]. It gives a similar representation as a shared memory system. Programmers can assign threads to different operations, where the threads are associated with the different memories in the distributed system. The underlying software handles all communication implicitly. Example languages for this model include UPC [15], Titanium [59], and Co-Array Fortran [32].

Message Passing Model In this model, the programmer is responsible for distributing work among nodes and handling all of the data transfers. Nodes communicate by sending *messages* to each other. Example implementations for this model include OpenMPI [14] and MVAPICH [39]. The work in this thesis utilizes the message passing model.

The most common version of the message passing model is *MPI*, which stands for message passing interface. It is a specification for parallel programming, with a focus on distributed memories. Implementations of the specification provide a software solution for dealing with explicit communication in distributed systems.

MPI can also be used for shared memory parallelization, but other tools, such as OpenMP, remain favored for that area.

At a high-level, MPI launches multiple copies of a program, giving many different processes that (likely) need to communicate with each other. The programmer specifies how the processes cooperate and coordinate with each other. In the program, the user can use a process id to condition certain parts of the code based on the current

```
1  int main() {
2    MPI_Init();
3    int rank;
4    MPI_get_rank(&rank);
5    if (rank == 0) {
6      printf("My rank is 0\n");
7      char *msg = "Hello from rank 0";
8      MPI_Send(msg, 18, MPI_CHAR, 1 /*dst rank*/, ...);
9      printf("Rank 0 sent message to rank 1\n");
10   } else { // rank == 1
11     printf("My rank is 1\n");
12     char msg[18];
13     MPI_Recv(msg, 18, MPI_CHAR, 0 /*src rank*/, ...);
14     printf("Rank 1 received message %s from rank 0\n", msg);
15   }
16   MPI_Finalize();
17 }
```

Figure 2-10: Simple MPI program with two ranks.

id. This way, the user has complete control over which process executes which part of the code. If the user does not supply any explicit synchronization, the individual processes would run completely concurrently.

Figure 2-10 shows a simple example of writing a program that runs in parallel with two processes. One process sends data (known as a *message*) to the other process which receives the message. An MPI `rank` is a unique identifier of an MPI process. By default, each MPI process will be assigned a unique, contiguous integer starting from 0. The `rank` is one of the most useful components in MPI. First, it can be used to condition parts of the code. In the figure, the `if` block shows an example of this. Depending on the `rank` of the process, we want to do one of two things: either we want to send data to the other `rank` (`rank` 0 sends to `rank` 1 using `MPI_Send`), or we want to receive that data (`rank` 1 receives from `rank` 0 using `MPI_Recv`). The `rank` is also used to specify which other processes to communicate with. This can be seen in the `MPI_Send` and `MPI_Recv` on lines 8 and 13.

Going into more detail, we are concerned with MPI functions for *two-sided communication*. In two-sided communication, two ranks communicating with each other must explicitly coordinate: one process needs to initiate a send, and the other needs to

initiate a receive. Based the category of the MPI function, communication may also require explicit synchronization. There are four main categories of communication functions in MPI: *blocking*, *non-blocking*, *synchronous*, and *asynchronous*. Blocking and non-blocking deal with how a thread's execution proceeds after making an MPI function call. Synchronous and asynchronous define what it means for an MPI function to complete. Synchronous and asynchronous only apply to sends, whereas blocking and non-blocking apply to both sends and receives.

1. **Blocking**: Control does not return to the calling thread until the MPI function has completed.

2. **Non-blocking**: Control immediately returns to the calling thread; it does not wait until it is complete. Explicit synchronization is needed to check when the MPI function has actually completed.

3. **Synchronous**: The MPI send is complete when it has transferred all the data froms its send buffer, and the receive has put all of the transferred data into its own buffer.

4. **Asynchronous**: The MPI send is complete when it has transferred all the data from its send buffer.

Sends in MPI can be the following combinations: blocking and synchronous (`MPI_Ssend`), blocking and asynchronous (`MPI_Send`), nonblocking and synchronous (`MPI_Issend`), and nonblocking and asynchronous (`MPI_Isend`). Receives can be blocking (`MPI_Recv`) or non-blocking (`MPI_Irecv`). MPI provides additional functions to query the status of nonblocking operations, such as `MPI_Wait`. `MPI_Wait` will block the calling thread until the specified operation has completed.

All messages sent between a given pair of ranks using these functions arrive in FIFO order. However, there is no guarantee on the ordering of messages across ranks. For example, consider the following scenario with two ranks, `r1` and `r2`, and four messages, `m1`, `m2`, `m3`, and `m4`.

1. `r1` sends `m1` to `r2`.

2. `r1` sends `m2` to `r3`.

3. `r1` sends `m3` to `r2`.

4. `r1` sends `m4` to `r3`.

In this case, `r2` is guaranteed to receive message `m1` before `m3` and `r3` is guaranteed to receive `m2` before `m4`. However, `r2` could receive both `m1` and `m3` before `r3` receives anything. Or `r3` could receive both `m2` and `m4` before `r2` receives any messages. Any interleaving (that still preserves FIFO ordering between a pair of ranks) could happen. MPI provides an optional function argument called a *message tag* that can be used to differentiate between messages.

Optimizing MPI Communication

Done naively, communication can easily become the bottleneck in a distributed program. A big part of getting good performance is overlapping communication with computation. With blocking communication, the calling thread cannot proceed until the communication is finished, so it cannot do any other useful work in the meantime. However, with non-blocking communication, the thread can do other useful work (assuming there is some) instead of waiting for the communication to finish.

Consider the example code in figures 2-11 and 2-13. Both figures give code for performing a distributed matrix-vector multiply. We assume that there are two ranks, `r0` and `r1`, and the data for the matrix and vector initially reside on the machine running the process for `r0`. We will have each rank operate on half of the matrix. This means that `r0` must send the vector and half of the matrix to `r1`, and then `r1` must send its results back to `r0`. We assume that data is row-major, and our matrix dimension is `M` rows by `N` columns. The difference between the two sets of code is that 2-11 uses blocking communication and 2-13 uses a combination of blocking and non-blocking communication. The blocking version starts by sending the entire vector from `r0` to `r1` on line 6. In line 7, `r0` the entire first half of the matrix to `r1`. `r0` must will block until the sends complete, so it cannot execute its multiply in the loop nest at line 8 until those sends are done. `r0` has no overlap of communication and

```
1  ...
2    MPI_Init(); int rank; MPI_get_rank(&rank);
3    if (rank == 0) {
4      vector = <read from file>; matrix = <read from file>
5      output = (float*)malloc(sizeof(float)*M);
6      MPI_Send(vector, N, MPI_FLOAT, 1, ...);
7      MPI_Send(matrix, (M/2)*N, MPI_FLOAT, 1, ...);
8      for (int r = M/2; r < M; r++) {
9        float sum = 0.0f;
10       for (int c = 0; c < N; c++) {
11         sum += matrix[r*N + c] * vector[c];
12       }
13       output[r] = sum;
14     }
15     MPI_Recv(output, M/2, MPI_FLOAT, 0, ...);
16   } else { // rank == 1
17     vector = (float*)malloc(sizeof(float)*N);
18     matrix = (float*)malloc(sizeof(float)*(M/2)*N);
19     output = (float*)malloc(sizeof(float)*M/2);
20     MPI_Recv(vector, N, MPI_Float, 0, ...);
21     MPI_Recv(matrix, (M/2)*N, MPI_FLOAT, 0, ...);
22     for (int r = 0; r < M/2; r++) {
23       float sum = 0.0f;
24       for (int c = 0; c < N; c++) {
25         sum += matrix[r*N + c] * vector[c];
26       }
27       output[r] = sum;
28     }
29     MPI_Send(output, M/2, MPI_FLOAT, 0, ...);
30   }
31   MPI_Finalize();
32 ...
```

Figure 2-11: Two process distributed matrix-vector multiply using blocking communication.

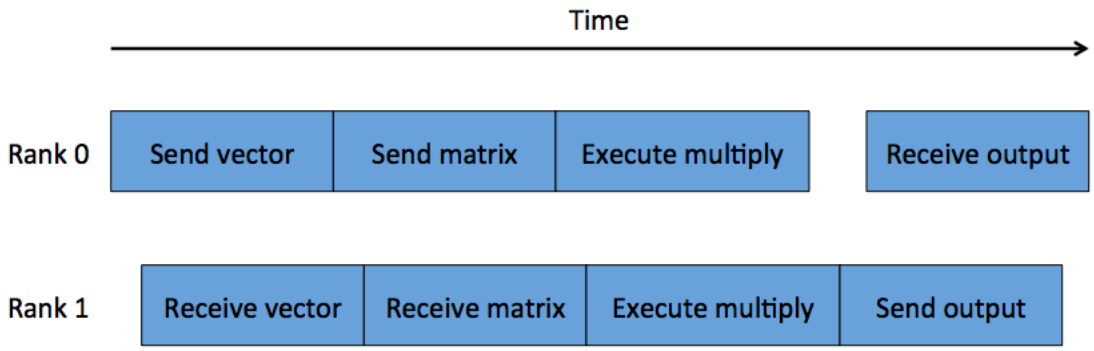


Figure 2-12: Timeline showing sequential communication within a rank due to the use of blocking functions.

computation here. `r0`'s last job is to get the results from `r1` and put them in the final output array. This occurs at line 15.

`r1` operates similarly to `r0`, except it receives the messages from `r0` (lines 20 and 21) and then sends its results to `r0` (line 29). Like `r0`, `r1` cannot begin computing the multiply starting at line 22 until its receive is complete. So `r1` also does not have any overlap of communication and computation.

Figure 2-12 shows an approximate (not-to-scale) timeline of execution for `r0` and `r1`. Each rank runs concurrently, but the actual operations within a rank run sequentially, so there is no overlap.

The combined blocking and non-blocking version takes advantage of the ability to overlap communication and computation. `r0` starts the same and transfers the `vector` all in one message on line 6. For the `matrix`, it sends one row of data at a time (line 10), so a total of `M/2` messages are sent. Since the send is non-blocking, it immediately returns and `r0` executes the multiply with its current row. `r0` ends with getting the output from `r1` as before, but it first needs to make sure all its non-blocking sends finished before launching more operations (this is the `MPI_Waitall` function on line 15). `MPI_Request` objects contain information on the status of functions, so MPI wait functions query these to see if an operation is complete. `r0` finishes by receiving the output from `r1` with a non-blocking function followed by another `MPI_Waitall` on lines 18 and 20.

`r1` also starts as before with a blocking receive of the `vector` on line 25. Since

```
1  ...
2    MPI_Init(); int rank; MPI_get_rank(&rank);
3    if (rank == 0) {
4      vector = <read from file>; matrix = <read from file>
5      output = (float*)malloc(sizeof(float)*M);
6      MPI_Send(vector, N, MPI_FLOAT, 1, ...);
7      MPI_Request *reqs = (MPI_Request*)malloc(sizeof(MPI_Request
           )*M/2);
8      for (int r = M/2; r < M; r++) {
9        MPI_Request req; reqs[r-M/2] = req;
10       MPI_Isend(&matrix[(r-M/2)*N],N,MPI_FLOAT,1,...,&req);
11       float sum = 0.0f;
12       for (int c = 0; c < N; c++) { sum += matrix[r*N+c] *
             vector[c]; }
13       output[r] = sum;
14     }
15     MPI_Waitall(M/2, reqs, ...);
16     for (int r = 0; r < M/2; r++) {
17       MPI_Request req; reqs[r] = req;
18       MPI_Irecv(&output[r], 1, MPI_FLOAT, 0, ..., &req);
19     }
20     MPI_Waitall(M/2, reqs, ...);
21   } else { // rank == 1
22     vector = (float*)malloc(sizeof(float)*N);
23     matrix = (float*)malloc(sizeof(float)*(M/2)*N);
24     output = (float*)malloc(sizeof(float)*M/2);
25     MPI_Recv(vector, N, MPI_FLOAT, 0, ...);
26     MPI_Request *reqs = (MPI_Request*)malloc(sizeof(MPI_Request
           )*M/2);
27     for (int r = 0; r < M/2; r++) {
28       MPI_Request req; reqs[r] = req;
29       MPI_Irecv(&matrix[r*N], N, MPI_FLOAT, 0, ..., &req);
30     }
31     for (int r = 0; r < M/2; r++) {
32       MPI_Wait(reqs[r], ...);
33       float sum = 0.0f;
34       for (int c = 0; c < N; c++) { sum += matrix[r*N+c] *
             vector[c]; }
35       output[r] = sum;
36       MPI_Request req; reqs[r] = req;
37       MPI_Isend(&outputs[r], 1, MPI_FLOAT, 0, ..., &req);
38     }
39     MPI_Waitall(M/2, reqs, ...);
40   }
41   MPI_Finalize();
42 ...
```

Figure 2-13: Two process distributed matrix-vector multiply combining blocking and non-blocking communication.

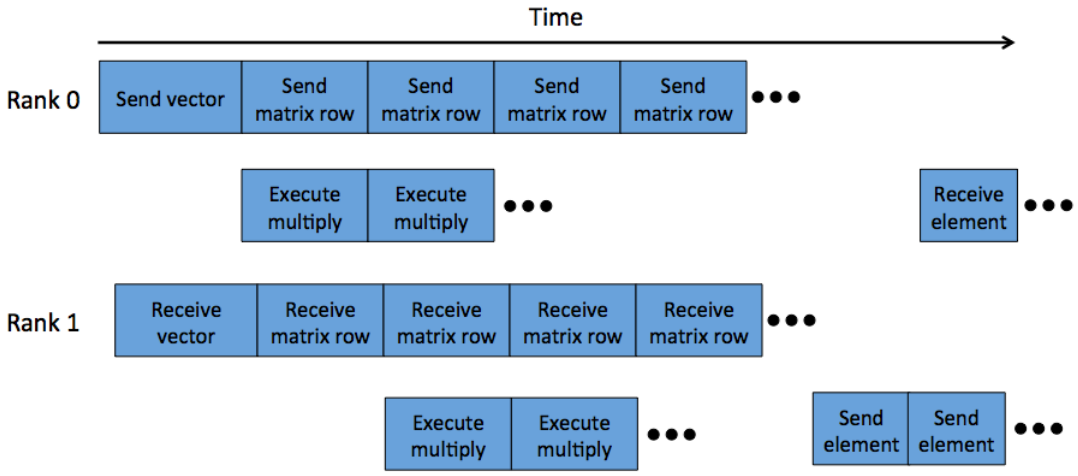


Figure 2-14: Timeline showing overlapping communication and computation with non-blocking functions.

the `matrix` was sent row-by-row, `r1` needs to receive the `matrix` row-by-row also, which it does on line 29. These function calls merely initialize the receives; control immediately returns to the thread and it proceeds to the loop nest on line 31. Before doing the multiply, we need to make sure the data for that row has been completely received. This is accomplished with the `MPI_Wait` on line 32. At the end of the loop nest, `r1` sends each output as it is finished with a non-blocking send on line 37, followed by the `MPI_Waitall` on line 39.

Figure 2-14 gives another timeline, this time showing what happens with the non-blocking operations. As before, `r0` and `r1` run concurrently, but now there is also concurrency within each rank as some of the sends and receives, as well as the computation, can overlap.

2.2.3 Limitations of Distributed Memory Systems

While distributed memory systems offer more scalability than their shared memory counterparts, these systems have other performance considerations such as communication. In some cases, the data may exist on only one of the nodes and need to be transferred to other nodes. If the actual computation executed is not very time consuming, the cost of transferring the data could outweigh any benefits. Data can also start distributed across several nodes, especially if there is too much data to fit

into the memory of a single machine. In this, data may have to be sent multiple times to deal with the memory limitations, so care should be taken when implementing the communication.

Additionally, distributed memory systems are harder to program than their shared memory counterparts for several reasons. First off, the programming model is different. Distributed memory code will look different than an equivalent shared memory implementation due to the synchronization and other such code needed to manage the different processes executing. Even small conceptual changes, like modifying the loop that is distributed, can result in a complete rewrite of the code.

Like parallelized shared memory code, distributed code is hard to debug because it introduces concurrency into a program. Also, programmers need to keep track of where pieces of data exist at any given time and make sure individual processes are operating on the correct version of the data.

2.3 Graphics Processing Units (GPUs)

Graphics processing units (GPUs) have become an integral part of high-performance computing in recent years. Traditionally used for rendering graphics, GPUs have expanded into many other fields such as linear algebra, deep learning, speech processing, image processing, weather simulations, and bioinformatics, thus sparking the term GPGPU (*general purpose GPU computing*). Compared to traditional CPU based systems which optimize for latency, GPUs optimize the throughput of operations, potentially running thousands of operations in parallel. GPUs also have very different architectures from CPUs, and they use a different programming model.

2.3.1 Architecture and Execution Overview

At the highest level, GPUs are collections of up to 1000s of individual compute cores, offering massive amounts of parallelism in comparison to typical CPUs. The way these individual cores are configured is different based on the GPU model. Figure 2-15 shows the architecture of the NVIDIA Pascal GP100 GPU. The computing units

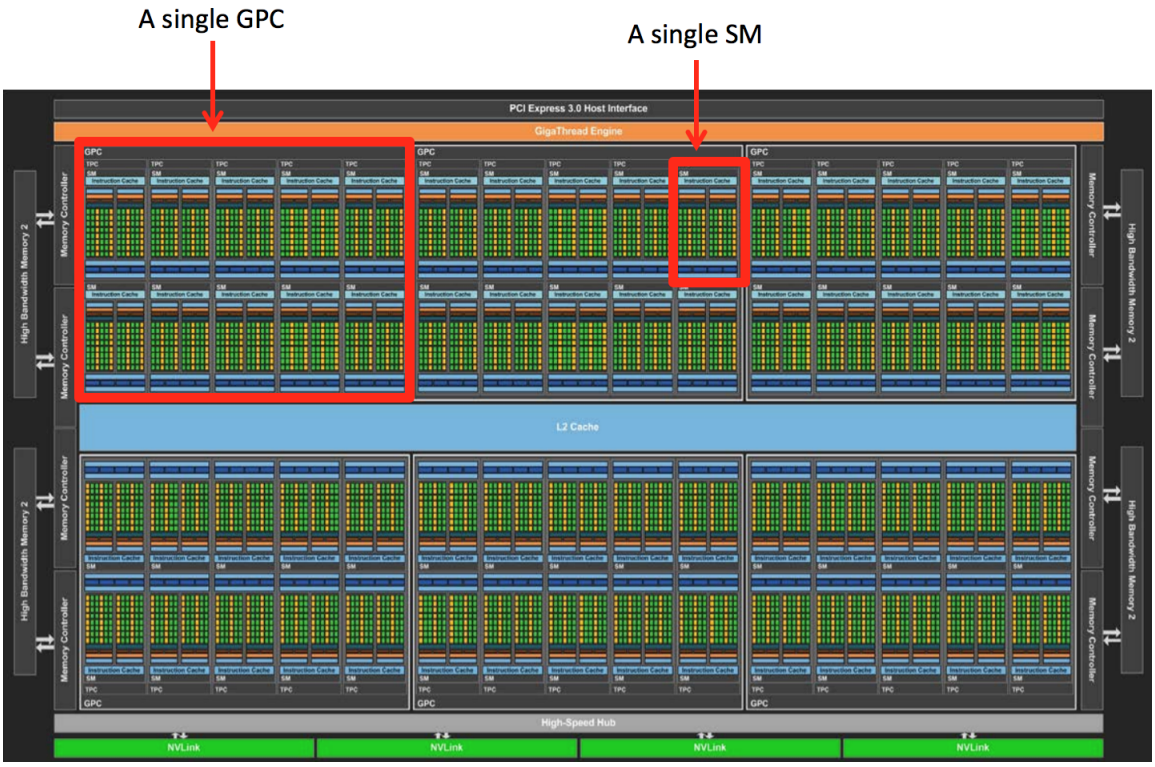


Figure 2-15: Overall architecture of the NVIDIA Pascal GP100 GPU [34].

on the GPU are broken down into groups called *streaming multiprocessors* (SMs). SMs are grouped into bigger units, which NVIDIA calls *graphics processing clusters* (GPCs). In this particular configuration, there are six GPCs arranged in an array, each containing 10 SMs. Each GPC shares the L2 cache shown in the middle. On the left and right are the interfaces to the main memory of the GPU, called *global memory*. Since this memory is separate from the CPU memory, this is similar to a distributed system, though the more appropriate term is heterogeneous. The top of the figure has a PCIe interface that connects to the CPU (also called the *host*), while the bottom has a special high-speed interface called NVIDIA NVLink that connects GPUs directly together. NVLink provides up to 5x more bidirectional bandwidth than traditional PCIe [34]. Note that previous NVIDIA GPU architectures, such as Fermi, Kepler, and Maxwell, do not have NVLink interfaces, so the GPUs would also communicate via PCIe.

The next figure, figure 2-16, shows the configuration of an SM for the Pascal

Figure 2-16: SM configuration of the NVIDIA Pascal GP100 GPU [34].

GP100. The individual `cores`, known as CUDA cores[1], are grouped into two symmetric sets. In this case, there are 32 CUDA cores per set, giving 64 total cores per SM. These cores are specialized for single precision arithmetic. The cores labeled `DP Unit` are specialized for double precision arithmetic. There are less of them, so the overall performance of double precision operations is less than single precision. The `LD/ST` units are load/store units, and the `SFU` cores are for special arithmetic operations such as `sqrt`. Each SM also has a set of registers, an L1 cache, and a shared memory cache. There are some additional memories, such as *constant memory*, but we do not consider those here.

Theoretically, this setup, with six GPCs and 10 SMs/GPC, would give 60 SMs and 3840 total cores. However, due to various constraints such as power consumption, different GPU models do not enable all of the SMs. For example, the Tesla P100, which is built from the Pascal GP100 architecture, uses 56 of the 60 SMs, giving 3584 total usable cores [34].

[1]In other GPUs, such as AMD ones, the cores are called *streaming processors* (SPs).

	NVIDIA GPU Architecture		
Feature	**Kepler**	**Maxwell**	**Pascal**
SMs	15	24	56
Total FP cores	2880	3072	3584
Total DP cores	960	96	1792
Main memory size	12GB	24GB	16 GB
Peak FP GFLOPs	5040	6840	10600
Peak DP GFLOPs	1680	210	5300
Max threads per warp	32	32	32
Max block size	1024	1024	1024
Max warps per SM	64	64	64
Max blocks per SM	16	32	32

Table 2.2: Architecture comparison between different generations of NVIDIA GPUs [34].

GPU caches have a different purpose than CPU caches. First off, in the CPU, caches are managed by hardware; in the GPU, they are partly managed by hardware, and partly managed by the programmer. Secondly, a GPU cache is mainly used to help reduce the memory bandwidth to global memory, rather than reducing latency like CPU caches [12]. In a GPU, there are thousands of threads accessing memory, and having them all accessing global memory causes a lot of contention. The GPU caches have less threads accessing them, so there is not as much contention. Even though the caches do not help much with latency in a GPU, the sheer number of threads running in parallel can mask latency in memory access.

The GPU uses an execution model known as *single instruction, multiple thread* (SIMT). In this model, all the threads that issue together in a group, called a *warp*, execute the same instruction in parallel on different pieces of data. Each thread maps to one of the CUDA cores in an SM. The overall code containing the instructions to execute on the GPU is called a *kernel*.

From the programmer's perspective, GPU threads are organized into *blocks* and *grids*. A block is a group of threads and a grid is a group of blocks. All the threads in a block can communicate with each other via shared memory. Configuring block size and grid size is important to performance. Blocks execute in parallel and must be independent from one another because they are unable to communicate and can

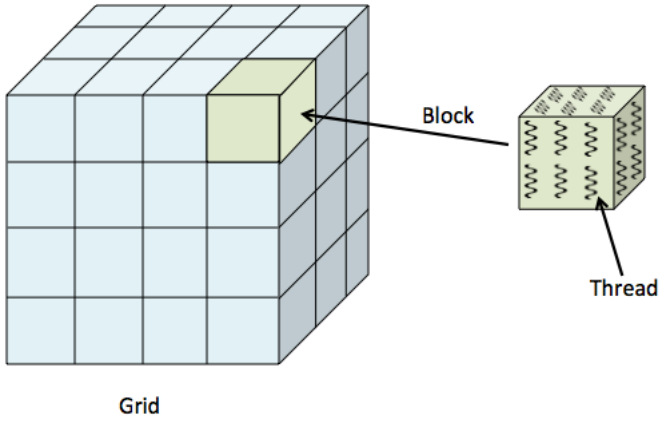


Figure 2-17: Grid, block, thread hierarchy.

execute in any order. Figure 2-17 gives a look at the thread, block, and grid hierarchy. Blocks and grids can be 1, 2, or 3-dimensional. The appropriate configuration depends on the program. Blocks are executed on SMs; multiple blocks can be assigned to a single SM, and the hardware will figure out how to schedule the underlying warps. Table 2.2 gives various GPU characteristics, such as warp size, block size, GFLOPs, etc, for the NVIDIA Kepler, Maxwell, and Pascal architectures.

2.3.2 High Performance GPU Code

There are several ways to program a GPU, from using low-level GPU assembly such as NVIDIA's PTX language [40], to slightly-higher level graphics programming languages such as OpenGL [58], or libraries like Microsoft's DirectX [24]. However, these languages are either too low-level to use for general purpose GPU computing, or are focused on graphics programming. In recent years, high-level languages like NVIDIA CUDA [30] and OpenCL [51] have emerged as the defacto way to program a GPU for general purpose usage. In this thesis, we utilize CUDA, so we will look at the CUDA language.

```
1 float *a, *b, *c;
2 ...
3 for (int i = 0; i < I; i++) {
4   a[i] = b[i] + c[i];
5 }
```

(a) The sequential loop.

```
1 __global__
2 void vector_add_kernel(float *a, float *b, float *c, int I) {
3   int global_idx = blockIdx.x * blockDim.x + threadIdx.x;
4   if (global_idx < I) {
5     a[global_idx] = b[global_idx] + c[global_idx];
6   }
7 }
```

(b) The equivalent CUDA kernel.

Figure 2-18: A sequential loop and the equivalent CUDA kernel for parallel execution on the GPU.

CUDA Programming Model

CUDA (which stands for Compute Unified Device Architecture) was introduced in 2007 by NVIDIA as a parallel software model for GPU programming. It was developed as an extension to C/C++ and provides control over everything from memory management to kernel execution.

With CUDA, programmers write C/C++ as usual, but are provided with library functions and declaration specifiers that mark code for either host or device (i.e. GPU) execution. GPU kernels are marked with the `__global__` specifier. In CUDA, the kernel is a function called from the host. Any other GPU functions called within the kernel are gpu-only functions.

The main component of any CUDA program is the kernel itself. A kernel can be thought of as the body of the sequential loop, where the loop iterations are executed as CUDA threads running in parallel. Figure 2-18a shows a simple sequential loop and figure 2-18b shows the equivalent CUDA kernel for parallel execution.

On line 3, `blockIdx.x`, `blockDim.x`, and `threadIdx.x` are built-in variables that let us compute the global index from the local block and thread indices. This is the

equivalent of the values for loop iterator `i` in the sequential loop example.

The next big component is data transfer and memory management. To get the best performing code in CUDA, programmers need to manage memory manually. Like using MPI for distributed memory, CUDA provides numerous functions for transferring data between CPU and GPU memory. Before a GPU kernel is executed, all the necessary data for that kernel has to be transferred to the GPU. Similarly, any data produced by the GPU that is needed by the CPU later on will have to be copied back down to the CPU. Newer versions of CUDA have a feature called *unified memory* that allows a single pointer to be used for both the CPU and GPU data. CUDA then handles all the necessary memory transfers automatically. But, this does not give the best performance, so we do not utilize unified memory in our work.

Asynchronous Execution Two important types of execution modes in CUDA are *synchronous* and *asynchronous*. These are similar to the blocking and non-blocking calls of MPI, respectively; synchronous CUDA calls block the caller until the operation is complete and asynchronous calls return immediately. Explicit synchronization is required for figuring out when an asynchronous call is actually complete. In CUDA, there are not separate functions for sending and receiving like in MPI.

By default, kernels are launched asynchronously. However, without additional modifications, asynchronous CUDA transfers and kernels will not overlap. As soon as another operation is launched, whether asynchronous or not, it will block until the previous asynchronous operation finishes. Overlap requires an additional feature called *stream*s. A stream contains a series of operations that execute sequentially, where an operation is something like a kernel or a data transfer. For example, say you have two kernels, `k1` and `k2`. If you execute `k1` followed by `k2` in the same stream, then `k2` will not execute until `k1` has completed, even though kernel launches in CUDA are asynchronous. However, the individual threads of the kernels will still run in parallel. Individual streams can run concurrently, so it is possible to run multiple kernels concurrently, as well as overlap kernel execution with data transfers. More recent NVIDIA GPUs support executing kernels, CPU-to-GPU, and GPU-to-

```
1  ...
2    int M = ...;
3    float *matrix_A = <read data from file>
4    float *matrix_B = <read data from file>;
5    float *matrix_C = (float*)malloc(sizeof(float*)*M*M);
6    float *d_A, *d_B, *d_C;
7    cudaMalloc(&d_A, sizeof(float)*M*M);
8    cudaMalloc(&d_B, sizeof(float)*M*M);
9    cudaMalloc(&d_C, sizeof(float)*M*M);
10   cudaMemcpy(d_A, matrix_A, sizeof(float)*M*M,
11              cudaMemcpyHostToDevice);
12   cudaMemcpy(d_B, matrix_B, sizeof(float)*M*M,
13              cudaMemcpyHostToDevice);
14   KERNEL<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>
15             (d_A, d_B, d_C, M);
16   cudaMemcpy(C, d_C, sizeof(float)*M*M,
17              cudaMemcpyDeviceToHost);
18 ...
```

Figure 2-19: CUDA code for synchronous communication and kernel execution.

CPU transfers concurrently, assuming each is placed in a separate stream.

Without explicit synchronization, streams will run concurrently, regardless of any dependences that exist between streams. Programmers can explicitly synchronize across streams using CUDA *events*. Events are operations that can be placed between other operations in streams and are used to determine when the previous operation in a stream has completed.

Figures 2-19 and 2-21 give some CUDA code for launching a kernel in between memory copies from CPU-to-GPU and GPU-to-CPU. We assume that the kernel is something that can execute on a row-by-row basis (such as some type of elementwise operation). Figure 2-19 uses synchronous (blocking) communication, while figure 2-21 uses a combination of synchronous and asynchronous (non-blocking) communication.

In the synchronous code, we first need to allocate memory on the GPU for the two input matrices (`d_A` and `d_B`), as well as the result matrix (`d_C`). The `cudaMalloc` function on lines 7, 8, and 9 allocate memory in GPU global memory. Throughout this example, we assume that the various parameters (such as data size) fit within the GPUs limits. Next, the matrices are copied up to the GPU on lines 10 through 13.

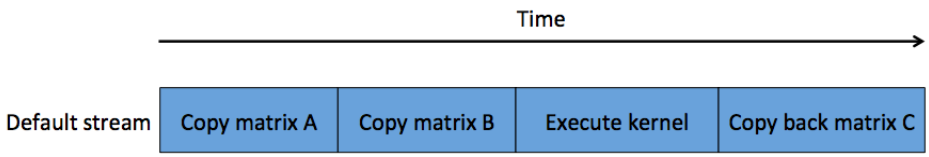


Figure 2-20: Timeline for kernel execution and communication using only synchronous CUDA operations.

`cudaMemcpy` is used for synchronous copies and can be used in any transfer direction.

Lines 14 and 15 use the CUDA launch the kernel. The parameters between the triple chevrons (`<<<...>>>`) are the grid size and block size, respectively. Our algorithm uses 1 thread per element, so the programmer needs to make sure that the `BLOCKS_PER_GRID` and `THREADS_PER_BLOCK` parameters cover the full dimensions of the matrices. Once the kernel is complete, the result is copied back to the CPU starting on line 16. The timeline for this code is shown in figure 2-20.

The combined synchronous/asynchronous code in figure 2-21 begins the same way as the synchronous code–it allocates GPU memory. On lines 10 to 13, we create three cuda streams. Unlike the synchronous code, here, we will copy the matrices up row-by-row and launch a kernel for each row. The row copies happen in lines 18 through 21 using `cudaMemcpyAsync`. This call is the same as `cudaMemcpy`, except it takes an additional parameter: the stream. On lines 25 and 26, the kernel is launched for one row. It also takes a stream identifier. The copy and the kernel execute in separate streams, thus copies and kernels can run concurrently. We also copy back one row at a time starting on line 29 using a different stream so that copy can run concurrently.

Note that we have a dependence between copies and kernel launches. Before launching the kernel for a row, we need to verify that the row has fully copied to the GPU. Similarly, before copying the result back to the CPU, we need to make sure that the kernel has completed. On lines 15 through 17 we create CUDA events to place in the streams. Line 22 places the `copy_event` event into the `cpu_to_gpu_stream`. Line 23 then tells the kernel's stream, `kernel_stream`, to wait for that event before launching. Likewise, line 27 places the `kernel_event` in the `kernel_stream` and then line 28 tells the copy back to wait for `kernel_event` to complete. This synchroniza-

```
1  ...
2    int M = ...;
3    float *matrix_A = <read data from file>
4    float *matrix_B = <read data from file>;
5    float *matrix_C = (float*)malloc(sizeof(float*)*M*M);
6    float *d_A, *d_B, *d_C;
7    cudaMalloc(&d_A, sizeof(float)*M*M);
8    cudaMalloc(&d_B, sizeof(float)*M*M);
9    cudaMalloc(&d_C, sizeof(float)*M*M);
10   cudaStream_t cpu_to_gpu_stream, gpu_to_cpu_stream,
        kernel_stream;
11   cudaStreamCreate(&cpu_to_gpu_stream);
12   cudaStreamCreate(&gpu_to_cpu_stream);
13   cudaStreamCreate(&kernel_stream);
14   for (int r = 0; r < M; r++) {
15     cudaEvent_t copy_event, kernel_event;
16     cudaEventCreate(&copy_event);
17     cudaEventCreate(&kernel_event);
18     cudaMemcpyAsync(&d_A[r*M], &matrix_A[r*M], sizeof(float)*M,
19                     cudaMemcpyHostToDevice, cpu_to_gpu_stream);
20     cudaMemcpyAsync(&d_B[r*M], &matrix_B[r*M], sizeof(float)*M,
21                     cudaMemcpyHostToDevice, cpu_to_gpu_stream);
22     cudaEventRecord(copy_event, cpu_to_gpu_stream);
23     cudaStreamWaitEvent(kernel_stream, copy_event, ...);
24     KERNEL<<<BLOCKS_PER_GRID, M/BLOCKS_PER_GRID,
25              kernel_stream>>>
26               (&d_A[r*M], &d_B[r*M], &d_C[r*M], M);
27     cudaEventRecord(kernel_event, kernel_stream);
28     cudaStreamWaitEvent(gpu_to_cpu_stream, kernel_event);
29     cudaMemcpyAsync(&C[r*M], &d_C[r*M], sizeof(float)*M,
30                      cudaMemcpyDeviceToHost, gpu_to_cpu_stream);
31   }
32   cudaDeviceSynchronize();
33 ...
```

Figure 2-21: CUDA code for a combination of asynchronous and synchronous communication and kernel execution.

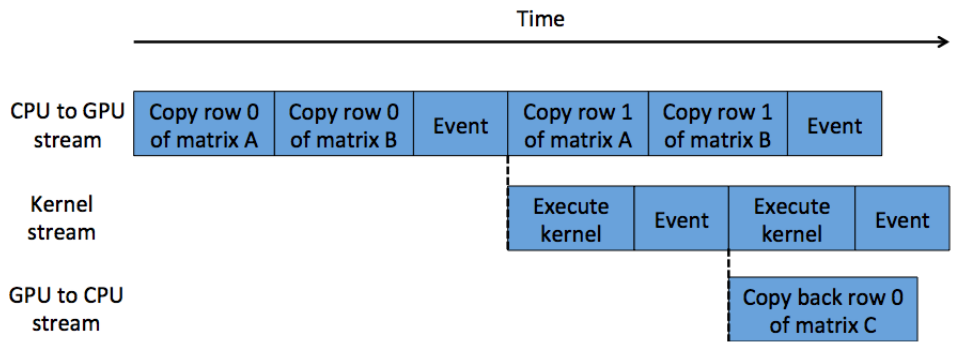


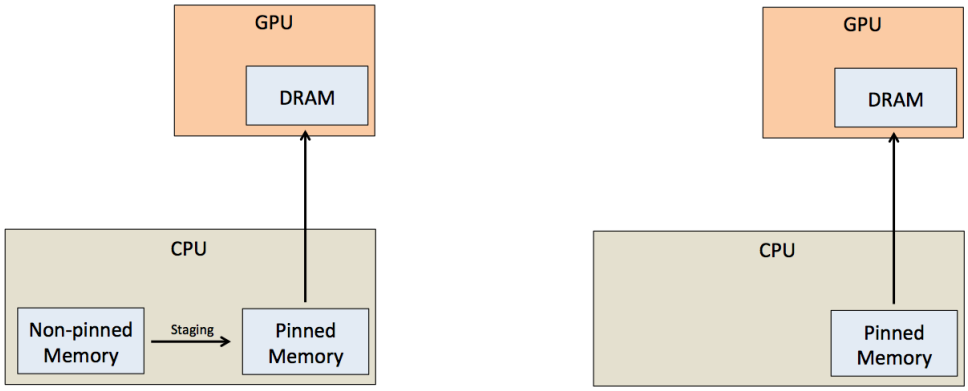
Figure 2-22: Timeline for kernel execution and communication using a combination of synchronous and asynchronous CUDA operations.

tion scheme forces the copies and kernel to execute sequentially within a single loop iteration, but run overlap across iterations. A timeline for this version is shown in figure 2-22.

Four additional performance points worth mentioning are *GPU occupancy*, *pinned memory*, *branch divergence*, and *memory coalescing*.

GPU Occupancy GPU occupancy is a measure that roughly translates to resource utilization. It is the ratio of the number of active warps executing on an SM to the maximum number of active warps allowed on an SM (i.e. running concurrently). A warp is active from the moment at least one of the threads in it begins until the moment the last thread finishes [1]. Block size and grid size are very important to GPU occupancy, so they should be tuned accordingly. Too low of an occupancy and the GPU threads are not being effectively used; too high and the amount of resources available to each thread (such as memory) is limited.

Pinned Memory Pinned memory is special page-locked memory allocated on the CPU. The GPU cannot directly access memory that is pageable when doing a transfer, so without pinned memory, data to transfer has to first be copied to a separate staging area in CPU memory, and then transferred to the GPU. The same goes for transferring data from the GPU back down to the CPU–the copy back would be staged in CPU memory and then moved to its final destination. The staging area itself is page-locked pinned memory, so by allocating page-locked memory before



(a) Copying to GPU DRAM with a staging copy required first.

(b) Copying to GPU DRAM without having to copy to the staging area first.

Figure 2-23: Copying to GPU DRAM without pinning memory beforehand (left), and copying with pinning beforehand.

hand, the additional copy to the staging area is eliminated, potentially saving a signficant amount of time, particularly for large transfers. Figure 2-23a shows the two copies required when staging and figure 2-23b shows the single copy to GPU DRAM without staging. Eliminating this staging copy gives what is known as a `zero-copy transfer`. CUDA asynchronous transfers require pinned memory, otherwise they will just execute synchronously [18].

Branch Divergence Branch divergence happens within a warp when a control flow operation is encountered. Threads in a warp can only issue the same instruction together, so if some of the threads resolve to one direction, and other threads go in another direction, the warp will execute those different branch directions sequentially. The warp will disable the threads that are not supposed to execute that direction, and then re-enable them when it is time to execute their branch. This means that a warp is most effective when all of its threads agree. As a general rule of thumb, it is a good idea to avoid branching in a kernel if possible, or at least minimize its impact on warp execution [36].

Memory Coalescing When accessing global memory within a kernel, a warp tries to group all of its memory accesses together into as few transfers as possible. This

is called memory coalescing. It is able to do this when memory accesses are either sequential (relative to the innermost thread dimension), or strided such that each access goes to a different memory bank. Grouping together transfers minimizes the contention in accessing the memory. If memory accesses are non-sequential, indirect, or strided such that they access the same memory bank, then the access will be un-coalesced and require additional transfers, increasing contention on the memory, and decreasing overall performance [17].

2.3.3 Limitations of GPUs

Even though GPUs offer the potential for massive speed up, they are not always better than CPUs. First off, programs need to have a high degree of parallelism to take advantage of the GPU, as the latency of an individual GPU core is higher than a CPU core. Thus, latency critical applications may not perform well, particularly if there is not enough parallelism.

Communicating with the GPU is also expensive, as it is similar to sending data in a distributed system. If an operation is memory-bound, the cost of transferring data may overtake any improved compute performance. Special care needs to be taken when implementing the communication, just like with distributed systems. Additionally, the memory on GPUs is much smaller than on shared memory systems (GPU DRAM is about 12GB-16GB). Pinning memory can also have negative performance impacts, so it needs to be handled carefully. Pinned memory reduces the amount of pageable memory (i.e. memory that can be swapped in and out of RAM), so too much of it and performance will suffer, but too little and it will be harder to see benefits from asynchronous data transfers.

As previously mentioned, branch divergence and un-coalesced memory accesses can also a significant performance impact, so if an application can not eliminate (or limit) the amount these occur, a GPU program may not execute well compared to an equivalent CPU implementation.

Programming GPUs requires a different programming model than CPUs, making them harder to program. Languages like CUDA certainly make it easier, but it

requires a different way of thinking. For example, loops turn into kernels, memory requires more careful handling, etc. Even calling a kernel requires a significant amount of coding.

Similar to parallelization on shared memory and distributed memory systems, debugging CUDA code is difficult. Bugs could be introduced not only from the parallelism, but also the asynchronous communication. NVIDIA provides tools like `nvprof`, `cuda-gdb`, and `cuda-memcheck` to help with debugging, but it remains a tedious process.

Chapter 3

The TIRAMISU Compiler

The work presented in this thesis makes use of an existing compiler called TIRAMISU, which was developed in the COMMIT group at MIT [53]. TIRAMISU is a middle-end polyhedral compiler for DSLs that abstracts the underlying architecture using a three-layer design. TIRAMISU's design makes it easier for programmers to modify parts of their program, such as data layout, without having to modify other components, such as the high-level algorithm. Utilizing the polyhedral model, loop transformations and optimizations become much easier as the programmer just has to specify what they want done to the loop, and the polyhedral model generates the correct loop representation.The programmer uses a *scheduling language* to specify the transformations, and as we will see, the scheduling language is a major component of TIRAMISU.

We begin this chapter with an introduction to the polyhedral model, which TIRAMISU uses to represent programs.

3.1 The Polyhedral Model

3.1.1 Overview

The *polyhedral model* is a mathematical model for representing sequential programs built around loop nests. The loop nests and transformations on those loop nests are described using algebraic operations. The polyhedral model composes transformation

operations together with the loop nest representation into a single algebraic operation that can be used to generate backend code. These loop transformations are typically done with performance optimization in mind. Consider the simple two-level loop nest in figure 3-1a, and the subsequent transformations, *loop distribution*[1], *loop tiling*, and *loop interchange* in figures 3-1b, 3-1c, and 3-1d, respectively. There are several parts to the polyhedral description of these loops.

The first is the *iteration domain*. The iteration domain is the set of all possible values for the *loop iterators* of the loop nest. The iteration domain for the previous loop nest would contain all tuples of iterators i and j, written as:

$$\{(i, j) \in \mathbb{Z}^2 : 0 \leq i < 100 \ \wedge \ 0 \leq j < 10\}$$

More specifically, the iteration domain is an integer set with values $\in \mathbb{Z}^d$, where d represents the depth of the loop nest. The inequalities in the iteration domain define a set of *constraints* that represent a polyhedron. In the polyhedral model, these constraints are *quasi-affine* inequalities and are expressed in terms of loop iterators, loop invariants, and integer literals. Affine means that the inequalities are linear, so they are of the form $\{x \in \mathbb{Z}^d : Ax + a \geq 0\}$. x represents the tuple of loop iterator values, and A and a represent a matrix and vector of loop invariants, respectively. Quasi means that the constraint can also contain existentially quantified variables, meaning that these variables have some integer value that exists and satisfies the constraint [5, 54].

The binary operators allowed in the definition of a quasi-affine constraint are the following: $+$, $-$, $\times$, $/$, `mod`, $\wedge$, $\vee$, $<$, $\leq$, $>$, $\geq$, $==$, $\neq$. The ternary operator ? is also allowed. The right hand side argument of / and `mod` has to be a positive integer literal, and one of the arguments to $\times$ has to be an integer literal [3].

At the basic level, the polyhedrons represented in the polyhedral model are *convex* polyedrons. Figure 3-2 gives a graphical representation of the difference between a convex and a non-convex polyhedron. All interior points in the convex polyhedron

[1]Loop distribution here is different than distribution used in the next chapter, which corresponds to distributed *execution*

```
1  for (i=0; i<100; i++) {
2    for (j=0; j<10; j++) {
3      A[i,j] = A[i,j] + 4;
4      B[i,j] = A[i,j] + 5;
5    }
6  }
```

(a) Original, untransformed loop nest

```
1  for (i=0; i<100; i++) {
2    for (j=0; j<10; j++) {
3      A[i,j] = A[i,j] + 4;
4    }
5  }
6  for (i=0; i<100; i++) {
7    for (j=0; j<10; j++) {
8      B[i,j] = B[i,j] + 5;
9    }
10 }
```

(b) Distribute loops i and j and create two separate loop nests.

```
1  for (ti=0; ti<50; ti++) {
2    for (tj=0; tj<5; tj++) {
3      for (ii=0; ii<2; ii++) {
4        for (jj=0; jj<2; jj++) {
5          A[ti*2+ii,tj*2+jj] = A[ti*2+ii,tj*2+jj] + 4;
6          B[ti*2+ii,tj*2+jj] = A[ti*2+ii,tj*2+jj] + 5;
7        }
8      }
9    }
10 }
```

(c) Tile loops i and j using a tile size of 2

```
1  for (j=0; j<10; j++) {
2    for (i=0; i<100; i++) {
3      A[i,j] = A[i,j] + 4;
4      B[i,j] = A[i,j] + 5;
5    }
6  }
```

(d) Interchange loops i and j

Figure 3-1: Original two-level loop nest (3-1a) and three example loop nest transformations under the polyhedral model.

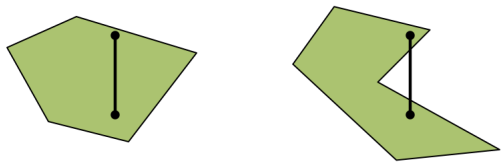


Figure 3-2: A convex (left) vs non-convex polyhedron.

can be connected by a line that stays in the interior, while the non-convex one cannot [26]. It is possible to apply the polyhedral model to non-convex sets since non-convex sets are just unions of convex sets, but that is beyond the scope of this paper.

Once the iteration domain of a loop nest is described, loop transformations are applied. These transformations are defined using *maps*, which are relations between integer sets. Take the code in figure 3-1a for example. A and B each share the same iteration domain, but we would write them separately as

$$\{A[i,j] : 0 \leq i < 100 \ \wedge \ 0 \leq j < 10\}$$

and

$$\{B[i,j] : 0 \leq i < 100 \wedge 0 \leq j < 10\}$$

(we borrow the notation for iteration domains and maps from [53, 55]). The loop distribution map in figure 3-1b is just the identity transformation: distributing loops is achieved by ordering computations, discussed later. The identity transformation is the following:

$$\{A[i,j] \rightarrow A[i,j] : 0 \leq i < 100 \ \wedge \ 0 \leq j < 10\}$$

$$\{B[i,j] \rightarrow B[i,j] : 0 \leq i < 100 \ \wedge \ 0 \leq j < 10\}$$

For the loop tiling in figure 3-1c, we would give:

$$\{A[i,j] \rightarrow A[ti,tj,ii,jj] : ti = floor(i/2) \ \wedge \ tj = floor(j/2) \ \wedge \ ii = i \bmod 2 \ \wedge \ jj = j \bmod 2 \ \wedge \ 0 \leq i < 100 \ \wedge \ 0 \leq j < 10\}$$

$$\{B[i,j] \rightarrow B[ti,tj,ii,jj] : ti = floor(i/2) \ \wedge \ tj = floor(j/2) \ \wedge \ ii = i \bmod 2 \ \wedge \ jj = j \bmod 2 \ \wedge \ 0 \leq i < 100 \ \wedge \ 0 \leq j < 10\}$$

And finally, the loop interchange in 3-1d gives:

$$\{A[i,j] \rightarrow A[j,i] : 0 \leq i < 100 \ \wedge \ 0 \leq j < 10\}$$

$$\{B[i,j] \rightarrow B[j,i] : 0 \leq i < 100 \ \wedge \ 0 \leq j < 10\}$$

This mapping is part of a larger concept known as the *schedule*. The schedule also contains information on how the computations are ordered in relation to one another. In all of the examples in figure 3-1, computation A needs to come before B. More information on the ordering is given in section 3.2.

Additionally, an *access relation* needs to be provided for each computation. An access relation is another map used to specify where the data produced at each iteration should be stored relative to a buffer (i.e. an array or storage location). For our previous examples, all of them store data in a contiguous buffer, so they all have the same access relation for A and B. The relation is:

$$\{A[i,j] \rightarrow A_{buff}[i,j]\}$$

$$\{B[i,j] \rightarrow B_{buff}[i,j]\}$$

3.1.2 Code Generation from the Polyhedral Model

After all the necessary components for a polyhedral program are specified, the iteration domain needs to be transformed with the schedule. This is done by applying the map for the schedule to the iteration domain. This resulting representation is then converted into an intermediate representation, such as an abstract syntax tree (AST). This requires finding an efficient set of nested loops that visits all points in the iteration domain exactly once. Heuristics that attempt to minimize code size and reduce control flow can be applied, but often, minimizing one can negatively impact the other [53]. This process is known as *scanning polyhedra* [5]. An understanding of the algorithms for a scan are not important here, but as we will see in section 3.2, the resulting AST from the scan is used to generate the final backend code.

3.2 TIRAMISU

TIRAMISU is a middle-end compiler based on the polyhedral model that abstracts away the underlying architecture so the programmer does not need to worry about backend code generation. Rather, TIRAMISU takes the programmer's architecture-independent representation and generates architecture-dependent backend code. The programmer simply provides a set of scheduling commands directing the code generator. TIRAMISU was initially developed with shared memory architectures in mind, but with the work done in this thesis, it has been extended to distributed CPU and GPU backends. Additional work done in [10] created an FPGA backend for TIRAMISU.

At the core of TIRAMISU is a unique three-layer design that breaks down the components of the polyhedral model to give a program representation that is much easier to transform. The top layer (Layer I) is known as the *abstract computation layer*. This layer describes the computations of an algorithm using iteration domains and producer-consumer relations between the computations. The middle layer (Layer II) is the *computation placement layer*, which applies loop transformations to the iteration domain, specifies the ordering among computations, and says where to compute a computation (such as on a CPU, on the GPU, etc). The bottom layer (Layer III) is the *concrete computation layer*, and this layer provides the access relations describing the data layout of produced values [53]. Figure 3-3 shows how the different parts of TIRAMISU connect together.

With the abstractions provided by the layered design of TIRAMISU, programmers can modify independent parts of their program without having to worry about changing all the code. For example, programmers can choose different loop transformations to apply in Layer II without having to worry about modifying the computations in Layer I, or modifying the data layout in Layer III. The commands available in Layer II and Layer III are referred to as a *scheduling language* as their purpose is to define the scheduling of the algorithm rather than the algorithm itself.

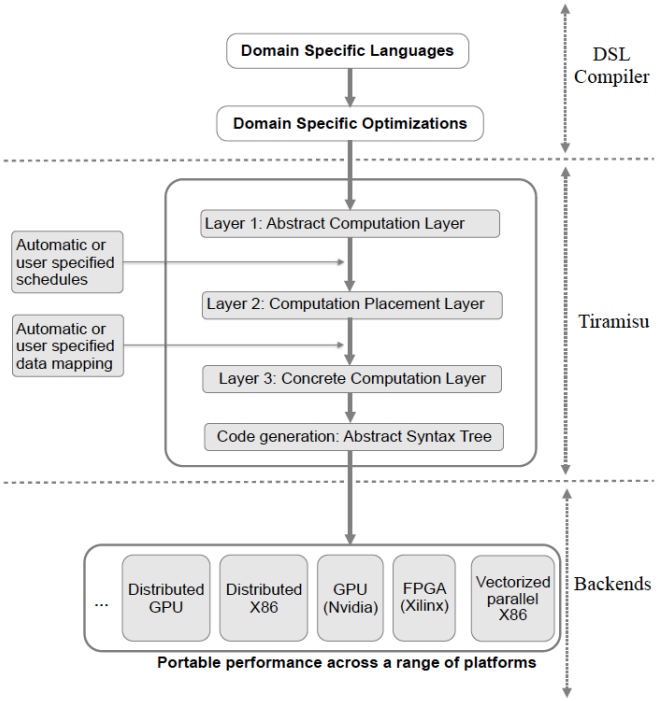


Figure 3-3: TIRAMISU overview [53].

3.2.1 Layer I: Abstract Computation Layer

In this layer, we specify the computations without considering the ordering, loop optimizations, or data layout. Going back to the blur example from section 1.0.1, we would have two computations: cx, representing the horizontal blur, and cy representing the vertical blur. We specify these two computations in Layer I as follows:

$$\{cx(i,j) : 0 \leq i < I \ \wedge \ 0 \leq j < J\} : (input(i,j) + input(i,j+1) + input(i,j+2))/3$$

$$\{cy(i,j) : 0 \leq i < I \ \wedge \ 0 \leq j < J\} : (cx(i,j) + cx(i+1,j) + cx(i+2,j))/3$$

We can easily see the producer-consumer relationship here, where cx consumes from $input$ and produces values consumed by cy.

3.2.2 Layer II: Computation Placement Layer

Layer II provides the schedule, which comprises loop transformations, computation ordering, and computation placement. In TIRAMISU, all of this information is stored

in a compact representation known as the *time-processor* vector. The time dimension is used for ordering, and the processor dimension gives placement. Say we wanted to tile cx and cy by 5, order cx before cy, and parallelize the outer resulting loops of both. The Layer II time-processor representation would be as follows:

$$\{cx(i,j) \rightarrow cx[0, ti(cpu), 0, tj, 0, ii, 0, jj, 0] : ti = floor(i/5) \wedge tj = floor(j/5) \wedge ii = i \bmod 5 \wedge jj = j \bmod 5 \wedge 0 \leq i < I \wedge 0 \leq j < J\}$$

$$\{cy(i,j) \rightarrow cy[1, ti(cpu), 0, tj, 0, ii, 0, jj, 0] : ti = floor(i/5) \wedge tj = floor(j/5) \wedge ii = i \bmod 5 \wedge jj = j \bmod 5 \wedge 0 \leq i < I \wedge 0 \leq j < J\}$$

Let us break down this representation. First off, we see the tiling constraints for ti, tj, ii, and jj as we did in the previous section on the polyhedral model. Now, there is also a processor tag on ti represented by (cpu). This says that the loop for ti should be parallelized on the CPU. In this form, $ti = 0$ would go on CPU 0, $ti = 1$ would go on CPU 1, and so on. Finally, there are constants (0 and 1) between each of the loop levels. These values are *static dimensions*, meaning their values are constants. In contrast, ti, tj, ii, and jj are dynamic dimensions. The static dimensions are used for ordering the computations. Ordering is done via a lexicographic comparison. If we remove the dynamic dimensions, we would have $cx[0, 0, 0, 0, 0]$ and $cy[1, 0, 0, 0, 0]$. Comparing each dimension lexicographically, we see that $cy > cx$, so cy should come after cx. Furthermore, cy is after cx in the 0th dimension, which means the entire loop nest for cy should come after the loop nest for cx.

3.2.3 Layer III: Concrete Computation Layer

In the final layer, the data layout for each of the values is specified. This data layout says how the values produced at each iteration should be stored in a buffer. For the blur again, we want to store the values produced by cx into an intermediate buffer cx_{buff}, and the output values produced by cy in another buffer, cy_{buff}. Additionally, we want to store values contiguously in the buffers. Each buffer will be

TIRAMISU Layer II Scheduling Commands C is a TIRAMISU computation; i and j are iterators	
Command	**Description**
C.interchange(i,j)	Interchange levels i and j
C.shift(i, s)	Shift level i by s iterations
C.split(i, s, i0, i1)	Split level i by s into i0 and i1.
C.vectorize(i, v)	Vectorize level i by v.
C.parallelize(i)	Parallelize level i.
C.unroll(i, v)	Unroll level i by v.
C.after(P, i)	Order C after computation P at level i.
C.inline()	Inline C where used.

Table 3.1: Example TIRAMISU scheduling commands that transform the iteration domain in Layer I [53].

two-dimensional. The necessary access relations are shown below:

$$\{cx(i,j) \rightarrow cx_{buff}[i,j]\}$$

$$\{cy(i,j) \rightarrow cy_{buff}[i,j]\}$$

Three different buffer types are available: `a_input`, `a_output`, and `a_temporary`. `a_input` and `a_output` indicate buffers allocated and freed explicitly by the programmer for the purposes of passing in input to the TIRAMISU pipeline (`a_input`) and getting results from the pipeline (`a_output`). `a_temporary` delegates allocation and freeing to TIRAMISU.

3.2.4 Scheduling Language

TIRAMISU's scheduling language is built from affine maps that are applied to the iteration domain in Layer I. These maps provide the programmer a simple API for describing complex loop transformations. TIRAMISU comes with a set of predefined scheduling commands for commonly used loop nest transformations. These scheduling commands are composable, so many other transformations can be built from this initial set, but programmers can add in their own scheduling commands as needed. Table 3.1 gives some example scheduling commands currently in TIRAMISU [53].

3.2.5 Code Generation in TIRAMISU

The first step in generating architecture-dependent backend code from TIRAMISU is generating the AST from the polyhedral representation. All of the polyhedral components in TIRAMISU are represented using the Integer Set Library (ISL) [55]. This library provides representations for sets and maps and all the operations needed to define the quasi-affine constraints. It also generates the AST used for code generation in TIRAMISU.

Once we have the AST, it is transformed into another intermediate representation (IR) that includes explicit loop structures, architecture-dependent details, etc. We currently use the IR from Halide [41]. From this IR, the final target code is generated and compiled.

The result of running a TIRAMISU program is an object file that contains the compiled pipeline wrapped in a function. This function can be linked into new or existing code and just needs to be called with appropriate input and output buffers as specified in the TIRAMISU definition.

Chapter 4

Distributed, Heterogeneous, and Cooperative Code Generation

The work for this thesis implements several primitives and scheduling functions for generating distributed and heterogeneous backend code. These backends can also be combined together to achieve cooperative heterogeneous execution, giving the ability to utilize as many compute resources as possible.

In this chapter, we look at the modifications made to the existing TIRAMISU code base. New computation types and scheduling commands were added into Layer II and Layer III. Layer I code remains unchanged. A new GPU backend was implemented as well to replace the existing single-GPU backend. With our implementation, programmers can easily switch between distributed and heterogeneous backends (or combinations of the two) with limited changes to existing code.

4.1 Scheduling for Distributed, GPU, and Cooperative Execution

This section describes the scheduling primitives and functions made available to the programmer for scheduling distributed, GPU, and cooperative code.

```
1 ...
2 var r, x, y;
3 computation c1("{[r,x,y]: 0<=r<5 and 0<=x<100 and 0<=y<100}",
4                c0(x,y)*7, true, p_float32, &fct);
5 c1.tag_distribute_level(r, true);
6 ...
```

(a) Using `tag_distribute_level` in TIRAMISU

```
1 distributed for (int r=0; r<5; r++) {
2   for (int x=0; x<100; x++) {
3     for (int y=0; y<100; y++) {
4       c1(x,y) = c0(x,y)*7;
5     }
6   }
7 }
```

(b) Pseudocode for the resulting loop nest.

Figure 4-1: TIRAMISU code for creating a distributed loop nest with `tag_distribute_level` and the resulting pseudocode for the loop nest.

4.1.1 Distributed

For distributed execution, we generate MPI code based on the schedule provided by the programmer. There are two main components to a programmer's distributed schedule: breaking up the computations into individual processes and transferring data. Data transfers are discussed in section 4.1.3. Breaking up the computation involves assigning parts of the loop nests to individual processes. The main scheduling command for this is `tag_distribute_level`. A loop level tagged with `tag_distribute_level` indicates that every iteration of that loop level corresponds to a distributed process. We borrow the term `rank` from MPI to refer to processes, so each iteration is a different rank, which we refer to with the pre-declared `rank` variable. This allows the programmer to condition on the rank so that they can schedule ranks independently. Figure 4-1a shows an example of using `tag_distribute_level` in the TIRAMISU syntax and figure 4-1b shows the pseudocode of the resulting loop. This code distributes the outer loop over 5 different ranks.

The full signature of `tag_distribute_level` is as follows:

```
void tag_distribute_level(var L, bool drop_rank_iter)
```

where `L` is the loop level to distribute, and `drop_rank_iter` says whether or not to use the value of the process' `rank` when computing the index into the buffers. Typically, when we distribute a computation, we want the generated code to index into the buffers starting from 0, which means we do not want the outer loop iterator (i.e. the rank iterator) used in the calculation of the index. If `drop_rank_iter`, then we remove the iterator from the indexing computation.

For the case where we want to schedule `ranks` independently, we introduce a scheduling command called `separate_at`, which separates a loop nest into multiple loop nests, effectively giving us separate computations. These computations can then be scheduled independently. This is important because different ranks may need to execute different operations. For example, if all the data starts on one node, we would need to schedule that one node to send the data out to all the other nodes, which would require a different schedule than the other nodes. Figure 4-2a shows the previous example, but with the addition of `separate_at` on the outer loop level `r`. The call to `separate_at` on line 5, gives two computations, which are accessed with `get_update`, as shown on lines 6 and 7. The resulting loop nest pseudocode is shown in figure 4-2b. Note that we ignore the ordering commands that would be included in the tiramisu IR. The signature of `separate_at` is as follows:

```
void separate_at(var L, vector<expr> separate_points, expr extent)
```

where `L` is the loop level to distribute, `separate_points` defines each point in the iteration vector of the loop level to separate the loop, and `extent` is the extent of loop level `L`. The other scheduling commands in TIRAMISU, are composable with `tag_distribute_level` and `separate_at`.

4.1.2 GPU

The commands in the previous section also apply to computations that we want to run on the GPU. However, running on the GPU requires the programmer to tag loop levels for GPU execution. The scheduling command `tag_gpu_level2`[1] is used

[1] `tag_gpu_level` uses the original GPU backend, whereas `tag_gpu_level2` uses the new backend

```
...
var r, x, y;
computation c1("{[r,x,y]: 0<=r<5 and 0<=x<100 and 0<=y<100}",
                c0(x,y)*7, true, p_float32, &fct);
c1.separate_at(r, {1}, 5);
c1.get_update(0).tag_distribute_level(r, true);
c1.get_update(1).tag_distribute_level(r, true);
...
```

(a) Using `separate_at` and `tag_distribute_level` in TIRAMISU.

```
distributed for (int r=0; r<1; r++) {
  for (int x=0; x<100; x++) {
    for (int y=0; y<100; y++) {
      c1(x,y) = c0(x,y)*7;
    }
  }
}
distributed for (int r=1; r<5; r++) {
  for (int x=0; x<100; x++) {
    for (int y=0; y<100; y++) {
      c1(x,y) = c0(x,y)*7;
    }
  }
}
```

(b) Pseudocode for the resulting loop nests.

Figure 4-2: TIRAMISU code for using `separate_at` and `tag_distribute_level` in the resulting pseudocode for the loop nests.

to tag loop levels as either GPU threads or blocks. The signature to use for tagging a two-level loop nest is the following:

```
void
tag_gpu_level2(var L_block, var L_thread, int op_sequence);
```

where `L_block` is the loop level to map to GPU blocks and `L_thread` is the loop level to map to GPU threads. `op_sequence` is the operation sequence to use when launching the GPU function. More detail on the GPU backend is given in section 4.3.3.

4.1.3 Data Transfers

For data transfers, we implement five data types and a function for creating the transfer. The data types are `xfer_prop`, `send`, `recv`, `send_recv`, and `wait` and the function is `create_xfer`. We also introduce a special scheduling command for data transfers called `collapse_many`.

`xfer_prop` is a `class` that takes a vector of `xfer_attr`s defining the type of data transfer to generate code for (for example, a non-blocking, asynchronous send). Possible `xfer_attr`s include: `SYNC`, `ASYNC`, `BLOCK`, `NONBLOCK`, `MPI`, `CUDA`, `CPU2CPU`, `CPU2GPU`, and `GPU2CPU`. An integer id can optionally be specified to represent an *operation sequence*. An operation sequence is an array of operations that run sequentially. It is similar to a CUDA stream, but abstracted so that it can generate code for other backends that support that type of execution model.

`send`, `recv`, and `send_recv` define different transfer types. `send` and `recv` are used for backends that generate separate send and receive functions for the source and destination ranks (i.e. MPI). `send_recv` is used for backends that combine the two operations into a single function (i.e. CUDA). Programmers create these transfers with the `create_xfer` function. Transfers are specified like computations; they require an iteration domain and an expression to consume. There are two versions of `create_xfer`: one creates individual `send` and `recv` operations, and the other creates just a `send_recv` function. The two signatures are as follows:

```
xfer
create_xfer(string send_iter_domain, string recv_iter_domain,
            expr send_dest, expr recv_src,
            xfer_prop send_prop, xfer_prop recv_prop,
            expr xfer_expr, function *fct);
```

and

```
xfer
create_xfer(string iter_domain, xfer_prop prop, expr xfer_expr,
            function *fct);
```

The first function creates individual `send` and `recv` operations, and requires an iteration domain for both. The source and destination nodes (`send_dest` and `recv_dest`) can have different properties (e.g. one could be blocking, the other could be non-blocking, etc), so they each take a separate set of `xfer_prop`s. An example of using the first one is in figure 4-3 and the second in figure 4-4. For the combined send and receive in the second figure, the destination node is determined with an outside function. We use this type of send and receive for GPU transfers, so the destination is set at runtime when the programmer selects which GPU they want to use.

With certain transfer properties (such as non-blocking MPI transfers and asynchronous CUDA transfers), there needs to be a mechanism to determine when the transfer has completed. In MPI, this mechanism is an `MPI_Request` and in CUDA, it is a `CUevent`. We abstract this with a `wait` object. When a `wait` is encountered during execution, the current thread blocks until the `computation` waited on is complete. The `wait` signature is the following:

```
wait wait(expr rhs, xfer_prop prop, function *fct);
```

where `rhs` is an access into the `wait` array and `prop` gives the type of `wait`.

The `collapse_many` scheduling function is used to merge together data to transfer. Usually, it is better to send messages bigger than just one element, but with the way transfers are defined, they default to sending one element at a time. This function collapses a loop down into a single iteration and packs all of the elements accessed in

```
var r, x, y;
computation c1("{[r,x,y]: 0<=r<5 and 0<=x<100 and 0<=y<100}",
               c0(x,y)*7, true, p_float32, &fct);
c1.separate_at(r, {1}, 5);
c1.get_update(0).tag_distribute_level(r, true);
c1.get_update(1).tag_distribute_level(r, true);
xfer_prop prop(p_float32, {SYNC, BLOCK, MPI, CPU2CPU});
xfer send_and_recv = c1.get_update(0).create_xfer("{[r,q,y,x]:
   0<=r<1 and 1<=q<5 and 0<=x<100 and 0<=y<100}", "{[r,y,x]:
   1<=r<5 and 0<=x<100 and 0<=y<100}", q, 0, prop, prop,
c0(x,y), &fct);
```

(a) Creating a transfer with separate `send` and `recv` functions.

```
distributed for (int r=0; r<1; r++) {
  for (int q=1; q<5; q++) {
    for (int x=0; x<100; x++) {
      for (int y=0; y<100; y++) {
        node r sends c0(x,y) to node q;
      }
    }
  }
}
distributed for (int r=1; r<5; r++) {
  for (int x=0; x<100; x++) {
    for (int y=0; y<100; y++) {
      node r recvs c0(x,y) from node 0;
    }
  }
}
distributed for (int r=0; r<1; r++) {
  for (int x=0; x<100; x++) {
    for (int y=0; y<100; y++) {
      c1(x,y) = c0(x,y)*7;
    }
  }
}
distributed for (int r=1; r<5; r++) {
  for (int x=0; x<100; x++) {
    for (int y=0; y<100; y++) {
      c1(x,y) = c0(x,y)*7;
    }
  }
}
```

(b) Pseudocode for the resulting loop nests.

Figure 4-3: Creating data transfers that have independent `send` and `recv` functions with `create_xfer`.

```
1 var r, x, y;
2 computation c1("{[r,x,y]: 0<=r<5 and 0<=x<100 and 0<=y<100}",
     c0(x,y)*7, true, p_float32, &fct);
3 c1.separate_at(r, {1}, 5);
4 c1.get_update(0).tag_distribute_level(r, true);
5 c1.get_update(1).tag_distribute_level(r, true);
6 xfer_prop prop(float, {SYNC, BLOCK, CUDA, CPU2GPU});
7 xfer send_recv = c1.get_update(0).create_xfer("{[q,y,x]: 1<=q<5
      and 0<=x<100 and 0<=y<100}", prop, c0(x,y), &fct);
```

(a) Creating a transfer with combined `send` and `recv` functions.

```
1  distributed for (int r=0; r<1; r++) {
2    for (int q=1; q<5; q++) {
3      for (int x=0; x<100; x++) {
4        for (int y=0; y<100; y++) {
5          node r sends c0(x,y) to GPU q, which receives it;
6        }
7      }
8    }
9  }
10
11 distributed for (int r=0; r<1; r++) {
12   GPU for (int x=0; x<100; x++) {
13     GPU for (int y=0; y<100; y++) {
14       c1(x,y) = c0(x,y)*7;
15     }
16   }
17 }
18 distributed for (int r=1; r<5; r++) {
19   GPU for (int x=0; x<100; x++) {
20     GPU for (int y=0; y<100; y++) {
21       c1(x,y) = c0(x,y)*7;
22     }
23   }
24 }
```

(b) Pseudocode for the resulting loop nests

Figure 4-4: Creating a single send and receive with `create_xfer`.

that loop level into a single message. Currently, this is only supported for contiguous data. If non-contiguous data needs to be sent, the programmer will have to create a separate computation that first packs the data together into a contiguous buffer, and then use that with the `collapse_many` function.

`send`, `recv`, `send_recv`, and `wait` can be ordered using the predefined scheduling commands such `before`, `after`, etc.

4.2 Data Layout

In a transfer, the sender acts as the producer and the receiver is the consumer. This requires access functions for `recv` and `send_recv` computations. Additionally, any operation that executes in a non-blocking manner (i.e. control immediately returns), require a second access function for storing their associated `wait`s. For example, each time a non-blocking transfer is launched, it generates a `wait`, which needs to be stored in a `buffer`. This special access is set with the `set_wait_access` function. Like `set_access`, this just takes an access relation to a buffer.

A new buffer type, `a_temporary_gpu`, is also provided. This is similar to the existing `a_temporary` type, but it allocates memory on the GPU instead.

Distributed and communicated data does not require any special handling on the programmer's part–they write and read from the same Tiramisu buffers. The backend functions, whether they be CUDA or MPI, may move data between different (non-Tiramisu) buffers, but that is all dependent on the actual implementation and does not concern our work.

All of these different primitives, functions, and scheduling commands can be combined together for cooperative computing. By using various combinations of non-blocking and asynchronous commands, as well as assigning separate processes/ranks to chunks of code, the programmer can configure some amount of work to run on the GPU and some on the CPUs. There are numerous different configurations to run in–the correct combination depends on the application at hand. Trying these different configurations becomes much easier thanks to the abstractions and scheduling

commands implemented here, as evidenced at the end of section 1.0.1.

4.3 Code Generation

4.3.1 ISL AST Generation

For the most part, generating the AST from ISL remains the same as in TIRAMISU. The main difference comes during index expression generation. This step generates `isl_ast_expr`s representing the index expressions into buffers. For distributed operations, programmers can choose to drop the distributed loop level iteration from the index expressions. If that is the case, this step removes the specified level by forcing the dimension in the time-processor vector to be 0.

4.3.2 Distributed Backend

For the MPI functions that we support, we implemented wrappers that both call the MPI function, as well as perform error checking. Before code generation, we run a lowering pass to check for any `send`, `recv`, and `wait` computations. This pass replaces those with call objects for the appropriate wrapper function, and also generates the necessary function arguments. During code generation, these calls get converted to Halide calls, which then generate LLVM.

We also process `send` and `recv` separately from other computations since independently they do not fit the expected producer-consumer model of computations. A `send` by itself does not explicitly have a left-hand side (it does not write to any buffer), and a `recv` by itself has no right-hand side (it only writes to a buffer). Together, they fit the producer-consumer model, but we need extra bookkeeping to keep track of the other side of the communication operation.

When a non-distributed loop level is encountered while traversing the ISL AST, the loop level is converted into a for loop, which in our case is a `Halide::For` object. However, if a distributed loop level is encountered, we generate a conditional statement instead that checks the value of `rank`. If `rank` is within the original bounds of

```
1 for (int i=0; i<5; i++) {
2   for (int j=0; j<10; j++) {
3     ...
4   }
5 }
```

(a) The original non-distributed loop nest

```
1 int rank = tiramisu_MPI_Comm_rank();
2 if (rank >=0 || rank<5) {
3   for (int j=0; j<10; j++) {
4     ...
5   }
6 }
```

(b) The distributed loop nest

Figure 4-5: Converting a for loop to a conditional.

that loop, then the conditional evaluates to `true`. An example of a for loop being represented as a distributed loop is shown in figure 4-5. This code shows a two-level loop nest (figure 4-5a) and the resulting loop nest (figure 4-5b) if we distributed the outermost level. With this, only MPI processes with a `rank` between 0 and 5 will execute the code in that loop nest. An additional change had to be made in the actual ISL code base to handle a certain corner case where we split a loop such that the outer loop only has one iteration. Usually, ISL would either mark this loop as *eliminated* or *degenerate*. An eliminated loop is completely removed as if it did not exist and a degenerate loop is removed, but a reference to its loop iterator is replaced with an affine expression. In any case, if we want to mark that outer loop as distributed, we want to make sure ISL does not remove it, otherwise the incorrect loop level would be tagged. To fix this, we tag the loop iterator with a special marker. In ISL, we change to the code to look for that marker. If it is found, we force ISL to keep the loop.

4.3.3 GPU Backend

While Halide provides a GPU backend that generates PTX code for NVIDIA GPUs, we chose to implement our own CUDA backend so that we could support additional features, such as asynchronous operations and multi-gpu execution. We utilize the CUDA Driver API for this [33]. Like MPI, we implement wrappers to the necessary CUDA functions and run a lowering pass that converts `send_recv` and `wait` computations into the appropriate function calls. Additionally, any buffer allocations with a `a_temporary_gpu` tag are converted into function calls to the wrapper for `cuMemAlloc`. These buffer allocations are automatically freed with another call to the wrapper function for `cuMemFree`.

Currently, we do not utilize shared memory, so the computations to run on the GPU are mapped directly into kernels. Kernel code generation is a multi-step process. First, when a loop nest marked for the GPU is encountered during the normal Tiramisu code generation stage, we intercept the process so that we can generate a kernel instead of a `for` loop. Since the programmer marks which loop iterators to map to CUDA threads and blocks, we substitute in the appropriate CUDA block and thread parameters for the original loop iterators. Combined with index expression generation, this gives us the access into the GPU buffers. Figure 4-6 shows an example of what a loop nest looks like as a kernel. In the first part, figure 4-6a, we just have a simple two level loop nest and on line 4, the linearized array index is computed. Figure 4-6b shows one way that the loop nest can be tagged for GPU execution. Line 1 specifies a 1D CUDA grid with 1024 blocks, and line 2 says that each block is also 1D and contains 64 threads. This representation looks at each matrix as a 1D array, where the array is 1024×64 elements wide. Figure 4-7 shows how this array version maps to the threads and the blocks. It would map as a 1D grid. Of course, we could also represent each matrix as a 2D grid of blocks rather than 1D if we wanted to.

Figure 4-6c shows the resulting kernel. On lines 3 and 4, iterators `i` and `j` are replaced with the built in CUDA variables, `blockIdx.x` and `threadIdx.x`, respectively. These variables are substituted into the linearized index expression on line

5. Finally, the kernel is launched on line 10 with the appropriate grid and block parameters (1024 and 64, respectively).

When a kernel is generated, a wrapper function to the kernel is also generated. The wrapper handles copying up any necessary constants to GPU memory and performs appropriate synchronization. It also packs the kernel arguments together and launches the kernel. Two different versions of the wrapper are created: the first one creates a CUDA event after the kernel launch so that later computations can synchronize on this kernel. The second one does not create an event. The programmer implicitly dictates which one is used based on how they implement any `wait` computations in their code.

Both the kernel and the wrapper are written out and compiled into object files, which get linked into the generated code at runtime. This lets the programmer see what kind of CUDA is generated for their kernel; programmers can even modify the generated code if they want. We generate CUDA instead of PTX and leave the task of generating optimized PTX from the CUDA code to the NVIDIA compiler, `nvcc` [35].

```
1 float *A, *B, *C...;
2 for (int i=0; i<1024; i++) {
3   for (int j=0; j<64; j++) {
4     int idx = i*64+j;
5     C[idx] = A[idx] + B[idx];
6   }
7 }
```

(a) The original loop nest

```
1 gpu block for (int i=0; i<1024; i++) {
2   gpu thread for (int j=0; j<64; j++) {
3     int idx = i*64+j;
4     C[idx] = A[idx] + B[idx];
5   }
6 }
```

(b) Tagging the loop nest for GPU execution

```
1  __global__
2  void kernel(float *gpu_A, float *gpu_B, float *gpu_C) {
3    int gpu_i = blockIdx.x;
4    int gpu_j = threadIdx.x;
5    int idx = gpu_i*64+gpu_j;
6    gpu_C[idx] = gpu_A[idx] + gpu_B[idx];
7  }
8  ...
9  float *gpu_A, *gpu_B, *gpu_C...;
10 kernel<<<1024,64>>>(gpu_A, gpu_B, gpu_C);
```

(c) The loop nest as a GPU kernel

Figure 4-6: Converting a loop nest to a GPU kernel.

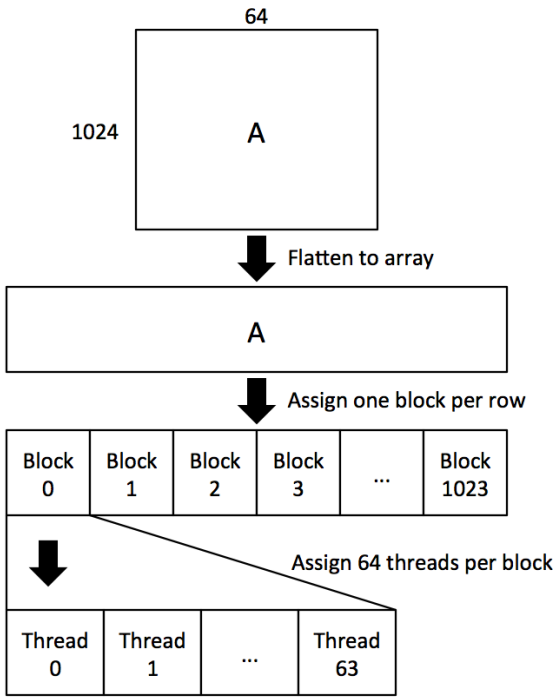


Figure 4-7: Mapping matrix `A` as a 1D grid of 1024 1D blocks, each containing 64 threads. Matrices `B` and `C` would be similarly mapped.

Chapter 5

Experiments

This chapter gives results for various experiments showing the different aspects of the work presented throughout this thesis, namely distributed, heterogeneous, cooperative, and NUMA-aware execution.

2D 3x3 Box Blur The main application we use to demonstrate our work is the 2D 3x3 box blur referenced throughout. As we will see, despite its simple algorithm, the box blur benefits from the different features we discussed throughout this thesis, such as asynchronous operation, heterogeneous excution, and cooperative execution. All experiments here use a 3x3 blur on a float32 matrix with dimensions 16000 by 2000000. The CPU experiments ran on dual-socket 12-core Intel Xeon E5-2695 v2 CPUs, and the GPU experiments used NVIDIA K40 and K80 GPUs. MPI communication uses OpenMPI version 1.6.5.

In this first set of experiments, we assume that the data can be algorithmically generated, so no explicit communication is needed between nodes. Table 5.1 shows the results from optimizing the blur using tiling, parallelization, and vectorization. The optimization schedules we derived for these experiments match the blur schedules presented in the introduction, just with different splitting factors. The blur computation has a lot of parallelism, so we expect parallelization and vectorization to help considerably. Blur also benefits from temporal and spatial locality, so tiling improves cache performance. The next set of experiments shows the blur executed on

3x3 Box Blur–Single CPU Node	
1 Node	18.023s
2 Nodes	7.905s
4 Nodes	7.650s

Table 5.1: Times for tiled, parallelized, and vectorized blur on a single shared memory CPU node.

3x3 Box Blur–GPU	Synchronous	Asynchronous
1 Node, 1 GPU/Node	30.232s	14.756s
1 Node, 2 GPUs/Node	15.156s	6.951s
2 Nodes, 1 GPU/Node	14.965s	7.500s
2 Nodes, 2 GPUs/Node	7.618s	4.008s
4 Nodes, 1 GPU/Node	8.447s	3.824s

Table 5.2: Times for blur on GPUs using synchronous and asynchronous configurations.

the GPU. Table 5.2 shows results using a fully synchronous and fully asynchronous configuration. The asynchronous configuration overlaps communication in both the host-to-device and device-to-host direction. We do one transfer per row, and immediately launch the kernel once the row is transferred to the device. For copying back results, we also copy back one row at a time as soon as the computation for the row is complete. First, notice the performance improvement of the asynchronous version as compared to the synchronous version. In the synchronous version, all the operations happen sequentially (threads within a kernel still run concurrently), so there is no way to benefit from overlapping communication and computation. Figure 5-1 shows a portion of the execution timeline as collected from NVIDIA's `nvprof` GPU profiling tool. The bottom line (labeled Stream 13) shows that the communication and kernels execute sequentially.

Compared to the CPU performance, we are about 2.0× faster with the asynchronous verison. Our initial intuition was that the GPU would be significantly faster, however, the communication between the host and device is a bottleneck. Figure 5-2 shows a portion of the execution timeline for the asynchronous version. The line labeled Stream 13 shows the kernel execution, Stream 14 shows the host-to-device transfers, and Stream 15 shows device-to-host. We can see the overlap clearly, but we

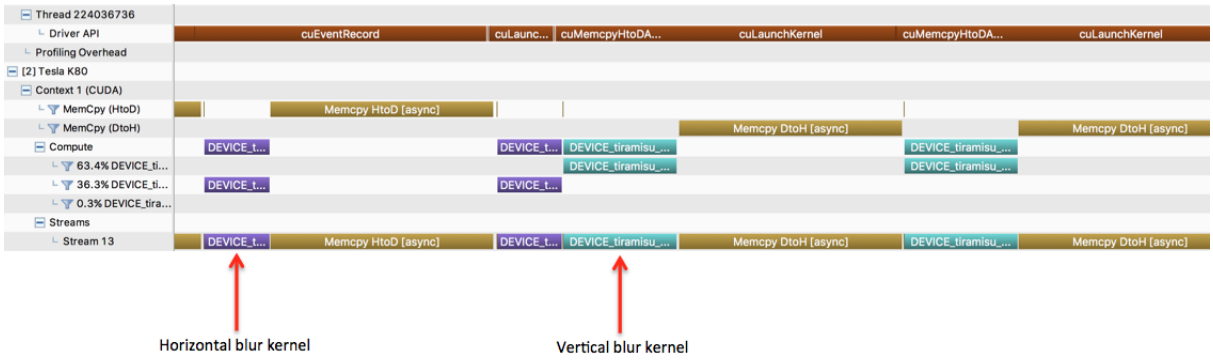


Figure 5-1: Timeline for synchronous execution of the blur.

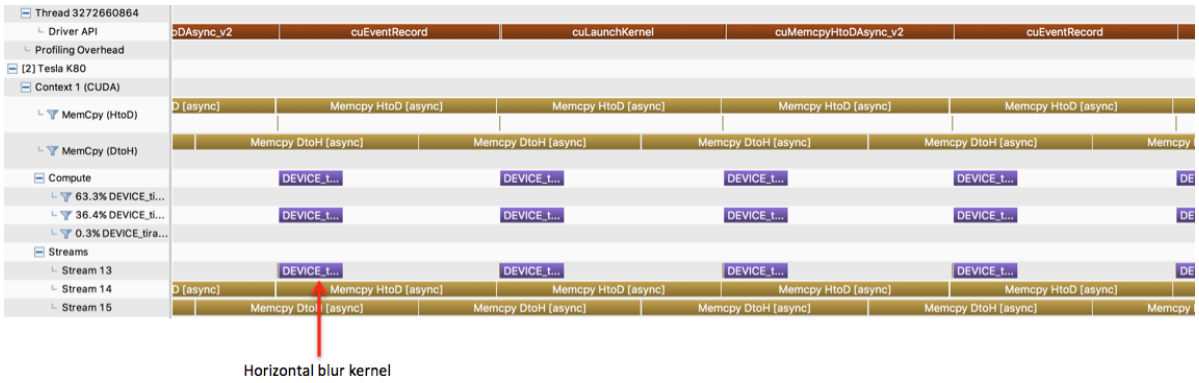


Figure 5-2: Timeline for asynchronous execution of the blur.

also see that the kernel execution time is about 3 times shorter than a given memory transfer. Each kernel has to wait until its row is fully transferred, so the GPU ends up idling while waiting for the data.

The next experiment uses a cooperative configuration. In our cooperative design, we partition part of the rows to execute on the CPU, and the rest on the GPU. We tried four different configurations for partitioning the rows, as shown in table 5.3. The only modification we had to make for the schedule was changing the number of rows each node/device was assigned. For each node, we use 20 CPU cores and 1 GPU. For example, the 2 node configuration would use 40 total CPU cores and 2 GPUs. Graph 5-3 gives an alternative view of the performance. Compared to our best time with GPU execution only, the cooperative configuration gives a speedup up of roughly 3.2×.

Our previous experiments assumed all the data existed on each node already, even the overlapping regions. This next experiment uses a distributed CPU configuration

3x3 Box Blur–Cooperative				
		GPU Rows/CPU Rows		
# Nodes	14000/2000	12000/4000	8000/8000	4000/12000
1	12.905s	11.184s	18.130s	27.363s
2	2.505s	4.211s	8.553s	14.452s
4	1.195s	2.968s	5.267s	6.851s

Table 5.3: Times for blur with cooperative execution using asynchronous execution and communication. We use 20 CPU cores and 1 GPU per node.

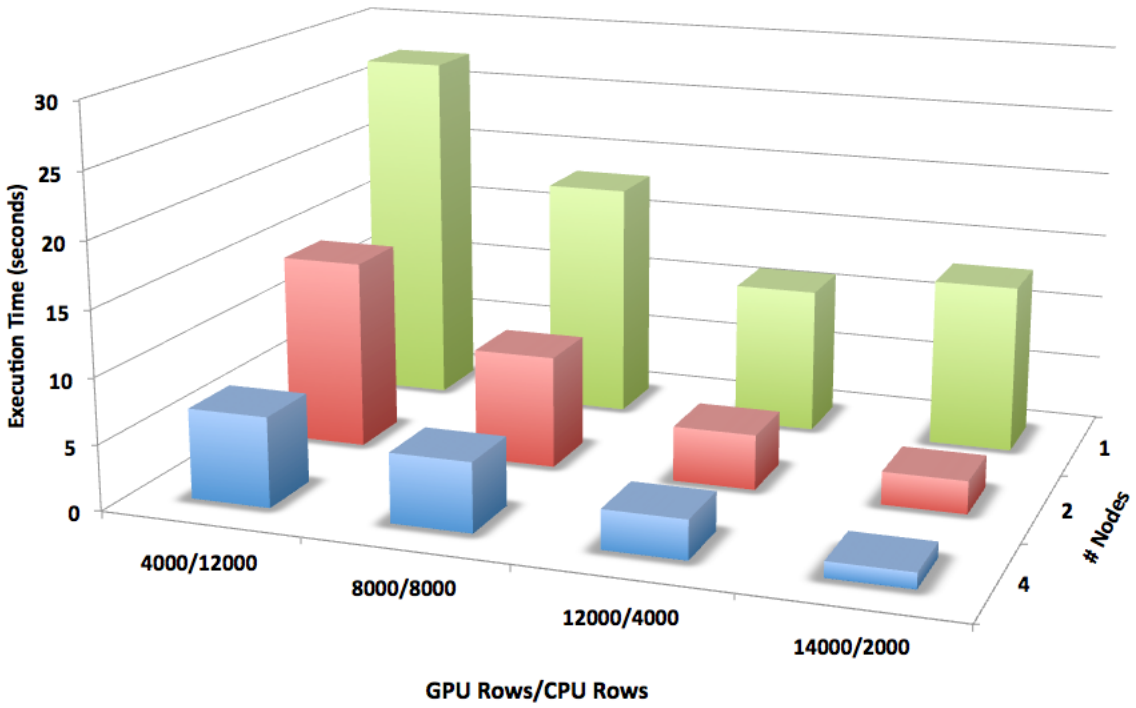


Figure 5-3: 3D chart for blur with cooperative execution using asynchronous execution and communication.

3x3 Box Blur–Transfer Ghost Zones		
	Single msgs	**Packed msgs**
2 Node	22.729s	15.152s
4 Nodes	19.336s	7.615s

Table 5.4: Times for blur on distributed CPUs where the ghost zones must be transferred.

as before, however, we assume that individual nodes need to transfer the ghost zones (the ghost zones are regions of overlap in the data split across nodes). We utilize asynchronous MPI communication. Table 5.4 shows the results. The first column, **single msgs** shows the results if each element of the ghost zone is sent as an individual message, and the second column, **packed msgs**, shows results if we pack all the messages together into one single message using the `collapse_many` function. Sending single messages is much slower–there is a non-trivial overhead associated with initiating a send and receive, particularly when you send thousands of messages back-to-back, as happens here. When we pack all the messages together into one transfer, our results are similar to the results in table 5.1. By using asynchronous communication, we are able to mask the extra time incurred from the transfer, so we do not take any peformance hit.

Our final experiment using the blur shows the advantage of using NUMA-aware execution. The blur is an excellent application for this because of the data reuse patterns it exhibits. We show three results for this in table 5.5. In each result, we execute the blur on a 100000x100000 element float32 matrix with the same tiled and vectorized schedule as before. However, we parallelize across 10 cores instead of 20. For the **matching** and **mismatch** xperiments, we execute only on the cores in socket 0. For the **Non-NUMA** experiment, we let the scheduler decide which cores to run on. For the left column labeled **matching** we bind the memory to socket 0, and in the right column **mismatch**, we bind the memory to socket 1. For **Non-NUMA**, we also let the scheduler decide where to assign the data. As expected, the configuration where the execution and memory are bound to the same socket performs better than the configuration where they are bound to separate sockets. The first configuration

3x3 Box Blur–NUMA		
Matching	**Mismatch**	**Non-NUMA**
5.217s	9.734s	5.977s

Table 5.5: Times for blur where execution and memory are bound to the same sockets (**matching**), separate sockets (**mismatch**), and not bound (**non-NUMA**)

runs approximately 1.8× faster, which is a a significant improvement for such a simple configuration change. The performance when leaving the decision up to the scheduler is close to our manual scheduling, which means the scheduler is able to effectively utilize the NUMA configuration as well.

Comparing with Distributed Halide We ran several experiments comparing the performance of our distributed CPU implementation to Distributed Halide [11]. The 5 benchmarks we use are the following:

> **Convolution**: Performs a simple 2D convolution. We use a 55000x25000x3 dimensional input. Communication is required between nodes.
>
> **cvtColor**: Converts an RGB image to greyscale. We use a 55000x25000x3 dimensional input. Communication is not needed between nodes.
>
> **Gaussian**: Performs a Gaussian blur. We use a 55000x25000x3 dimensional input. Communication is required between nodes.
>
> **Pipeline**: A pipeline fusing together two operations on an image: 1) brightening and 2) creating a negative. We use a 55000x25000x3 dimensional input. Communication is not needed between nodes.
>
> **Sobel**: A convolution used in edge detection algorithms for images. We use a 100000x42500 dimensional input. Communication is required between nodes.

All benchmarks were run on dual-socket 12-core Intel Xeon E5-2695 CPUs using MVAPICH2 2.0 as the MPI implementation.

Graph 5-4 shows the execution time of the 5 benchmarks distributed across 16 CPU nodes for our system against Distributed Halide. Our system outperforms the

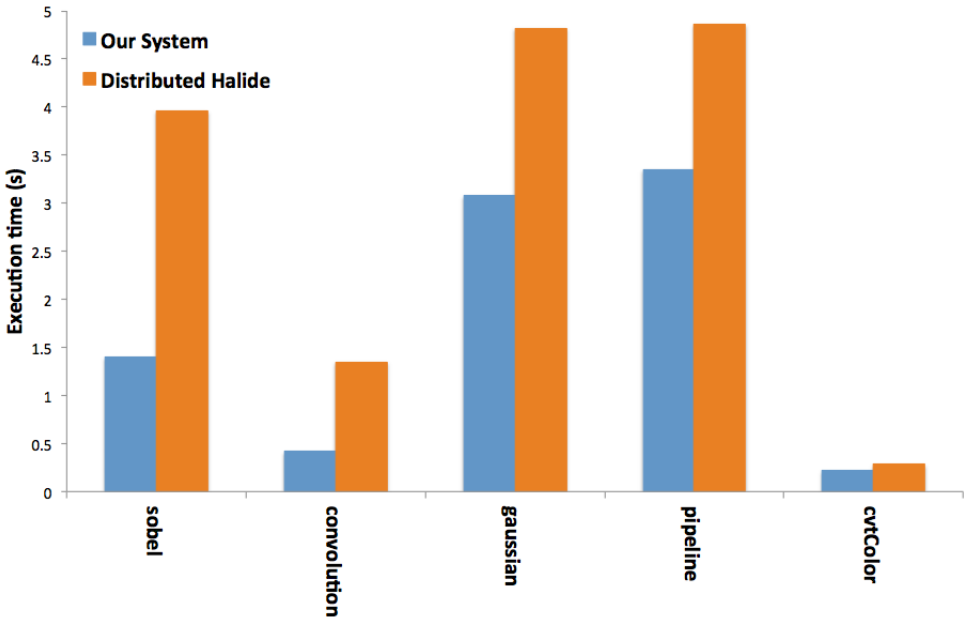


Figure 5-4: Results comparing the performance of our distributed system versus Distributed Halide on 16 CPU nodes.

Distributed Halide implementation on all the benchmarks. For the benchmarks requiring communication, we are faster than Distributed Halide for two reasons: 1) Distributed Halide overestimates the amount of data to transfer; 2) Distributed Halide unnecessarily packs together 0contiguous data. For `cvtColor` and `pipeline`, which do not require any communication, we outperform Distributed Halide thanks to reduced overhead in distribution (`cvtColor`) as well as loop fusion in TIRAMISU (`pipeline`).

The graph in 5-5 shows our performance on the same pipelines, but across 2, 4, 8, and 16 CPU nodes. As expected, the execution time decreases as we scale up the number of nodes for each benchmark.

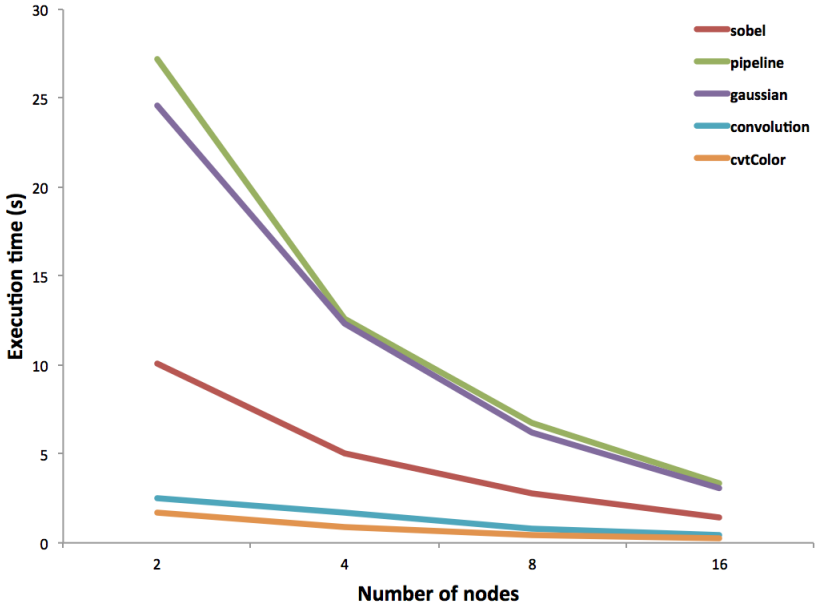


Figure 5-5: The performance of our distributed system across 2, 4, 8, and 16 CPU nodes.

Chapter 6

Discussion

Aside from the actual experimental results, there are several critical pieces related to heterogeneous and cooperative systems that should be noted. One of the main goals we had for this work was getting a sense of what types of abstractions are appropriate for heterogeneous systems. There are many questions we had, including:

1. Exposing functionality: What underlying functionality should be exposed to the programmer?

2. Comparing programming models: What are the similarities and differences in the programming model for the various resources?

3. Exposing architecture-specific optimizations: What optimizations are useful for the various resources and architectures available, and how can programmers access them?

From this work, we begin to put together answers for each of these.

Exposing functionality The libraries used for parallelization, distributing, and GPU execution contain numerous functions and options for generating code. For example, in MPI, there are almost 400 different functions available for communication. However, many of those functions are for different types of transfers. Some are for two-sided communication, while others are for one-sided communiation, collective

communication, RMA (remote memory access), etc. Exposing each of these to a programmer in a model such as TIRAMISU would not be very helpful, so we need an abtraction on top of those, which is what our work provides. If we look at the MPI functions, we can see that they all boil down to transferring some piece of data from point A to point B. Even if there is not an explicit receive anywhere, we can abstract all these different transfers as sends and receives if we think of point A as executing a send and point B as the receive. The programmer should be able to write one set of general communication operations, and then select what the actual backend will be later on. In TIRAMISU, we have the core `send` and `recv` types, as well as the `send_recv` type, which is really just represented underneath as a `send` and `recv`. There is also the `wait` object that represents synchronization for non-blocking type functions. Regardless of whether we use MPI or CUDA, the programmer just has to deal with a single `wait`, rather than one per different backend.

Comparing programming models As we have seen throughout this thesis, each of the different backends, whether it be shared, distributed, or gpu, requires a different way of programming and thinking. With a shared memory system, the programmer sees all of the data in one space. With distributed, the programmer now has to think about where data is and how to get it from point A to point B. And with the GPU, the programmer has to think in terms of kernels instead of the usual for loops. The libraries for the different models are also very different: shared memory has OpenMP and Pthreads, distributed has MPI, and GPU has CUDA (there are numerous other libraries as well). These libraries are full of backend specific functions, which at first glance may not seem related whatsoever. However, we can see similarities if we look at the underlying operations. For example, transfers representing moving data from point A to point B. Another similarity is the notion of "launching" an operation: with shared memory that could be launching a thread; in distributed, it could be launching a process; with the GPU, there is launching a kernel. At a lower level, these operations fit into the producer-consumer type paradigm, so when you look under all the functions (which are really just wrappers to the underlying producers

and consumers), these seemingly different paradigms have many similar constructs.

Exposing architecture-specific optimizations A big difference between the programming models is the underlying hardware they support. For example, a GPU has cache-like storage that can be managed by the programmer, as opposed to caches in a typical shared memory system which are largely hardware managed. So, this memory hierarchy should be exposed to the programmer. Another example is loop tiling. It's not typically helpful on GPUs, but can be very helpful for shared memory systems, so it also needs to be implemented. Looking at MPI and CUDA, we have the different synchronous/asynchronous, blocking/non-blocking operations, which have a big impact on performance. An underlying framework needs to support these different types of communication. We expose these operations through scheduling commands, allowing programmers to specify what type of transfer they want, for example, without having to worry about what the underlying function actually is. The TIRAMISU code generator handles picking out the appropriate function based on the programmers specification.

The scheduling language is also useful for incremental optimization. Scheduling functions provide the advantage of not having to manually change the loop nests, which makes it easier to apply transformations, especially when it comes to composing transformations and generating different backends. However, it can be harder to visualize what a loop nest looks like after a bunch of scheduling commands have been applied, so it can be less straightforward how to decide which optimization to apply next, or even just figure out what the code is actually doing. Several transformation passes are additionally applied through ISL and Halide to the code, so the resulting code may look very different from what the programmer expected. How to visualize the schedules created for a program is still an open question.

Creating a Unified Programming Model

The work done in this thesis gives a step towards understanding how to unify the different programming models for heterogeneous and cooperative execution. There

is still much more research to be done, especially at a higher level. Our code is not intended to be handwritten, rather it should be generated from a higher-level language. The question remains about what that high-level language should look like though. It could be a DSL, which only utilizes certain parts of our work, or it could be a full-fledged programming language focused on creating a unified programming model for distributed, heterogeneous, and cooperative computing.

A full-fledged heterogeneous programming language is an interesting approach, as most current approaches to heterogeneous computing are either library-based, or built on top of an existing programming model, such as the GPU programming model. While this certainly makes heterogeneous programming possible, it is not necessarily a great fix, particularly as new architectures are developed. These approaches try to convert one programming model into another, which can result in missing features, sub-optimal performance, and messier code. As heterogeneous and cooperative computing becomes more important and necessary, it makes sense to take a step back and design a language that provides first class support for these systems.

Getting High Performance through Trial-and-Error

While a lot of current high-performance computing work currently focuses on automating optimizations, our approach moves away from automation. Our scheduling language approach makes it easier to try out different optimizations, but our system is not an automatic optimizer. Automated approaches leave a lot of performance on the table since they rely on heuristics to optimize code. For the high-performance programmer, this automatic approach is not useful because they want to get the best performance. These programmers will revert to hand-optimizing all their code. However, even hand-optimization can lead to suboptimal performance because the programmers can only try out so many optimizations. Getting the best performance is an iterative trial-and-error process. Some optimizations help, some do not. Some work well on their own, and then perform poorly when composed with optimizations. Many optimizations also have parameters to tune as well, such as the tile size in loop tiling transformations. Automated approaches cannot deal with all of this variability,

while manual approaches take too much time.

With our work, we maintain the freedom that comes from hand-optimization, but make it simpler to apply these optimizations. The programmer does not have to sacrific performance for ease of use. The Halide language [41] is an excellent example of the utility of the scheduling language approach. Prior to Halide, programmers would write high-performance image processing code by hand. Switching to Halide, programmers could perform the same optimizations, but were able to try many more compositions of optimizations in a shorter amount of time, leading to improved performance over the expertly hand-optimized code.

Chapter 7

Related Work

The major goal of this work is enabling heterogeneous, cooperative execution through abstractions on the underlying architecture, without sacrificing control over the low-level performance-critical features, such as communication. Prior work is limited in the support of heterogeneous and cooperative execution, and either abstracts the underlying architecture too much, or does not abstract it enough.

Distributed Halide [11] is an extension to the Halide framework supporting distributed execution on CPU nodes. This framework use interval analysis to automatically determine the set of data to transfer between nodes, so the programmer does not have control over the number of messages sent, how much overall data to transfer, or when to initiate a transfer. As seen in the prior results section, we can outperform Distributed Halide by specifying exactly how and what to transfer.

OpenACC [13] is similar to OpenMP and provides pragmas and compiler directives for parallelizing a loop nest on the GPU. However, OpenACC only focuses on a single heterogeneous node; any distributed computation needs to be manually implemented by the programmer. Additionally, the programmer has limited control over data transfers between the CPU and GPU.

Several compiler frameworks for heterogeneous computing exist, but to the best of our knowledge, our work is the first that provides the ability to generate heterogeneous and cooperative code from a single algorithm, while still giving the programmer control over low-level features such as communication.

The Legion parallel programming model [4] provides abstractions that allow the programmer to control features such as communication in heterogeneous, distributed systems, however, programmers must write their own backend-specific code, such as CUDA kernels. From the same group comes the Realm framework, which, like our work, represents the mechanisms necessary to generate distributed, heterogeneous, and cooperative code. However, it lacks the abstractions we provide for loop optimizations thanks to our integration with TIRAMISU.

Dandelion [45] is a compiler and runtime system for cooperative, heterogeneous computing. Programmers provide sequential code and Dandelion attempts to parallelize the code and decide how to map computations to compute devices. The programmer has limited control over the resulting code generation, and must rely on the Dandelion compiler to make such decisions.

Qilin [23] is another framework that automatically maps computations to heterogeneous processing elements. The Qilin API provides a stream-API and threading-API for writing heterogeneous code. In the stream-API approach, programmers write a single Qilin function and the Qilin compiler generates code for heterogeneous architectures. However, the API only provides a limited number of data parallel operations, and defers most of the execution to libraries like MKL and CUBLAS. If a programmer requires more control over the code generation, they must write their own separate code for the different architectures using the threading-API, and then use Qilin to hook the code together. Qilin uses an adaptive learning method to determine the best way to map computations to processing elements. We see Qilin as a complementary approach to TIRAMISU and this work.

dOpenCL [21] provides a unified programming model based on OpenCL which allows distributed, heterogeneous, and cooperative execution. However, programmers are required to write kernels manually for different architectures, and set up all of the necessary code for launching kernels. libWater [16] is a library-extension to OpenCL that provides similar functionality to dOpenCL. But, like dOpenCL, programmers must write their own kernels for different architectures.

Chapter 8

Conclusion

Heterogeneous and cooperative computing is here to stay. With Moore's law nearing its end, programs will need to take as much advantage as possible of available resources if performance gains are to be seen. Currently, GPUs are at the forefront of heterogeneous computing, but even with all the effort currently going into computing with GPUs, there is still much to be done. Programming for a GPU is fundamentally different than programming for CPUs. Even optimizing code for a CPU can still be difficult, despite decades of advances made in optimizing compilers, DSLs, and other high-performance tools. With different configurations, such as distributed memory, handwriting highly-optimized code is a nearly insurmountable task. Thus, tools are needed to abstract optimizations and the low-level differences among architectures, without sacrificing the ability to achieve high-performance.

This thesis aimed to achieve two major goals: 1) Understand the types of features that should be exposed to the programmer for generating high-performance shared memory code, distributed memory code, and heterogeneous code; 2) Implement an initial set of abstractions for generating high-performance distributed and heterogeneous code, as well as enabling cooperative execution in these sytems. Getting the best performance out of a heterogeneous system requires the programmer to experiment with different computation mappings, data layouts, etc, all while utilizing as many of the resources as possible with cooperative execution. Without our approach, small changes, such as changing how computations are mapped to processing

elements, can result in a complete rewrite of code, possibly requiring multiple programming languages. Our work provides a practical and efficient way to harness the full computing power of these systems and reach the ultimate performance.

Our work provides the mechanisms for generating shared memory, distributed memory, heterogeneous, and cooperative code, with the ability to optimize architecture-specific features like communication. Our results show that we are able to generate efficient code for each of these architectures, without any modification to the original algorithm.

The abstractions introduced in our work mask the low-level details required to implement distributed and heterogeneous code. We provide a unified interface that generates MPI code for distributed execution and communication and/or CUDA code for GPU execution and communication. The programmer writes the same high-level primitives regardless of their backend, and then applies a set of scheduling commands to optimize and transform the resulting loop nests. These primitives and associated functions provide the ability to generate code specific to each backend, such as asynchronous execution and communication in MPI and CUDA. The schedules for generating CUDA and MPI can also be composed with the existing schedules in TIRAMISU, giving additional shared memory optimizations such as parallelization and vectorization. At code-generation time, the programmer selects their backend and our compiler handles applying the transformations and generating the appropriate backend code.

Overall, this thesis provides insight into the mechanisms required for heterogeneous, cooperative code execution and the challenges faced in optimizing code for these configurations. Future research can build off of this work and further explore how to create either new programming languages or models that provide intrinsic support for shared memory systems, distributed memory systems, heterogeneous systems, cooperative execution, and any other architectures or configurations that may come about.

Bibliography

[1] *Achieved Occupancy*. URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

[2] InfiniBand Trade Association et al. "The InfiniBand Architecture Specification". In: *http://www. infinibandta. org/specs/* (2000).

[3] Riyadh Baghdadi et al. "Pencil: A platform-neutral compute intermediate language for accelerator programming". In: *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE. 2015, pp. 138–149.

[4] Michael Bauer et al. "Legion: Expressing locality and independence with logical regions". In: *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press. 2012, p. 66.

[5] Mohamed-Walid Benabderrahmane et al. "The Polyhedral Model Is More Widely Applicable Than You Think." In: *CC* 6011 (2010), pp. 283–303.

[6] Robert D Blumofe et al. "Cilk: An efficient multithreaded runtime system". In: *Journal of parallel and distributed computing* 37.1 (1996), pp. 55–69.

[7] Bill Carlson et al. *Programming in the Partitioned Global Address Space Model*. 2003. URL: http://upc.gwu.edu/tutorials/tutorials_sc2003.pdf.

[8] George Chrysos. "Intel® Xeon Phi™ coprocessor-the architecture". In: *Intel Whitepaper* 176 (2014).

[9] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[10] Emanuele Del Sozzo et al. "A Common Backend for Hardware Acceleration on FPGA". In: *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE. 2017, pp. 427–430.

[11] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. "Distributed halide". In: *ACM SIGPLAN Notices*. Vol. 51. 8. ACM. 2016, p. 5.

[12] Joel Emer. *Graphics Processing Units (GPUs)*. 2017. URL: `http://csg.csail.mit.edu/6.823/lectures/L22.pdf`.

[13] CAPS Enterprise. "Cray Inc". In: *Nvidia, and the Portland Group,"The OpenACC application programming interface, v1. 0* (2011).

[14] Edgar Gabriel et al. "Open MPI: Goals, concept, and design of a next generation MPI implementation". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2004, pp. 97–104.

[15] Tarek El-Ghazawi and Lauren Smith. "UPC: unified parallel C". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 27.

[16] Ivan Grasso et al. "LibWater: heterogeneous distributed computing made easy". In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM. 2013, pp. 161–172.

[17] Mark Harris. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. 2013. URL: `https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/`.

[18] Mark Harris. *How to Optimize Data Transfers in CUDA C/C++*. 2012. URL: `https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/`.

[19] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[20] *Intel Omni-Path Fabric Edge Switches 100 Series*. Intel. 2016. URL: https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-edge-switch-100-series-brief.html.

[21] Philipp Kegel, Michel Steuwer, and Sergei Gorlatch. "dOpenCL: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE. 2012, pp. 174–186.

[22] Paul Krzyzanowski. *Distributed Systems*. 2017. URL: https://www.cs.rutgers.edu/~pxk/417/notes/01-intro-ds.html.

[23] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2009, pp. 45–55.

[24] Frank D Luna. *Introduction to 3D game programming with DirectX 10*. Jones & Bartlett Publishers, 2008.

[25] Robert M Metcalfe and David R Boggs. "Ethernet: Distributed packet switching for local computer networks". In: *Communications of the ACM* 19.7 (1976), pp. 395–404.

[26] Carla C Morris and Robert M Stark. *Finite Mathematics: Models and Applications*. John Wiley & Sons, 2015.

[27] David Mulnix. *Intel Xeon Processor Scalable Family Technical Overview*. 2017. URL: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm (visited on 12/26/2017).

[28] ARM NEON. *VFP intel (SSE4) programming reference*.

[29] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[30] John Nickolls et al. "Scalable parallel programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53.

[31] *numactl(8)-Linux man page*. 2004.

[32] Robert W Numrich and John Reid. "Co-Array Fortran for parallel programming". In: *ACM Sigplan Fortran Forum*. Vol. 17. 2. ACM. 1998, pp. 1–31.

[33] *NVIDIA CUDA Driver API*. 2017. URL: `http://docs.nvidia.com/cuda/cuda-driver-api/index.html`.

[34] *NVIDIA Tesla P100 The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU*. Tech. rep. WP-08019-001_v01.2. NVIDIA, 2017, p. 45. URL: `http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper-v1.2.pdf`.

[35] CUDA NVIDIA. "Compiler driver nvcc". In: *Options for Steering GPU Code Generation URL* (2013).

[36] CUDA Nvidia. *Programming guide*. 2010.

[37] David Ott. *Optimizing Applications for NUMA*. 2011. URL: `https://software.intel.com/en-us/articles/optimizing-applications-for-numa`.

[38] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011.

[39] Dhabaleswar K Panda et al. "The mvapich project: Evolution and sustainability of an open source production quality mpi library for hpc". In: *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Int'l Conference on Supercomputing (WSSPE)*. 2013.

[40] NVIDIA Compute PTX. "Parallel thread execution". In: *NVIDIA Corp., Jun* (2008).

[41] Jonathan Ragan-Kelley et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 519–530.

[42] James Reinders. "AVX-512 instructions". In: *Intel Corporation* (2013).

[43] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[44] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced computer architecture and parallel processing*. Vol. 42. John Wiley & Sons, 2005.

[45] Christopher J Rossbach et al. "Dandelion: a compiler and runtime for heterogeneous systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 49–68.

[46] Daniel Sanchez. *Cache Coherence*. 2017. URL: `http://csg.csail.mit.edu/6.823/lectures/L14.pdf`.

[47] Daniel Sanchez. *Memory Consistency Models*. 2017. URL: `http://csg.csail.mit.edu/6.823/lectures/L16.pdf`.

[48] Daniel Sanchez. *Memory Management: From Absolute Addresses to Demand Paging*. 2017. URL: `http://csg.csail.mit.edu/6.823/lectures/L04.pdf`.

[49] *Skylake - Microarchitectures - Intel - WikiChip*. URL: `https://en.wikichip.org/wiki/intel/microarchitectures/skylake` (visited on 12/26/2017).

[50] Avinash Sodani et al. "Knights landing: Second-generation intel xeon phi product". In: *Ieee micro* 36.2 (2016), pp. 34–46.

[51] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), pp. 66–73.

[52] Ken Strandberg. *Inside View: Tokyo's Tech's Massive TSUBAME 3 Supercomputer*. 2017. URL: `https://www.nextplatform.com/2017/08/22/inside-view-tokyo-techs-massive-tsubame-3-supercomputer/`.

[53] "Tiramisu: A Three-Layered Abstraction for Hiding Hardware Complexity from DSL Compilers". In: *PLDI 2018* (Under review).

[54] Sven Verdoolaege. "barvinok: User guide". In: *Version 0.23, Electronically available at http://www. kotnet. org/~ skimo/barvinok* (2007).

[55] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model." In: *ICMS*. Vol. 6327. Springer. 2010, pp. 299–302.

[56] Hao Wang et al. “Memory Scheduling Towards High-throughput Cooperative Heterogeneous Computing”. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT ’14. Edmonton, AB, Canada: ACM, 2014, pp. 331–342. ISBN: 978-1-4503-2809-8. DOI: 10.1145/2628071.2628096. URL: http://doi.acm.org/10.1145/2628071.2628096.

[57] Josef Weinbub. “Frameworks for Micro- and Nanoelectronics Device Simulation”. dissertation. Technical University at Vienna, 2014. URL: http://www.iue.tuwien.ac.at/phd/weinbub/.

[58] Mason Woo et al. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[59] Kathy Yelick et al. “Titanium: A high-performance Java dialect”. In: *Concurrency Practice and Experience* 10.11-13 (1998), pp. 825–836.

[60] Mohamed Zahran. “Heterogeneous Computing: Here to Stay”. In: *Queue* 14.6 (2016), p. 40.