```
if (sch == 1 || sch == 2 || sch == 3) {
    // Add accelerator interface (no buffers).
    hw_output
       .tile(x, y, xo, yo, xi, yi, tileSize, tileSize)
       .hw_accelerate(xi, xo);
    padded16.stream_to_accelerator();
}

if (sch == 2 || sch == 3) {
   // Add three buffers in the middle of the application design.
   lgxx.compute_at(hw_output, xo);
   lgyy.compute_at(hw_output, xo);
   lgxy.compute_at(hw_output, xo);
}

if (sch == 3) {
   // Add all of the buffers to the application design
   grad_x.compute_at(hw_output, xo);
   grad_y.compute_at(hw_output, xo);
   lxx.compute_at(hw_output, xo);
   lyy.compute_at(hw_output, xo);
   lxy.compute_at(hw_output, xo);
   cim.compute_at(hw_output, xo);
   cim_output.compute_at(hw_output, xo);
}
```

Code 6.1: Halide schedule variations for Harris. The `sch` variable chooses which blocks of scheduling code are applied to each algorithm. Each schedule successively adds more buffering to reduce needed recomputation.

6.4 Halide Scheduling Strategies for CGRAs

In the previous chapters, we saw how Halide scheduling was extended so that an application can be mapped to hardware accelerators. However, these scheduling primitives require the user to appropriately place these scheduling primitives on their application. Additionally, each scheduling primitives takes arguments (such as a tile size or unroll quantity). Below is guidance on where I have found these scheduling primitives most useful, and how to tune different arguments to maximize common performance metrics, such as application execution time.

Buffering Intermediate Values

We see that generally for image processing pipelines and machine learning pipelines that values should be buffered to avoid recomputation. Image processing line buffers are very efficient and there are plenty of memory tiles on the CGRA, so a line buffer should be placed between every pair of compute kernels that could use one. Code 6.1 shows a sequence of three schedules where buffers are successively addedd to the Harris corner algorithm. Table 6.1 shows the number of PEs, MEMs, and runtime with these three schedules. Notice how many PEs are saved by simply adding in some buffers. These intermediate buffers dramatically reduce the number of PEs with negligible increases

Table 6.1: Compiler results for Harris application with different Halide schedules. Each subsequent schedule adds an additional memory using `store_at().compute_at()` as shown in Code 6.1.

Harris Schedule	# PEs	# MEMs	Runtime (cycles)
sch1: recompute all	769	3	4097
sch2: recompute some	145	5	4103
sch3: no recompute	83	5	4146

in memory tiles and runtime.

Note that every computation kernel can be buffered, even element-wise operations. For compute kernels that do not need any buffering, the memory mapping analysis will recognize that no memory is needed and create a "memory" with no capacity, meaning a wire is created instead. This optimization occurs during memory mapping in Clockwork during dependency analysis. Due to this optimization, the user should not be worried about creating too many buffers, since any unneeded buffers are optimized away. The only drawback is slightly more compile time for Clockwork analysis.

Buffering dramatically reduces the number of PEs needed, and buffering elementwise operators is optimized to wires. Due to these observations, I recommend to:

> **Recommendation 1:** Buffer after every compute kernel, even if you are unsure if it is necessary. This will prevent expensive recomputation, and any unnecessary buffering will be optimized away.

Tiling due to Memory Constraints

Once buffers are placed, we have the problem of how to fit the intermediates in memory. Our memory tiles on the Amber CGRA are limited to a capacity of 2048 kB. We can tile the output image, which in turn tiles all intermediates and inputs in Halide. These tiles help each unified buffer fit within an SRAM. However, how large should tiles be? We can tile computation with many small tiles or a few large tiles.

Figure 6.4 shows how compute occupancy changes as we increase the size of a tile for two applications. Compute occupancy measures how often the hardware accelerator is outputting useful pixels. We find after scaling that larger tiles have better compute occupancy. Compute occupancy drops when computation does not produce useful output data. For image processing, the end of each input tile leads to several cycles of invalid outputs as the stencil computation wraps from one input line to the next. The fewer times the image wraps, the better the compute occupancy. Larger input tiles have a smaller percentage of the image classified as an edge. Thus, we prefer larger tiles.

One drawback of larger tiles are their longer compile time. Table 6.2 shows the compile times for gaussian and gemm. Compile time is longer since we must execute a full tile during Clockwork

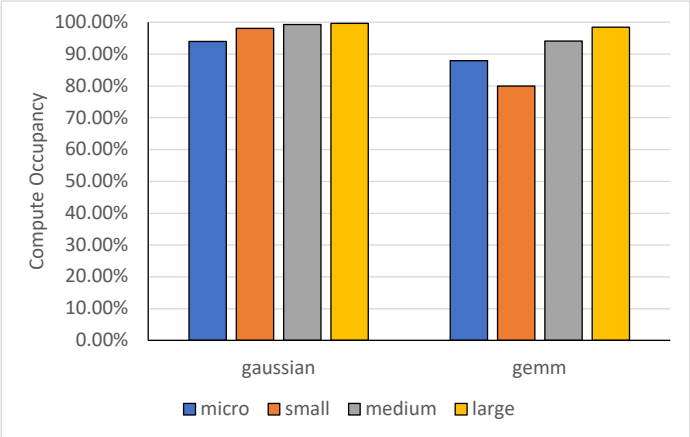


Figure 6.4: Scaling based on an increased image tile size. Compute occupancy is greatest for large input tiles for both gaussian blur and gemm.

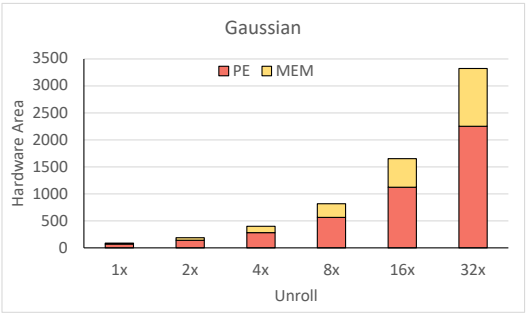
Table 6.2: Compilation times for different tile sizes. Compile time increases for larger tiles.

application	tile size	total compile time	app compilation	app generation	clockwork sched
gaussian	micro (64 × 64)	33	2.58	18.5	12.2
	small (240 × 180)	52	1.74	18.7	32.0
	medium (640 × 480)	183	2.55	18.3	162.5
	large (1920 × 1080)	1040	2.53	18.4	1018.9
gemm	micro (32 × 32)	33	2.58	17.0	13.8
	small (128 × 128)	38	2.39	16.9	18.7
	medium (512 × 512)	373	2.39	17.3	353.3
	large (2048 × 2048)	22386	2.42	18.1	22365.8

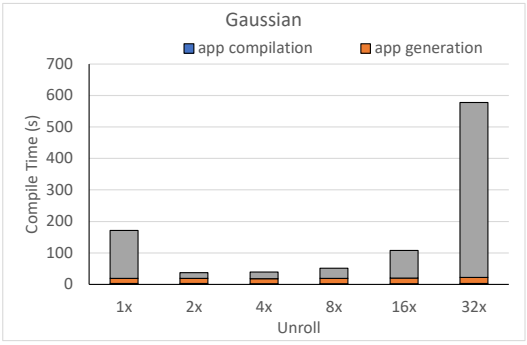
scheduling. However, generally we accept the longer compile time and prefer a solution with the greatest compute occupancy. Therefore, I recommend to:

Recommendation 2: Tile algorithms with tiles as large as possible. Large tiles have a greater compute occupancy, leading to a shorter total runtime for the entire application.

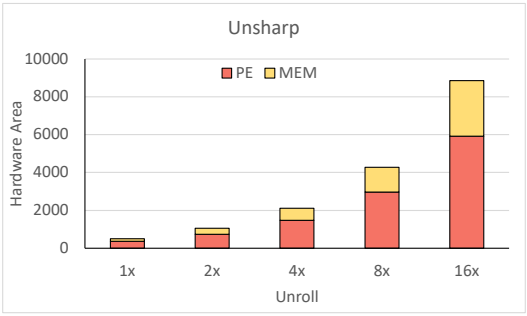
Code 6.2 shows how tiling is used on an application. Notice that the sizing creates a unified buffer that fits within a single 2048 kB memory tile. Furthermore, the output is tiled such that an integer number of tiles covers the entire output image. This ensures that each full run of the accelerator produces useful output values rather than computing on partial images.



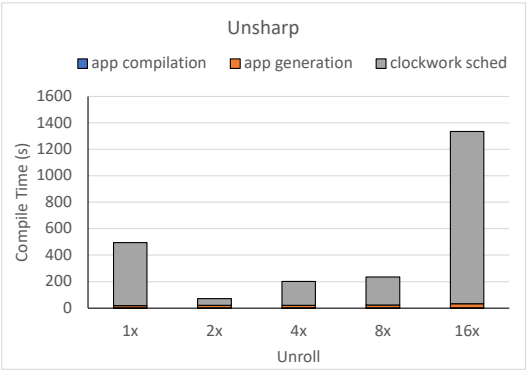
(a) gaussian hardware area



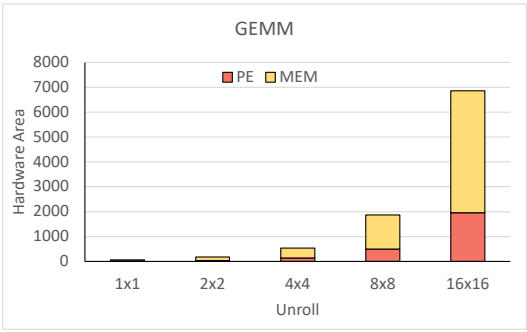
(b) gaussian compilation time



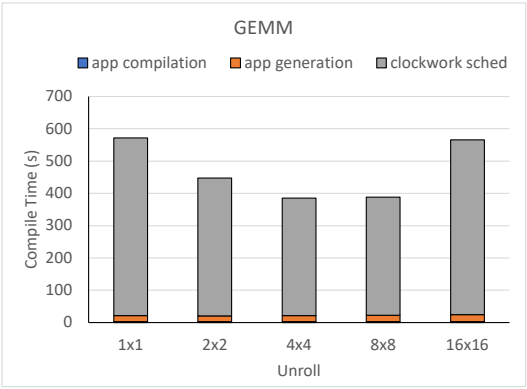
(c) unsharp hardware area



(d) unsharp compilation time



(e) gemm hardware area



(f) gemm compilation time

Figure 6.5: Hardware area and compilation time for gaussian, unsharp, and gemm as they unrolled. Gaussian and unsharp use input images sized 640×480 while gemm compute on 512×512 matrices. Unrolling applications increases the hardware area to implement them on the CGRA. Additionally, the compile time increases for larger hardware designs, mainly for Clockwork scheduling.

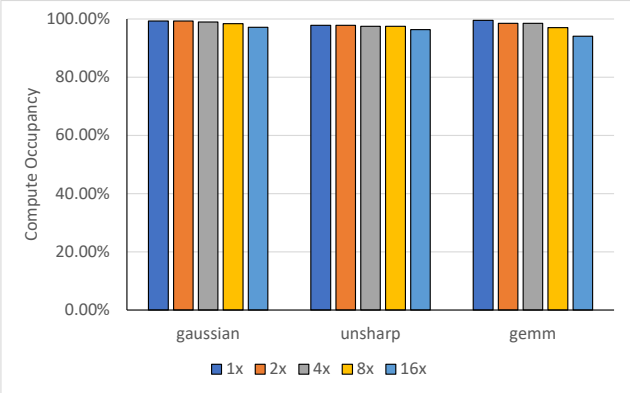
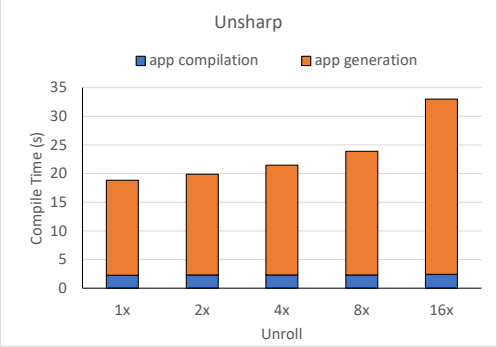
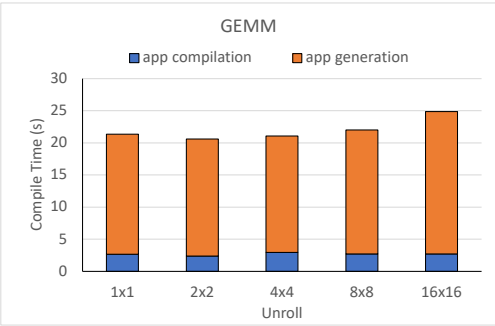


Figure 6.6: Compute occupancy with increasing unroll factor. A higher unroll factor has a lower compute occupancy for every application.



(a) unsharp frontend compilation time



(b) gemm frontend compilation time

Figure 6.7: Compilation time for the first two steps of the compiler. Modest increases in compile time as each application is unrolled more.

```
// Input image size: 6000 x 4000
// Output tile size: 600 x 400
// Number of executions for full image: 10x10

int tileWidth = 600;
int tileHeight = 400;
hw_output_mem
  .tile(x, y, xo, yo, xi, yi, tileWidth, tileHeight)
  .reorder(z, xi, yi, xo, yo);

int glbWidth = tileWidth;    // GLB size same as tile
int glbHeight = tileHeight;
hw_output_glb
  .tile(x, y, xo, yo, xi, yi, glbWidth, glbHeight)
  .reorder(z, xi, yi, xo, yo);
```

Code 6.2: How to tile memories to fit on the CGRA. This sample code tiles the upsample application with $10 \times 10 = 100$ iterations to create a 6000×4000 output image. Due to line buffering, only a few lines are needed for the memory tiles; but the GLB holds the full tile for an accelerator execution.

Unrolling to Duplicate Hardware

Besides buffering memory, another essential part of a hardware accelerator is using the available compute resources. For our Halide applications, we use the unroll scheduling directive to perform hardware duplication. Unrolling a loop increases the number of compute resources used in the loop body which in turn decreases the overall runtime, but it also affects other performance metrics.

Figure 6.5 shows how unrolling compute affects the total hardware area as well as total compilation time. Each application uses a fixed input size, and then is scheduled with a varying unroll factor shared across every compute kernel. The total hardware area converts the utilized hardware tiles into their respective area size. As expected, there is a linear increase in PEs and MEMs used as we increase the unroll factor. By unrolling, we create a design with better spatial utilization and faster execution time.

However, unrolling the application also slightly decreases temporal compute occupancy and increases compile time. Figure 6.6 shows the decrease in compute occupancy with increasing unroll factor. Unrolling a loop effectively decreases the input tile size that each compute unit sees. Based on the findings on image size in the last subsection, we expect and observe a decreased compute occupancy with these smaller tile sizes.

Figure 6.5 shows the total compilation time for each application with increasing unroll factor. Note that Clockwork scheduling includes a required execution of the accelerator. We find that compilation time at first decreases and then increases. This is because Clockwork scheduling has an increased runtime for bothh longer loop iteration lengths as well as more hardware. When the algorithm is unrolled with a small factor of 1, the iteration time is large. When the unroll factor is a large factor of 16 or 32, the large amount of duplicated hardware causes long Clockwork scheduling times. In the middle with $2\times$ or $4\times$ unroll factors, we see a sweet spot with lower compile times.

The compile times in Figure 6.5 are dominated by Clockwork scheduling and it is difficult to see the compilation trends for the first two stages. Therefore, Figure 6.7 shows the compilation time for just the first two frontend compilation steps. The frontend compilation times have a modest increase with unroll factor, but not as drastic as Clockwork scheduling. The frontend compilation in Halide is not as affected by large designs as Clockwork scheduling is.

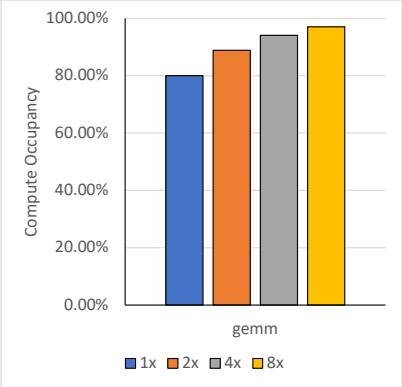


Figure 6.8: Increasing IO unroll factor on gemm using a compute unroll of 8×8 on 512×512 matrices. Greater IO unroll factor helps keep the compute kernels busy.

Although we have focused mainly on unrolling compute, it is also critical to unroll IO ports as well. Duplicating IO ports ensures that the compute and memory units are supplied with enough data so that they do not idle. Unrolling IO supports the compute rate for image processing pipelines, while unrolling IO for DNNs reduces the initialization and read-out phases. Figure 6.8 shows how compute occupancy of gemm increases as IO is unrolled with a greater factor. The target design has 8 input and 8 output channels unrolled for the compute kernel. The IO ports are then unrolled from 1 to 8. Unrolling the IO up to the rate of the compute kernel gives steady increases in the compute occupancy.

We observe that unrolling as much as the reconfigurable fabric can handle is best. Decreasing execution time using the available compute units is the purpose of having a large compute array on a hardware accelerator. Therefore, I recommend to:

Recommendation 3: Unroll compute kernels to fill the CGRA compute fabric as much as possible. Additionally, unroll IO to match the throughput of the compute kernels.

Code 6.3 shows how we can unroll a DNN layer to create multiple MACs and use multiple IOs.

```
// Example tiling in conv3_1 layer of ResNet
// Unroll compute convolutions
output_cgra.update()
  .unroll(r.x).unroll(r.y);  // Unroll the convolution reduction fully

// Additionally, unroll input and output channels for compute
int k_oc = 8;
int k_oc = 8;
output_cgra
  .unroll(w, k_oc);  // Unroll the output channels partially
output_cgra.update()
  .unroll(w, k_oc)
  .unroll(rz_unroll, k_ic);  // Unroll the input channel reduction partially

// Unroll IO streams to match rates
int glb_o = 1;  // No duplication for output stream for this particular layer
int glb_i = 4;
hw_output.unroll(w, glb_o);
output_glb.unroll(w_cgra, glb_o);
input_glb.unroll(z, glb_i);
input_cgra.unroll(z_cgra, glb_i);
```

Code 6.3: Different use cases of unroll: duplicating convolution reductions, duplicating compute across channels, and matching rates for IOs.

Sharing Compute Kernels to Save Resources

The last chapter, Chapter 5, described how compute sharing can be added to an Halide application's schedule.

We observe that compute sharing can be applied to pyramid applications and matching compute kernels to reduce necessary compute resources for large applications. When applications additionally have downsampling and reduced temporal usage of their compute kernel, this can be performed with minimal reduction in execution time.

> **Recommendation 4:** Use compute sharing to increase utilization of compute elements on the hardware accelerator. The reduced resources are freed up with minimal increases in execution runtime. Computing sharing is most easily applied and has the most impact on applications with pyramid structures.

Buffering to Create Memory Hierarchies

The Halide scheduling we have described so far are helpful strategies for making the most of the target accelerator. However, additionally a hardware accelerator has a fixed memory hierarchy that must be followed. Following Halide's philosophy, we do not need to describe this target-specific hierarchy in the algorithm. Instead, we can use scheduling to alter the algorithm to accommodate

the CGRA's memory hierarchy.

The CGRA's memory hierarchy consists of a host DRAM, global buffer, memory tiles, as well as small register files on each PE. Memory transfers between these entities should be depicted in our IR so that they can be mapped to our CGRA interconnect. Halide's `in()` schedule is used to represent these transfers in the memory hierarchy. Each successive call to `in()` on a buffer refers to another copy of the same value. This scheduling primitive simply creates a statement that copies from one buffer to another with the understanding that each buffer copy will be mapped to its own memory in the hierarchy. For example,

```
output.in()
```

refers to the output values on the global buffer while `output` refers to the output values on memory tiles.

Besides creating memory copies, it is also important to label buffered intermediates as memory elements on which part of the hierarchy. This is solved by using the parameter of `store_at()`. The `store_at()` scheduling primitive dictates that a memory primitive should be created while the output memory (specified using the correct number of `in()`s) determines which level in the hierarchy that it should be placed. For example,

```
conv1.store_at(output.in(), xo)
```

specifies that `conv1` should be stored at the memory tile level.

The final element of constructing the hierarchy is creating the hardware accelerator interface. We use

```
input.stream_to_accelerator()
```

on the input memory tile buffer to specify that it is an input that comes in through the global buffer. Similarly,

```
output.in().hw_accelerate(xi, xo)
```

specifies that the output global buffer is the output of the accelerator. We observe that our accelerator interface and Halide's `in()` calls can represent an accelerator's memory hierarchy well.

Recommendation 5: Utilize Halide scheduling to construct a memory hierarchy and accelerator interface to match the target hardware accelerator.

Altogether, Code 6.4 shows how we schedule a single call to create a memory hierarchy using all of the above calls.

```
// Output tiling and output stream
hw_output.in()        // host level
  .compute_root()
  .tile(x, y, xo, yo, xi, yi, 360, 360)    // GLB level sized 360x360
  .hw_accelerate(xi, xo);
hw_output             // GLB level
  .store_in(MemoryType::GLB)
  .tile(x, y, xio, yio, xii, yii, 60, 60)  // MEM level sized 60x60
  .compute_at(hw_output.in(), xo);

// Intermediate buffer at MEM level
conv2
  .store_at(hw_output, xio)
  .compute_at(hw_output, xio);

// Input stream
hw_input.in().in()    // MEM level
  .compute_at(hw_output, xio);
hw_input.in()         // GLB level
  .compute_at(hw_output.in(), xo)
  .store_in(MemoryType::GLB);
hw_input              // host level
  .compute_root()
  .accelerator_input();
```

Code 6.4: Memory hierarchy that creates the output stream, intermediate buffer at the MEM level, and an input stream.

Using Generator Parameters for Flexible Designs

Many of the applications we construct can be created with slight adjustments from previous applications. For example, ResNet has many layers, but many of these layers share the same structure. The differences are the number of input channels, output channels, and kernel size. Instead of creating many Halide applications, one for each layer, we can create a ResNet layer generator. This generator has a generator argument for each variable that changes from layer to layer. Then, we can call the application with different generator parameters for each distinct layer. This same technique can be used for image processing applications where the kernel size of a convolution could be changed. One such example in our application suite is a chain of convolution kernels where the convolution kernel size as well as number of successive convolution kernels are user-specified arguments.

Code 6.5 shows how these generator parameters are created in Halide, and then called for each an example layer in ResNet.

Recommendation 6: Construct extensible applications using generator arguments to reuse your code and create multiple DNN layers or convolutions from a single application.

```
// make clockwork HALIDE_GEN_ARGS="in_img=14  pad=1 ksize=3 stride=1 n_ic=256 n_oc=256"

// in_img determines the input image size
GeneratorParam<int> in_img{"in_img", 56};    // default: 56
// pad determines the padding to the input image size
GeneratorParam<int> pad{"pad", 1};           // default: 1
// ksize determines the output stencil size
GeneratorParam<uint8_t> ksize{"ksize", 3};   // default: 3
// Stride determines the sampling rate for the down sample
GeneratorParam<int>  stride{"stride", 1};    // default: 1
// n_ic determines the total number of input channels
GeneratorParam<int> n_ic{"n_ic", 32};        // default: 32
// n_oc determines the total number of output channels
GeneratorParam<int> n_oc{"n_oc", 32};        // default: 32

// Selected usage of GeneratorParams in the application:
// Defining tile size based on input and computation size
int tilesize = floor( (in_img + 2*pad - ksize) / stride ) + 1;
output_glb
  .tile(x, y, x_glb,y_glb, x_cgra,y_cgra, tilesize,tilesize_y, TailStrategy::RoundUp)

// Bounding input and output channels
kernel_glb.bound(w, 0, n_oc);
input_glb.bound(z, 0, n_ic);
```

Code 6.5: Usage of generator parameters in Halide application to create extensible layers.

Auto-scheduling Hardware Accelerators

With these observations on how to schedule image processing and machine learning applications, we have found a mechanical process in scheduling applications. New applications in our domains generally follow similar structures to other examples that we have seen. Since this guidance works with many applications, we could also look to auto-schedule these applications. An auto-scheduler and auto-tuner would take in the hardware constraints (such as number of compute tiles, number of IOs, capacities in the memory hierarchy, and number of levels in the memory hierarchy). Then, an input applications would be tiled and unrolled to fit the target accelerator size. The application would try to minimize the execution time under the constraining size of the target accelerator.

This auto-scheduling problem has a smaller search space by focusing on the guidance of the above observations. Furthermore, a user would be able to refine and improve the suggested schedule after it has been auto-scheduled.

6.5 System Evaluation

We have seen how Halide applications are compiled to CGRA implementations, and then mapped to CGRA bitstreams. This section goes through an evaluation of the compiler usability and productivity. Then, we look at an overall evaluation of the hardware implementations for our suite of image processing and machine learning applications.