Westfälische
Wilhelms-Universität
Münster

Auto-Tuning of Stencil Applications running on Modern Many-Core Systems

Projektseminararbeit

vorgelegt von:

Martin Wrodarczyk, Richard Schulze, Bernd Eissing, Christian Esch, Luis Wetzel, Alexander Holthaus, Philipp Hugenroth

Matrikelnummern: , 396610, , , , , 425967

Studiengang: MSc. Informatik

Thema gestellt von:
Prof. Dr. Sergei Gorlatch

Arbeit betreut durch:
Dipl.-Inf. Ari Rasch

Münster, October 11, 2016

Abstract

Phil:

This paper presents an utilization and improvement of the *OpenTuner* framework for the purpose of implementing performance-portable, stencil-type kernels written in CUDA, OpenCL and Halide. This is done by introducing hardware-dependent launch parameters and then finding the best values for these using auto-tuning, achieving speedups up to 50% when compared to the same kernels without tuning. We also examine the runtime effects of out-of-bounds checking and different shared memory configurations.

Keywords: GPGPU, auto-tuning, stencils, parallel programming

Contents

1 Introduction

(**Phil**): With the physical limitations of processor speeds fast approaching, means of speeding up applications have changed: no longer is this done by increasing processing power alone, but instead by dividing problems into subproblems, which can then be calculated in parallel. *General-purpose computing on graphics processing units* (GPGPU) in particular has been popular in the last few years.

However, achieving optimal execution times depends heavily on the specifications of the hardware used, and porting applications to different hardware is often error-prone and time-consuming. Thus, in this paper, we examine how to use dynamic parameters in combination with an auto-tuning application, which can find the optimal parameters for the current hardware by performing a strategic search on the possible parameter space. This approach achieves optimal performance on CPUs and GPUs without having to rewrite the program.

This paper is divided into two main parts:

The first part deals with auto-tuning in general. Existing solutions are being evaluated and adapted to our specific needs to efficiently and conveniently tune the developed application.

The second part explains the concept behind so-called *stencil applications*, deals with the implementation of performance-portable applications using tuneable parameters and shows the effects of auto-tuning on two test applications running on CPUs/GPUs.

2 OpenCL, CUDA and Halide

2.1 OpenCL

(**Phil:**)

OpenCL is a vendor- and device-independent open-source framework for writing parallel programs. OpenCL programs, called *kernels*, are programmed using a modified version of the C language. Each kernel is started by a *host program*, which also controls how data is distributed among the compute units of the used hardware. This is done by setting the number of "*work groups*", each of which contains a number of threads ("*work items*"). The number of work groups is set by passing an array containing the amount of work items in the work groups in the x- and y-dimensions to the kernel upon execution, called *localSize*. The total number of work items is set in a similiar way, by passing the *globalSize* array to the kernel.

Each work item executes the same kernel, but works on different data based on its absolute ID ("*global ID*") or ID relative to its work group ("*local ID*").

OpenCL abstracts the physical memory of the device by introducing a memory hierarchy which is mapped to its hardware equivalent transparently:

- Global memory
- Local memory
- Private memory

Global memory can be accessed by all work items and can be used for inter-workgroup communication. On GPUs, this region is mapped to the GPU's main memory, while on CPUs the RAM is used.

Local memory is local to each work group, meaning only work items that are in the same work group can share data through local memory. On GPUs, local memory is mapped to the fast "*shared memory*" segment of the GPUs hardware, and thus accessing it is much faster than accessing global memory. On CPUs however, local memory is instead mapped to the RAM as well.

Lastly, *private memory* provides the fastest memory access, but is restricted to each work item and as such cannot be read or written by any other work item.

2.2 CUDA

(**Phil:**)

CUDA is NVIDIA's proprietary framework for parallel programs. CUDA kernels are restricted to running on NVIDIA GPUs, but are of essentially the same structure as equivalent OpenCL implementations with the exception of different terminology ("*thread*" instead of "*work item*" and "*thread block*" instead of "*work group*") and different API calls.

However, newer CUDA implementations provide control over how the shared memory is divided into user-managed local memory and automatically managed L1 cache segments. Depending on preference, the shared memory can allow more memory for one of the segments, shrinking the other in return.

2.3 SkelCL

(**Chris:**)

SkelCL is a software library for providing high-level abstractions which greatly simplify programming of modern heterogeneous systems consisting of multiple multi-core CPUs and/or GPUs. It is currently being developed by Michel Steuwer and other members of the research group *parallel and distributed systems* at the University of Münster, Germany.

Built on top of the OpenCL standard, which has been briefly described in 2.1, SkelCL provides means for high-level programming of heterogeneous systems without the complexity of low-level approaches like OpenCL or CUDA. Procedures like manual memory management, handling of multiple compute devices, explicit data movement between devices and other hardware specific functions have proven to be error-prone and repetitive when implemented directly in low-level environments. As such SkelCL offers pre-implemented recurring computation and communication patterns for simplifying and reducing the workload of programmers. SkelCL calls these patterns *Skeletons* and additionally provides an abstract vector data type responsible for implicit data transfers to and from devices. Primarily SkelCL has been developed with GPUs in mind, but since it is based on OpenCL it is not bound to any specific hardware and can be executed on any OpenCL-capable device.

2.3.1 Using SkelCL

In order to use SkelCL it is necessary to always include the standard `SkelCL.h` header and any other header corresponding to the pattern that will be used. Further, before the first call to any Skeleton, SkelCL needs to be initialized. While doing so, it is also possible to pass parameters that define traits for selecting the OpenCL devices, for example that any OpenCL device used for

Skeleton execution is a GPU or that the calculation should be distributed over four devices, and so forth.

As the next step, initializing objects of the provided *Skeletons* is necessary. Currently available *Skeletons* are `Allpairs`, `Map`, `MapOverlap`, `Reduce`, `Scan`, `Stencil` or `Zip`. For our task only `MapOverlap` and `Stencil` are of relevance.

Listing 2.1: Skeleton Constructor.

```
1 MapOverlap<Tout(Tin)>(const Source& source, unsigned int
      overlap_range, Padding padding, Tin neutral_element, const
      std::string& func);
2
3 Stencil(const Source& source, const std::string& func, const
      StencilShape& shape, Padding padding, Tin neutral_element);
```

A MapOverlap Skeleten is used by first constructing it with the constructor shown in line 1 in 2.1. The first parameter is a string containing a user function written in OpenCL-C, that will be executed for each workitem. This function must be callable by another `__kernel` function and may also contain additional subroutines. In case the function name to be called is not `func` or there are multiple function defined in the first parameter, the fifth parameter defines the name of the function to call. The second parameter informs the Skeleton of the range of the stencil, so it may load additional values at runtime to ensure correct behavior. The third parameter defines the padding type for out of bound access. In case the selected padding type needs a user defined value, such as for neutral value padding, the fourth parameter can be used to pass this.

Very similar to the MapOverlap Skeleton, the Stencil Skeleton takes the same parameters, even if in a different order, with the one exception being the range. Instead it takes a StencilShape, which is a set of two ranges for each dimension of the input data. By doing so stencil operations that do not take a line, square or cubic form with the origin in the center can be more effectively calculated.

Internally what happens by calling these constructors, is that SkelCL takes a pre-written kernel and embeds the user defined function into this kernel. This kernel does takes care of loading any data that might be needed into local memory before the actual execution of the kernel and ensures the correct padding behavior.

In order to execute the Skeleton first the data that will be passed to it, has to be enclosed in either one of SkelCLs `SkelCL::Vector` or `SkelCL::Matrix` data types. This enables SkelCL to manage data distribution and any copy operations that might be necessary for the user. Once this has been done, the overloaded parenthesis operator of the Skeleton class may be used with the data as the first argument and any additional parameters needed by the user function following right after it.

SkelCL completely takes over memory management and also avoids a lot of boiler plate code compared to directly using OpenCL, allowing for fast and easy implementation of OpenCL based programs.

2.4 Halide

"*Halide - a language for image processing and computational photography*" (TODO: cite halide homepage.)

Named after the silver halides used in the development for early photographic films and papers, the software library Halide is a programming language specifically designed for high performance image processing on modern machines. Developed by the Massachusetts Institute of Technology - Computer Science and Artificial Intelligence Laboratory (MIT CSAIL) in collaboration with Adobe, the concept Halide is based on, was first publicly presented at the SIGGRAPH 2012.

Currently many more people are actively developing Halide, like Intel and others, and the source code is publicly accessible on GitHub under a commercially permissive MIT license, allowing for academical as well as commercial use of the software.

Halide is calling itself a programming language, and there is indeed an intermediate Halide language representation, but currently the front end is completely embedded into C++. This alleviates the need of learning a specific new language for a narrow field of applications.

The base concept of Halide centers around the idea of separation of algorithm and schedule. Or more exactly of the separation of the definition of "What is being calculated?" from the definition "In what order is it being calculated?". In practice this is of interest, because in image processing it is often the case that while the general idea of what needs to be calculated never changes, like for example a complicated photo effect filter for color correction, but depending on the hardware the optimal execution order may vary greatly. And even more important it dramatically speeds up the process of finding effective and highly optimized execution orders during development, granting improved performance at reduced development time.

As a language itself Halide uses it's own compiler, which is in turn using LLVM as a backend to target a wide range of platforms including x86/SSE, ARM v7/NEON, CUDA, Native Client, and OpenCL.

2.4.1 Using Halide

(**Luis:**)

The main benefit of using Halide over directly utilising low-level APIs or even no API at all is that it allows for quick and easy iteration over different optimisation strategies. This is achieved by, on one hand, separating the definition of the calculation ("What do you want to compute?") from the optimisations used to calculate it ("Where and when do you want to compute it?") and on the other hand by taking a lot of work away from the user through automatic code generation. Halide internally applies the optimisations, from now on referred to as *schedule*, to the given calculation, from now on referred to as *algorithm*. Because of the automation the user does not have to make big changes in the code and spend a lot of time ensuring correctness of the algorithm whenever he makes changes to the schedule. This allows for quick testing of a variety of different schedules and finding the optimal one in less time. Halide provides additional features to simplify the quick change of different aspects of the program including different techniques of out-of-bounds handling, back-end APIs and devices. Since the code for the selected back-end, like OpenCL, CUDA or plain C, is automatically generated, you can quickly test a variety of schedules on the different back-ends without having to port any code.

We will now take a closer look at how Halide works and how to use it by examining a simple example of a 3×3 box blur program.

Defining the algorithm

Halide utilises C++ classes and operator overloading to create a language embedded into C++ which is used to define the algorithm for a program. The algorithm is a pure function, that is a function which has no side effects and where the same parameters always evaluate to the same output. It takes image coordinates of the output image as parameters and evaluates to the value of the output image at that given point. Listing 2.2 is a simple example of an algorithm. The Var class is used to define each of the image coordinate parameters, objects of the Func class describe the actual function. The overloaded operators of the Var class are used to build expressions based on the parameters and numeric literals in an intuitive way. During the execution of the calculations this function will be evaluated once for each pixel of the output image.

Listing 2.2: Example of a simple algorithm.

```
Halide::Var x, y;
Halide::Func gradient;
gradient(x, y) = x + y;
```

Listing 2.3 shows the 3×3 box blur example. The input can be either a function itself or an image which is a wrapper class template Halide provides for image data buffers. It uses two stages for its calculation: first a function blur_x is defined which takes the arithmetic mean of three input values in the x-direction, then a the function blur_y is defined which uses the results of blur_x and also averages them in the y-direction. Halide automatically recognizes the dependency between blur_x and blur_y and allows for different optimisations for how this dependency may be resolved, which are defined with the schedule.

Listing 2.3: 3x3 box-blur example.

```
Halide::Func blur_x, blur_y;
blur_x(x, y) = (input(x - 1, y)
    + input(x, y)
    + input(x + 1, y)) / 3;
blur_y(x, y) = (blur_x(x, y - 1)
    + blur_x(x, y)
    + blur_x(x, y + 1)) / 3;
```

Defining the schedule

As stated before the actual evaluation of a Func object is done by evaluating it once for each pixel in the output image and storing the output in that output image. By default Halide assumes that this is done by nesting as many loops as there are parameters for the function and letting them iterate over the functions domain. Listing 2.4 shows a pseudocode representation of the default schedule. The function is evaluated over a domain $(x, y) \in [0, OUTPUT\_WIDTH - 1]$×[0, OUTPUT_HEIGHT-1] where OUTPUT_WIDTH and OUTPUT_HEIGHT are the width and height of the output image.

Listing 2.4: Default loop nesting.

```
for(int y = 0; y < OUTPUT_HEIGHT; y++)
  for(int x = 0; x < OUTPUT_WIDTH; x++)
    output(x,y) = func(x,y);
```

Halide provides basic operations which rearrange the loop nesting of the default schedule to create other schedules. The most basic operation is reordering, which simply changes the order of a loop and another loop deeper into the nesting (Listing 2.5). Loops can be split into an outer loop and an inner loop using a split factor. The outer loop then iterates over blocks of indices whose size is given by the split factor, and the inner loop then iterates over the indices within each block (Listing 2.6, split factor 2). Finally two loops can be fused into one where the resulting loop iterates over an index range equal to the product of the two ranges of the original loops (Listing 2.7). These basic operations can be used to construct more complex oprations like tiling. The constructed loops can then be parallelized and vectorized to improve performance.

Listing 2.5: Reordering the default loops.

```
for(int x = 0; x < OUTPUT_WIDTH; x++)
  for(int y = 0; y < OUTPUT_HEIGHT; y++)
    output(x,y) = func(x,y);
```

Listing 2.6: Splitting the default loops.

```
for(int y=0; y < OUTPUT_HEIGHT; y++)
    for(int x_out=0; x_out < OUTPUT_WIDTH/2; x_out++)
        for(int x_in=0; x_in < 2; x_in++)
            output(x,y) = func(2 * x_out + x_in ,y);
```

Listing 2.7: Fusing the default loops.

```
for(int xy = 0; xy < OUTPUT_HEIGHT * OUTPUT_WIDTH; xy++)
  output(x,y) = func(xy % OUTPUT_WIDHT,
       xy / OUTPUT_WIDTH);
```

Listing 2.8 is a schedule for the 3×3 box blur example for which the algorithm was already shown. For blur_y the tile operation is used, which is composed of two splits, one in x- and one in y-direction and then a reorder. Because this is a common operation Halide provides a shorthand for it. This results in two outer loops which iterate over the tiles (x and y are reused for that in the example) and two inner loops which iterate over each value within the tile (x_item, y_item in the example). The tile operation is then used again creating smaller tiles within the tiles. The constants GPU_WG_SIZE_X, GPU_WG_SIZE_Y, GPU_ITEM_SIZE_X, GPU_ITEM_SIZE_Y specify over how many elements the inner loops should iterate in each tile. The gpu_blocks and gpu_threads are specialised operations used for GPU schedules, which specify that instead of iterating over the loops they should be calculated in parallel using GPU blocks (Work groups in OpenCL) for the x and y loops and GPU threads (Workitems in OpenCL) for the x_item and y_item loops. The x_elem and y_elem loops remain unparallelized and will be calculated as loops within each GPU thread. Halide provides similar operations for CPU schedules.
As seen in the definition of the algorithm for the 3×3 box blur the blur_y function is dependent on the blur_x function, we have to specify when this function is evaluated. One option is to use the compute_root function to evaluate all values required by blur_y once in the beginning of the calculation and storing them in a lookup buffer for blur_y. Another option is to evaluate blur_x whenever needed, effectively inlining blur_x into blur_y. This example uses an intermediate approach by calculating all values used in a single GPU block (same as one unique configuration of x and y) and storing them in a lookup buffer before evaluating that block. The calculation for blur_x for each block is again tiled and parallelized using GPU threads.

Listing 2.8: Schedule for the 3x3 box-blur.

```
Halide::Var x_item, y_item, x_elem, y_elem;
blur_y.tile(x, y, x_item, y_item, GPU_WG_SIZE_X, GPU_WG_SIZE_Y)
  .tile(x_item, y_item, x_elem, y_elem, GPU_ITEM_SIZE_X,
     GPU_ITEM_SIZE_Y)
  .gpu_blocks(x, y)
  .gpu_threads(x_item, y_item);
blur_x.compute_at(blur_y, Halide::Var::gpu_blocks())
  .tile(x, y, x_elem, y_elem, GPU_ITEM_SIZE_X, GPU_ITEM_SIZE_Y)
  .gpu_threads(x, y);
```

Compiling the pipeline

Listing 2.3 shows the algorithm of the 3×3 box blur example. One way to provide the input object for that snippet is to create a Func object which reads the input image with out-of-bounds handling. Halide provides shorthands for a few options like a neutral border element or clamping the coordinates to the edges of the input. For the 3×3 boxblur example we will use zero as a border element. Listing 2.9 shows how to generate the input used in the algorithm. First a C++ array of appropriate size is filled with the input data. The data is then passed to a Halide Buffer object which is then used as data source for a Halide float Image. The constant_exterior boundary condition creates the Func object which, if the coordinates are within the bounds of the images, evaluates to the value of the input image at the given coordinates and to zero otherwise.

Listing 2.9: Creating input with out-of-bounds check.

```
float *inputData =
  fillData(new float[INPUT_WIDTH * INPUT_HEIGHT]);
Buffer input_buffer = Buffer(Float(32),
      INPUT_WIDTH,
      INPUT_HEIGHT,
      0,
      0,
      (uint8_t*)inputData);
Image<float> input_image = Image<float>(input_buffer);
Func input = BoundaryConditions::constant_exterior(input_image,
     0);
```

Once the algorithm and schedule are defined and settings like the back-end or the device to use are set, Halide can compile the pipeline. Halide provides two methods of compilation, ahead-of-time (AOT) and just-in-time (JIT). Ahead-of-time compilation generates a static library and a header file, which can be included in another program and used for calculations there. Just-in-time compilation generates the code internally without outputting it and allows the user to run the calculations directly from the Halide application. Listing

2.10 demonstrates how to just-in-time compile and run the 3×3 box blur example using OpenCL as back-end. First a C++ Array for the output data is allocated and passed to a Halide buffer object. Then we get the Target object using get_host_target. The Target object manages a variety of settings used for compilation and execution. We set the back-end to OpenCL in the target. Finally, we use the compile_jit function to just-in-time compile blur_y and then execute using realize. The realize function takes the output buffer as argument and automatically uses the dimensions of the buffer as domain for the parameters of blur_y.

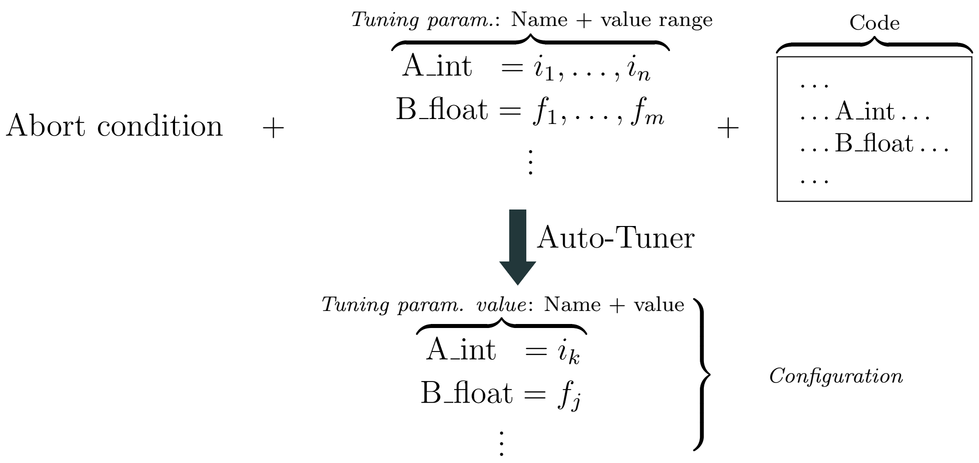
Listing 2.10: Compiling and executing the pipeline.

```
float *outputData =
  new float[OUTPUT_WIDTH * OUTPUT_HEIGHT];
Buffer output_buffer = Buffer(Float(32),
      OUTPUT_WIDTH,
      OUTPUT_HEIGHT,
      0,
      0,
      (uint8_t*)outputData);
Target target = get_host_target();
target.set_feature(Target::OpenCL);
blur_y.compile_jit(target);
blur_y.realize(output_buffer);
```

3 Automatic Tuning

(Ber:) *Automatic tuning* is an approach to automatically generate optimized code. This approach is based on two basic thoughts: Firstly, when writing an application you can use parameters to affect the performance. For example you can affect the run time of your application by starting two threads to divide the work done. Such parameters are called *tuning parameters* and can be optimized in regard to the hardware and input size of the application you wish to tune. Secondly, the tuning parameters are determined using an automated search algorithm. This algorithm is searching for different values that can be used as values for the tuning parameters.

If you wish to tune your application or code you need to set an abort condition (for example a tuning time) as well as define tuning parameters. The tuning time is how long the auto-tuner will run and the tuning parameters are defined in a special way so the auto-tuner can recognize them.



Tuning param.: Name + value range
Code
A_int $= i_1, \dots, i_n$
B_float $= f_1, \dots, f_m$
Abort condition +
+
...
...A_int...
...B_float...
...
Auto-Tuner
Tuning param. value: Name + value
A_int $= i_k$
B_float $= f_j$
Configuration

Figure 3.1: The concept of automatic tuning.

As you can see in the image (see Figure 3.1), a tuning parameter is a variable (`int`, `float`, etc.) grouped together with a value range. These tuning parameters have to be defined inside the code you wish to tune. Once you have prepared your application as described above the auto-tuner will start by searching for values for the tuning parameters. The names together with the values of the tuning parameters are called configuration. The auto-tuner will then replace the tuning parameters with the values and start the application which is supposed to be tuned. Once the application is done it has to report the result (run time, memory used, etc.), so the auto-tuner can use it to determine if the configuration found was good or bad. The result will then be written into a database so the auto-tuner can use this information to find a better configuration for the next tuning run. This process will repeat until the tuning time set is over, which is when the auto-tuner will report the best configuration together with the result for that configuration. Auto-tuner can also be configured to run a certain amount of configurations, as tuning time can be hard to determine because some applications run longer or shorter than others. There are more ways to control how long the auto-tuner will run; they just have to be configured the way best suited for the application that is going to be tuned.

3.1 Running Example: Matrix-Vector Multiplication

(Ber:) Throughout the project seminar we have been tuning an application called GEMV (Generalized matrix-vector multiplication). Because we started evaluating auto-tuner right at the beginning of the project seminar, we needed an application to tune. Since the stencil application was not yet done, GEMV was tuned for the automatic tuning chapter. This application expects a matrix M (which need not be a square matrix) and a vector V, which needs to be appropriate to the size of the matrix. The application will simply multiply the matrix with the vector and the result is the output vector R.

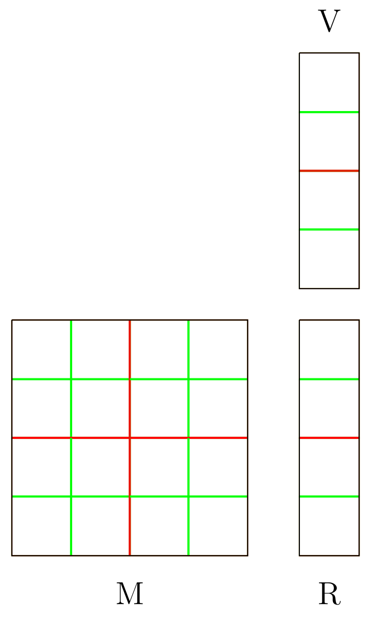


Figure 3.2: Generalized matrix-vector multiplication (GEMV).

The tuning parameters in the example (see Figure 3.2) are colored in green and red. Take the matrix M for example: The green squares represent the number of work items and the red squares represent the number of work groups. This means the matrix M has four work groups and one work group has four work items. The tuning parameters also have to adhere to certain constraints:

- The number of work groups (2) in one dimension has to divide the input size of the matrix (4) in the same dimension.
- The number of work items has to divide the input size divided by the number of work groups. In this example the matrix has a size of 4 in one dimension and 2 work groups divide that dimension divided by 2 work items.

These constraints make sure that the elements of the matrix are divided evenly among the work items.

3.2 OpenTuner

Martin:
OpenTuner is a general framework for auto-tuning programs with user-defined parameters, developed by the MIT. Like explained in the introduction, auto-tuning is a search problem and unlike other auto-tuners OpenTuner is using ensembles of search techniques. Search techniques used in the OpenTuner are i.e. pattern search, greedy mutation and differential evolution, making the tuning of a wide range of different programs efficient.

To use the OpenTuner to tune a program the user has to write a Python script. That script defines what the search space looks like and how a single configuration will be tested by the tuner. At first, the user needs to inherit from the class `MeasurementInterface` and override the methods `manipulator` and `run`. Listing 3.1 shows a simple working python script. The lines that need to be changed by the user to tune his own program with a simple tuner are highlighted.

Listing 3.1: OpenTuner: Simple example.

```
class OpenTunerImpl(MeasurementInterface):
  // definition of search space
  def manipulator(self):
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(
      IntegerParameter('A_int', i_1, i_n))
    manipulator.add_parameter(
      FloatParameter('B_float', f_1, f_m))
    ...
    return manipulator

  // run of a configuration
  def run(self, desired_result, input, limit):
    // fetch configuration
    cfg = desired_result.configuration.data

    // define gcc-command
    gcc_cmd = 'g++ prog.cpp'
    gcc_cmd += ' -D A_int='+str(cfg['A_int'])
    gcc_cmd += ' -D B_float='+str(cfg['B_float'])
    ...
    gcc_cmd += ' -o ./tmp.bin'

    // compile program
    compile_result = self.call_program(gcc_cmd)
    if compile_result['returncode'] != 0:
      return Result(time=sys.maxsize)

    // run program
    run_result = self.call_program('./tmp.bin')
    if run_result['returncode'] != 0:
      return Result(time=sys.maxsize)
```

```
33
34      // return runtime of program
35      return Result(time=run_result['time'])
```

`manipulator` (lines 3-10) defines the search space. In the beginning it is required to create a `ConfigurationManipulator` object. This object manages the spearch space and the user can then add tuning parameters by using the `add_parameter` method. OpenTuner already has many different predefined parameter types like `IntegerParameter` and `FloatParameter` to add parameters. In the example, the name of the first parameter that is supposed to be tuned is `A_int` where `i_1` is the beginning and `i_n` is the end of the value range.

`run` (lines 13-35) represents a single run of one configuration. At first, the data of the configuration needs to be fetched from the `desired_result`. After that the user needs to define a command for the compiler. In the example a C++ program is tuned, so a gcc command is defined there. The first line describes which compiler is used and the name of the source file. Then the user has to specify the tuning parameters with the `-D` flag. `A_int` and `B_int` are the names of the user-defined parameters in the program. Then the program has to be compiled and executed. Furthermore the example checks the compilation and run results for errors. If an error occurs, `sys.maxtime` will be returned as the runtime to mark the tested configuration as a bad configuration. This is necessary to easily recognize invalid configurations in OpenCL programs. OpenCL programs always have the requirement that the local size needs to divide the global size and by error checking, the tuner can recognize configurations which do not meet this requirement. At the end of `run`, if no errors occurred, the calculated runtime will be returned.

3.2.1 Tuning Matrix-Vector Multiplication

Martin:
In this chapter the matrix-vector multiplication implementation GEMV, which was discussed in section 3.1, will be tuned. The implementation contains four different integer tuning parameters. NUM_WG_0 and NUM_WI_0 representing the first dimension with the value range 1-M and NUM_WG_1 and NUM_WI_1 representing the second dimension with the value range 1-N. Listing 3.2 shows the python script for tuning GEMV. Only those lines that need to be changed in the simple tuner are highlighted. `manipulator` (lines 1-11) adds the four integer parameters with the specified value ranges to the manipulator object. `run` (lines 13-21) defines the gcc-command. There the program name is altered and the four parameters are added with the `-D` flag to the command.

Listing 3.2: Tuning GEMV with a simple tuner.

```
1 def manipulator(self):
2   ...
```

```
    manipulator.add_parameter(
      IntegerParameter('NUM_WG_0', 1, M))
    manipulator.add_parameter(
      IntegerParameter('NUM_WI_0', 1, M))
    manipulator.add_parameter(
      IntegerParameter('NUM_WG_1', 1, N))
    manipulator.add_parameter(
      IntegerParameter('NUM_WI_1', 1, N))
    ...

  def run(self, desired_result, input, limit):
    ...
    gcc_cmd = 'g++ gemv.cpp'
    gcc_cmd += ' -DNUM_WG_0='+str(cfg['NUM_WG_0'])
    gcc_cmd += ' -DNUM_WI_0='+str(cfg['NUM_WI_0'])
    gcc_cmd += ' -DNUM_WG_1='+str(cfg['NUM_WG_1'])
    gcc_cmd += ' -DNUM_WI_1='+str(cfg['NUM_WI_1'])
    gcc_cmd += ' -o ./tmp.bin'
    ...
```

Listing 3.2 shows a working tuner, but it would take much time to find parameters with a good runtime. Most of the time, the tuner is working on testing invalid configurations, because the program is compiled and subsequently run for every single invalid configuration. Only after checking for runtime errors after executing the program the tuner is testing the next configuration. Tests showed that the tuner did not find a single valid configuration after 20 minutes of tuning. Additionally, the search space is very large. A matrix with an input size of 16384×16384 for example allows $7,21 \times 10^{16}$ different configurations in the search space.

To solve the problem of large search spaces the value range and input size can be restricted to powers of two. That reduces the search space tremendously: In the example of the 16384×16384 matrices, it reduces the space from 7.21×10^{16} to 50625. `manipulator` in listing 3.4 shows the restriction of the parameters value ranges to the exponents of powers of two. Below the values of the tuning parameters will therefore be exponents of powers of two. In the example of the 16384×16384 matrices, `DATA_SIZE_M_EXP` and `DATA_SIZE_N_EXP` both have the value 14 because the result of 2^{14} is 16384.

Listing 3.3: GEMV: Requirements

```
1. dimension: NUM_WG_0 | M and NUM_WI_0 | (M / NUM_WG_0)
2. dimension: NUM_WG_1 | N and NUM_WI_1 | (N / NUM_WG_1)
```

Differing from the first implementation, where the requirements explained in section 3.1 were not used, this implementation ensures that the tuner will only test configurations meeting the requirements shown in listing 3.3.

The first requirement is that the number of work groups should divide the input size. That is always fulfilled because if there are two exponents of powers of two `X` and `Y` and `X<Y` then 2^X always divides 2^Y. Now the second

requirement NUM_WI_0 | (M / NUM_WG_0) has to be fulfilled. It is sufficient to only observe the first dimension since the reasoning is the same for the second dimension. Due to the implementation only working with powers of two M / NUM_WG_0 can be replaced with M - NUM_WG_0, because $2^{\mathtt{M}}/2^{\mathtt{NUM\_WG\_0}}$ results in $2^{\mathtt{M}-\mathtt{NUM\_WG\_0}}$. To fulfill the remainder of this requirement NUM_WI_0 has to be less than M / NUM_WG_0, otherwise NUM_WI_0 would not be a dividend. This implementation divides NUM_WI_0 with M - NUM_WG_0 and takes the remainder of this division as the new number of work items to meet this second requirement. The lines 15-18 are realizing that. DATA_SIZE_M_EXP is the exponent of the input size M. First the value of NUM_WI_0 and the value of NUM_WG_0 have to be fetched from the configuration, and using the modulo operator the remainder of the division is calculated. Since DATA_SIZE_M_EXP - cfg['NUM_WG_0'] is the biggest valid value for NUM_WI_0 one has to be added to the divisor. After calculating the new value for the number of work items the gcc command has to be defined (lines 20-26). There every parameter has to be exponentiated since this implementation is only working with the exponents. Starting from now this implementation of tuning GEMV is called the *modulo tuner*.

Listing 3.4: Tuning GEMV with the modulo tuner.

```
1  def manipulator(self):
2    ...
3    manipulator.add_parameter(
4      IntegerParameter('NUM_WG_0', 0, DATA_SIZE_M_EXP))
5    manipulator.add_parameter(
6      IntegerParameter('NUM_WI_0', 0, DATA_SIZE_M_EXP))
7    manipulator.add_parameter(
8      IntegerParameter('NUM_WG_1', 0, DATA_SIZE_N_EXP))
9    manipulator.add_parameter(
10     IntegerParameter('NUM_WI_1', 0, DATA_SIZE_N_EXP))
11   ..
12
13 def run(self, desired_result, input, limit):
14   ...
15   num_wi_0 = cfg['NUM_WI_0']
16   % (DATA_SIZE_M_EXP - cfg['NUM_WG_0'] + 1)
17   num_wi_1 = cfg['NUM_WI_1']
18   % (DATA_SIZE_N_EXP - cfg['NUM_WG_1'] + 1)
19
20   gcc_cmd = 'g++ -std=c++11 gemv.cpp'
21   gcc_cmd += ' -DNUM_WG_1_CPU='+str(2 ** cfg['NUM_WG_1'])
22   gcc_cmd += ' -DNUM_WI_1_CPU='+str(2 ** num_wi_1)
23   gcc_cmd += ' -DNUM_WG_0_CPU='+str(2 ** cfg['NUM_WG_0'])
24   gcc_cmd += ' -DNUM_WI_0_CPU='+str(2 ** num_wi_0)
25   gcc_cmd += ' -lOpenCL -w'
26   gcc_cmd += ' -o ./tmp.bin'
27   ...
```

3.2.2 Evaluation

Martin:

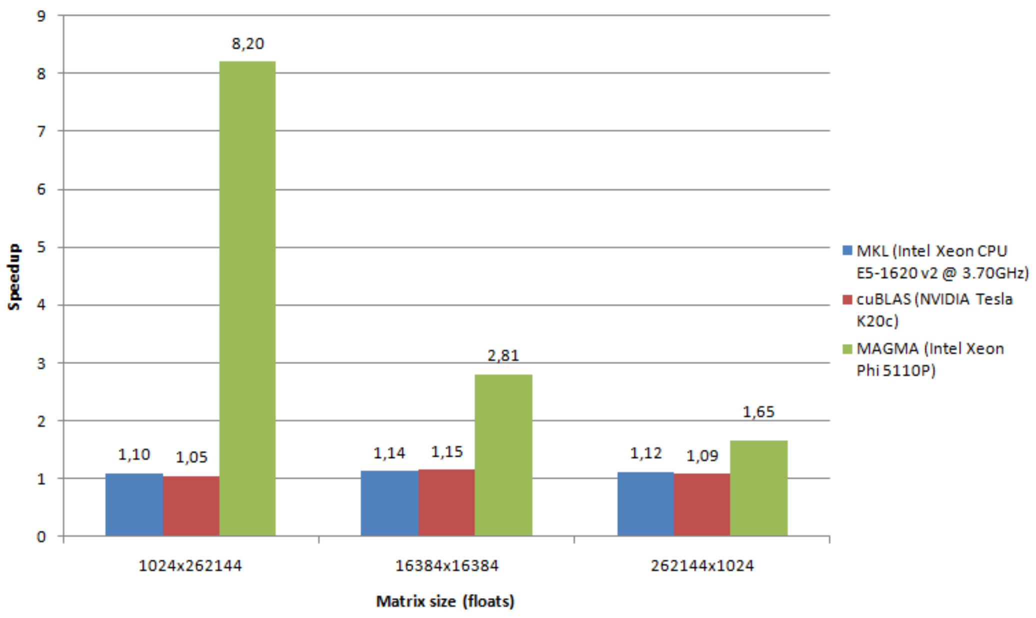
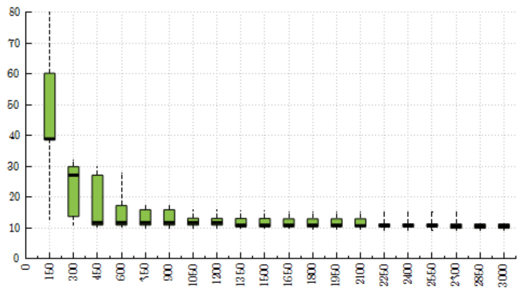


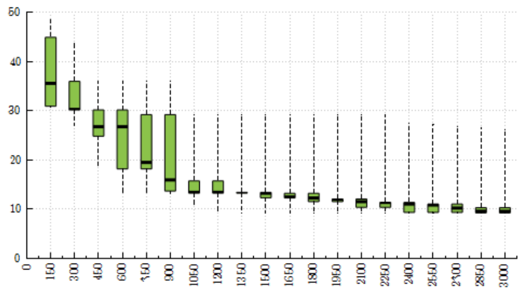
Figure 3.3: GEMV-Tuning: Comparison of the modulo tuner's results on different devices and input sizes with reference implementations.

Figure 3.3 shows the tuning run results of GEMV with the modulo tuner. The best kernel execution time found on an Intel Xeon CPU E5-1620 was compared with the Intel® Math Kernel Library(MKL[1]), on a NVIDIA Tesla K20c GPU with NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS[2]) and on an Intel Xeon Phi 5110P with MAGMA[3]. As can be seen in the figure the modulo tuner achieved a better kernel execution time than the reference implementations on every device with every input size. The kernel execution time the modulo tuner achieved on the CPU and GPU device were only slightly better than the implementations of MKL and cuBLAS, but the speedup on the Phi device was very high with results being between 1.65 and 8.20 times faster. In general, the OpenTuner provided very good results for every device that the GEMV implementation was tested on.
Figure 3.4 shows the performance of auto-tuning the GEMV implementation with the modulo-tuner on an Intel Xeon Phi 5110P and Figure 3.5 on an Intel Xeon CPU E5-1620 v2 and NVIDIA Tesla K20c. Due to longer runtimes of single configurations on Phi devices, the tuning time was chosen longer than on

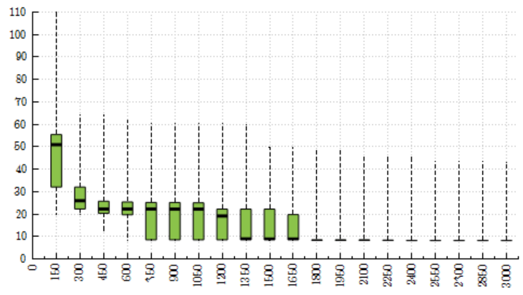
[1]`https://software.intel.com/en-us/intel-mkl`
[2]`https://developer.nvidia.com/cublas`
[3]`http://magma.maths.usyd.edu.au/magma`



(a) Input size: 1024x262144

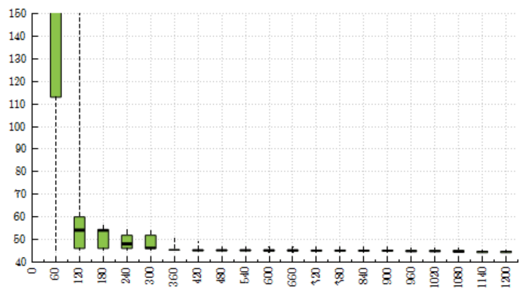


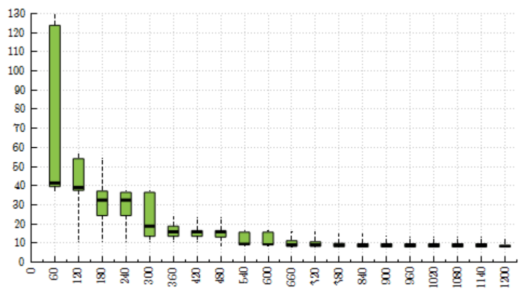
(b) Input size: 16384x16384

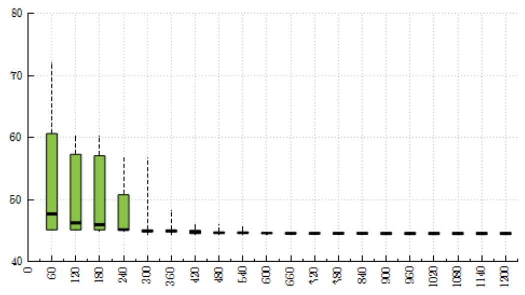


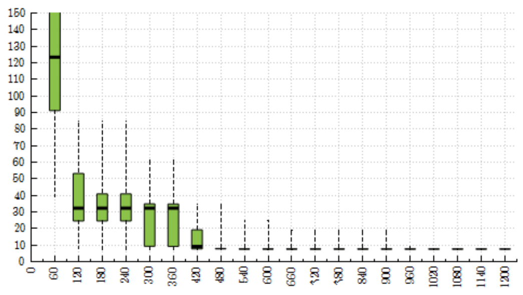
(c) Input size: 262144x1024

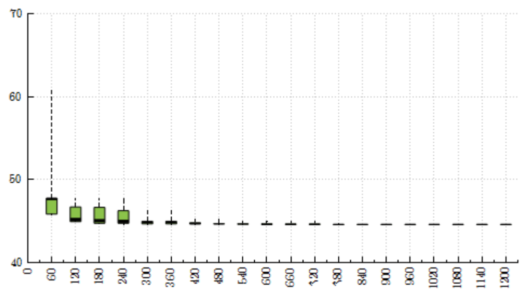
Figure 3.4: GEMV-Tuning on an Intel Xeon Phi 5110P: X-axis shows the tuning time (seconds) and y-axis shows the kernel execution time (milliseconds).


(a) Input size: 1024x262144


(b) Input size: 1024x262144


(c) Input size: 16384x16384


(d) Input size: 16384x16384


(e) Input size: 262144x1024

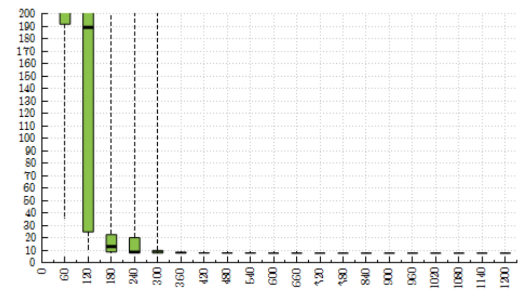

(f) Input size: 262144x1024

Figure 3.5: GEMV-Tuning on an Intel Xeon CPU E5-1620 v2(left column) and NVIDIA Tesla K20c(right column): X-axis shows the tuning time (seconds), the y-axis the kernel execution time (milliseconds) and the caption the input size of the matrix.

CPU and GPU devices. Each figure shows the results of five runs as a boxplot with the 25th and 75th percentiles. The black bar indicates the median. The boxplot shows the best result found during the given tuning time. What is noticeable is that every curve flattens after a short tuning time. It shows that even though the tuner only tested a small percentage of the search space, it already has found an almost ideal result. Every curve is also very steep in the beginning, because the first tested configurations are randomly chosen by the tuner.

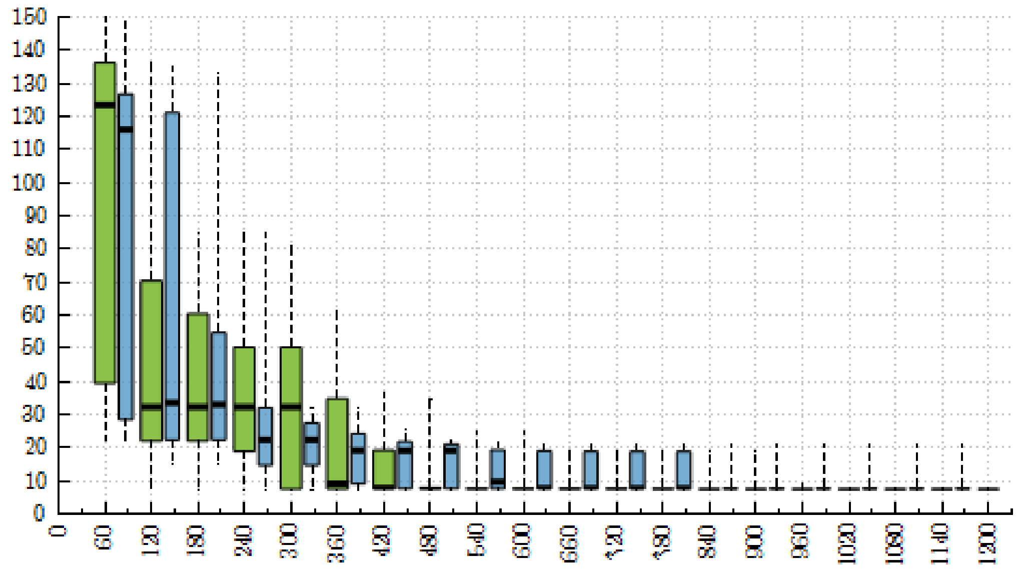


Figure 3.6: Comparison of the tuner implementations with int values (green) and float values (blue) on a NVIDIA Tesla K20c with a matrix input size of 16384 × 16384.

The implementation of the modulo tuner was only one possible way of solving the problem without spending too much time by testing invalid configurations. There are other ways to guarantee that the the tuner will only test configurations meeting the requirements. One other possibility still restricts the input size and value ranges of parameters to powers of two, but the four tuning parameters are now float parameters ranging from 0.0 to 1.0. In `run` the float parameters are then projected to their real values by multiplying the float value with the biggest valid value for their corresponding parameter. That result then has to be rounded to the next integer. Finally, the result has to be exponentiated with a factor of two. The final result will be then used in the flags for the command. That solution was proposed by Jason Ansel, one of the developers of OpenTuner. However, the results of testing this implementation on different devices and input sizes showed that the modulo tuner was slightly better in finding good configurations. Figure 3.6 shows results of ten tuning runs of this implementation on a GPU device with a matrix input size of 16384×16384. We also calculated how long both tuners took in terms of time and count of configurations to reach a result which is 5% worse than the best result found or better. In this example the tuner with the float values took 500.2 seconds and 44.5 configurations on average to reach the 5% threshold. The tuner using integer values took only 422.8 seconds and 31.1 configurations on average.

Concludingly, the OpenTuner provides a good framework for tuning diverse programs. The modulo tuner found configurations with a good runtime after a reasonable tuning time even though it only tested a small percentage of the search space. This chapter showed that for independent parameters and fixed value ranges it is simple to write a python script for a good tuner. In other circumstances however it is rather complex to create an efficient tuner.

3.3 Auto-Tuning Framework

Richard:
As seen in the last chapter, the overall performance of OpenTuner is quite satisfying. The tuned GEMV implementations could all beat the reference implementations regarding execution time. However, in terms of usability, there is a lot of room for improvements. Writing several lines of Python code makes the auto tuning process difficult, especially if no previous knowledge of the Python language is assumed.

To face these challenges, the use of the OpenTuner framework was replaced by using the Auto-Tuning Framework (ATF), an implementation of an automatic tuner by Markus Damerau, former student at Westfälische Wilhelms-Universität Münster. In his diploma thesis he developed the ATF in the fashion of a compiler. It is configured by submitting *pragmas* (directives for the ATF) that contain all necessary information about the tuning parameters.

Listing 3.5: General definition of an ATF pragma

```
#pragma tune A_int in int {i_1-i_n}, B_float in float {f_1-f_m
    }; DEFAULT(A_int) = i_1, DEFAULT(B_float) = f_m; A_int /
    B_float >= 2; order = 0;
```

Listing 3.5 shows the general form of an ATF pragma containing two tuning parameters. The user specifies the name of the parameter, the data type and range, default values that are relevant when tuning in multiple order groups, predicates that have to hold among the tuning parameters, and a number representing the order group. By submitting order group numbers, the user is able to group the tuning parameters: Each group of tuning parameters gets tuned individually, so dividing the search space into order groups, if possible, can drastically reduce the size of the search space.

Tuning a program that is annotated with these pragmas works as follows in the ATF. First, a list of all valid configurations is created. This is done by iterating over the possible combinations of values for the tuning parameters and testing them against the predicates defined in the pragmas. Then, the ATF tunes the order groups one by one, starting at the lowest order group number. Tuning them in this order means that every order group with a lower number than the currently tuned order group has already been tuned. For these, the ATF can use the best parameter values already found. Every order group with a higher order group number, is going to be tuned afterwards. So for higher order groups the default values have to be used. When the ATF has finished tuning every order group, the whole tuning process is done.

3.3.1 Tuning Matrix-Vector Multiplication

Richard:
To be able to tune the GEMV application with the ATF, three steps have to

be taken. First the application has to be annotated with pragmas as described in the last section. Second, a configuration file has to be created that tells the ATF where to look for the pragmas and how to compile and run GEMV. And third, the actual tuning process has to be started by calling the ATF executable.

The pragmas can be designed fairly easily. GEMV has four tuning parameters, two for each dimension. The tuning parameters of one dimension are dependent on each other, so they have to be defined in one pragma. Because dividing the tuning parameters into order groups is not possible in this case, the optional definition of default values is omitted. Listing 3.6 shows the resulting pragmas for a matrix size of 16384×16384.

Listing 3.6: The pragmas for tuning GEMV

```
#pragma tune NUM_WG_0 in int {1-16384}, NUM_WI_0 in int
    {1-16384};; 16384 % NUM_WG_0 == 0, 16384 / NUM_WG_0 %
    NUM_WI_0 == 0; order = 0;
#pragma tune NUM_WG_1 in int {1-16384}, NUM_WI_1 in int
    {1-16384};; 16384 % NUM_WG_1 == 0, 16384 / NUM_WG_1 %
    NUM_WI_1 == 0; order = 0;
```

Another difference in comparison to tuning with OpenTuner is that there is no longer the need to restrict the value range for the tuning parameters to powers of two manually. Because the first predicates of each pragma request that `NUM_WG_0` and `NUM_WG_1` divide the input size $16384 = 2^{14}$, only powers of two for these parameters will be valid. Similarly `NUM_WI_0` and `NUM_WI_1` are restricted to powers of two by the second predicates. These predicates will also work if the input size is not a power of two.

As mentioned earlier, another requirement for tuning with the ATF is creating a configuration file called `config_AT`. The contents are one line for the compiler call, one line for the execution and one line for a list of files that contain pragmas. Listing 3.7 shows the file for the GEMV example. When calling the ATF executable, the user has to submit the path to this configuration file.

Listing 3.7: The configuration file for tuning GEMV

```
COMPILER_CALL=g++ -std=c++11 /home/tuner/GEMV/GEMV.cpp -lOpenCL
     -w -o /home/tuner/GEMV/gemv.o
EXECUTABLE=/home/tuner/GEMV/gemv.o
FILES=params/def_param_set.h
```

The basic setup for tuning GEMV with the ATF is thereby established. But a major weakness of the ATF ist the lack of suitable search algorithms. So far only a brute force algorithm is implemented. Because the GEMV application has a search space of 14400 valid configurations for the examined input sizes and testing a configuration takes a few seconds on average, a brute force algorithm can not be utilized. In contrast, the OpenTuner has a lot of search algorithms built in. To overcome this weakness of the ATF, the OpenTuner was integrated into the ATF.

3.3.2 Integrating OpenTuner into the Auto-Tuning Framework

Richard:
Integrating OpenTuner into the ATF was done with two main goals in mind. First, the results of the tuning process should be equal to the ones obtained by tuning with OpenTuner. Second, the overall tuning process should be made easier for the user. A combination of the usability of ATF and tuning capabilities of OpenTuner promises to accomplish these goals.

When the ATF was developed, it was already designed with the aspect of extensibility in mind. An interface for search algorithms makes it possible to add new ones without altering any of the existing code. The `optimise` method of that interface has to be implemented and expects a list of all valid configurations. The ATF assembles this list and lets the `optimise` method select the configurations to test. In short, integrating OpenTuner into the ATF means calling OpenTuner from the `optimise` method. Of course calling the OpenTuner also means a fitting Python script has to be created automatically by the method.

A first step that was not directly part of the integration was to change the search space representation in the ATF from a list to a set of trees. The list representation that was present when work on the integration started was inefficient regarding the required memory space. Because every configuration also saved the parameter names redundantly, tuning with larger search spaces like the one of the GEMV example quickly drained the available RAM of the test computer (16 GB). The new representation defines one tree per pragma with two properties:

1. Every level of the tree, except for the root, corresponds to a tuning parameter

2. The parameter values along every path from the root to a leaf of the tree, corresponds to a valid configuration

Figure 3.7 shows two trees with these properties for the GEMV example and an input size of 4 in dimension 0 and 2 in dimension 1.
By changing the search space representation to a set of trees, the parameter names are less redundantly saved. Another positive aspect is the much clearer structure of a tree compared to a list regarding the predicates. When inspecting a node and therefore a parameter value in the tree, the user is easily able to identify the valid configurations containing this value by just iterating over its children. Listing 3.8 shows the interface with the new search space representation.

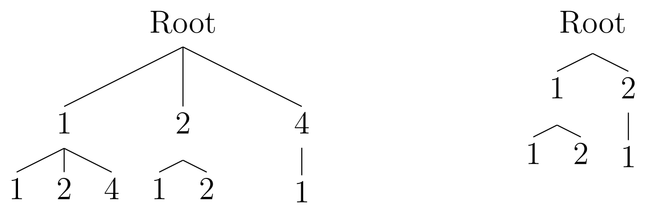


Figure 3.7: Two trees representing a search space. The first level represents valid values for NUM_WG_0 on the left and NUM_WG_1 on the right, the second levels represent NUM_WI_0 and NUM_WI_1.

Listing 3.8: The interface for search algorithms in the ATF

```
class SearchAlgorithm {
public:
    virtual paramConfig optimise(PragmaForest&) = 0;
}
```

As already described in section 3.2.2, OpenTuner is not able to handle tuning parameters with dependencies directly. The proposed solution, using modulo to project invalid values to valid ones, also works in general, when representing the search space as a set of trees. The following paragraph describes how an OpenTuner script can tune a program making use of the tree representation. A following paragraph will then describe how the process of creating such a script can be automated in the ATF.

The basic structure of the OpenTuner script does not have to be changed. A class `FrameworkMeasurementInterface` will be derived from the OpenTuner class `MeasurementInterface`. There is still the need to define tuning parameters and add them to the `manipulator` and also the `run` method needs to be implemented. But prior to that, the trees representing the search space have to be embedded in the script. The Python language has no tree type built in, so to easily define a tree-like structure, Python *dictionaries* (also known as *maps* in other programming languages) have been used.

Listing 3.9: Pseudo code representing two example trees

```
trees[(NUM_WG_0, NUM_WI_0)].child(1).child(1)
trees[(NUM_WG_0, NUM_WI_0)].child(1).child(2)
trees[(NUM_WG_0, NUM_WI_0)].child(1).child(4)
trees[(NUM_WG_0, NUM_WI_0)].child(2).child(1)
trees[(NUM_WG_0, NUM_WI_0)].child(2).child(2)
trees[(NUM_WG_0, NUM_WI_0)].child(4).child(1)

trees[(NUM_WG_1, NUM_WI_1)].child(1).child(1)
trees[(NUM_WG_1, NUM_WI_1)].child(1).child(2)
trees[(NUM_WG_1, NUM_WI_1)].child(2).child(1)
```

Listing 3.9 shows a pseudo code representation of the two example trees in figure 3.7. The Python script represents the trees in a similar way with the help of two classes. First, because a list can not directly be used as a key in a Python dictionary, a class `ListKey` wrapping a list was implemented. Second, to simplify the creation of the tree structure, a class `Tree` that wraps a dictionary and supplies a method called `get_child` was added. This method returns a value of the wrapped dictionary. If the key does not exist, it is added to the dictionary. With these alterations, the pseudo code can be expressed in Python as shown in listing 3.10.

Listing 3.10: Excerpt from the Python code representing the two example trees

```python
keyObj = ListKey(['NUM_WG_0', 'NUM_WI_0'])
paramValues[keyObj] = Tree()
paramValues[keyObj].get_child(1).get_child(1)
...
```

With the search space trees given in the OpenTuner script, the `manipulator` containing the OpenTuner tuning parameters can be created. Listing 3.11 shows in detail how this is done. The outer for loop iterates over the keys and therefore the trees contained in `paramValues`. The first inner loop and the following call to `calculateMaxChildren` are responsible for calculating the maximum number of children a node has per level. This is necessary because the maximum number of children dictates the range for the tuning parameters defined for each level of the tree. The second inner loop then iterates through the levels of the currently processed tree and adds an `IntegerParameter` for each of them to the manipulator.

Listing 3.11: The manipulator for tuning on search space trees

```python
def manipulator(self):
    manipulator = ConfigurationManipulator()
    for key in paramValues:
        for i in xrange(0, len(key.list)):
            maxChildren[i] = 0
        self.calculateMaxChildren(0, paramValues[key])

        for i in xrange(0, len(key.list)):
            manipulator.add_parameter(IntegerParameter(key.list
                [i], 0, maxChildren[i]))
    return manipulator
```

The `run` method is equivalent to the one described in section 3.2, except for the part that reads and appends the tuning parameters to the compiler call. Because in this case, the OpenTuner only tunes the indices of the tree nodes, a conversion from indices to actual parameter values is necessary. Listing 3.12 shows the conversion process. In its outer for loop it iterates over the trees contained in `paramValues`. The inner loop iterates through the tree levels or the parameter names in the current tree. The first line of the inner loop adds the parameter name and value to the compiler call. `key.list[i]` is the name of the parameter and `tree.children.keys()[...]` requests the actual parameter value. This is also where the projection with the modulo operator takes place. `cfg[key.list[i]]` is the index as supplied by OpenTuner. It can be greater than the amount of children with the same parent node, because as described earlier the OpenTuner parameters are defined with a wider range. To prevent accessing a node that does not exist `% len(tree.children)` is added. It makes sure the accessed index is valid.

Listing 3.12: Conversion of node indices to actual parameter values

```python
cfg = desired_result.configuration.data
for key in paramValues:
    tree = paramValues[key]
    for i in xrange(0, len(key.list)):
        gcc_cmd += ' -D'+str(key.list[i])+'='+str(tree.children
            .keys()[cfg[key.list[i]] % len(tree.children)])
        tree = tree.children.values()[cfg[key.list[i]] % len(
            tree.children)]
```

The second line of the inner for loop applies the projection again. But this time it accesses `tree.children.values()` which is the list of child nodes of `tree`. It does so to let `tree` point to the just processed node. The next iteration of the inner for loop will therefore operate on the next level of the tree.

The last lines of the Python script create an object of the `FrameworkMeasurementInterface` class and save it in a variable called `tuner`. This is necessary to be able to access the object from C++ later on. Additionally the `main` method of that object is called to start the tuning process.

The described Python script is able to tune a program based on the search space trees. The next step is to automatically create such a python script in the `optimise` method. To accomplish this, a template file for the script was created with place holders for the parts of the script that are dependent on the actual search space or the program to tune. The `optimise` method reads the template into a variable and replaces the place holders with actual content.

The first placeholder `:::parameter_values:::` has to be replaced by the search space trees in the way described earlier in this section. Listing 3.13 shows how the Python code for creating the trees in the python script is created. The outer for loop iterates over the trees of the search space. Then the while loop iterates over the levels of the current tree. This has to be done first to create a list of all the parameter names of the tree which is used as the key in the Python dictionary (see listing 3.9 for more details). Another thing done in the while loop is saving the index of the pragma each tuning parameter is a part of. This gets relevant when reading the best found configuration back from OpenTuner, as the order of the tuning parameters may be changed and OpenTuner also does not keep the division into multiple pragmas. After the while loop has finished, a recursive method `convertTreeToPython` is called that expects a pointer to the tree to convert, a vector of ints and a string to concatenate the created Python code to. The vector of ints represents the currently chosen path by containing an index for every level of the tree. The method then traverses the tree in a post order manner to create the necessary Python code.

Additionally to the parameter values, the compiler and executable call from the `config_AT` file are being placed in the python script. Because these placements are simple string replacements, they will not be further described in this section.

Listing 3.13: Creating the Python code for the trees automatically

```
map<string, int> paramPragmaIndices;
string paramValues;
int index = 0;
for (auto treeIter = search_Space.getTrees().begin(); treeIter
    != search_Space.getTrees().end(); treeIter++) {
    paramValues += "keyObj = ListKey(['";
    PragmaTreeNode* tmp = &(*treeIter)->getRoot();
    while (tmp->getChildCount() > 0) {
        tmp = (PragmaTreeNode*) &tmp->getChild(0);
        paramValues += ((PragmaTreeChildNode*)tmp)->getData().
            name + "', '";
        paramPragmaIndices[((PragmaTreeChildNode*)tmp)->getData
            ().name] = index;
    }
    paramValues = paramValues.substr(0, paramValues.length() -
        3) + "])\n";
    paramValues += "paramValues[keyObj] = Tree()\n";
    vector<int> path;
    convertTreeToPython(**treeIter, path, paramValues);
    index++;
}
```

After the Python script is created in the `optimise` method, it has to be executed. This can be done by embedding Python into C++. Listing 3.14 shows how to call the generated Python script from `optimise`, where `pythonCode` contains the generated Python script. To simply call a Python script, this is all that has to be done. The `PyRun_SimpleString` call blocks further execution until the script is done executing, meaning in this context, until the tuning process has finished.

Listing 3.14: Executing a Python script from C++

```
PyRun_SimpleString(pythonCode.c_str());
```

What is left to do after the tuning has finished is reading the best found configuration back to the C++ code. The Python template supplies a method `get_final_config` that returns the final configuration in form of a Python dictionary. Because these dictionaries do not exist in C++, a conversion has to be done. Listing 3.15 shows how `get_final_config` is called. In line 3 and 4 the `tuner` variable, that contains the `FrameworkMeasurementInterface` object and therefore the desired method, is being retrieved and saved in the `pTuner` variable. Line 6 gets a reference to the `get_final_config` method and saves it in the `pFunc` variable. Line 9 shows the actual call to the Python method. The return value of that call, a Python dictionary, can not directly be processed by C++ Code. The conversion in line 11 is shown in greater detail in the next paragraph. The rest of listing 3.15 mainly shows error checking for the retrieved references. Lines 22, 27 and 28 tell the embedded Python environment that the references in `pFunc`, `pTuner` and `pModule` are no longer used and garbage collection may take place.

Listing 3.15: Calling a Python method from C++

```
 1 PyObject *pModule, *pTuner, *pDict, *pFunc;
 2 PyObject *pKey, *pValue;
 3 pModule = PyImport_ImportModule("__main__");
 4 pTuner = PyObject_GetAttrString(pModule, "tuner");
 5 if (pModule != NULL && pTuner != NULL) {
 6     pFunc = PyObject_GetAttrString(pTuner, "get_final_config");
 7 
 8     if (pFunc && PyCallable_Check(pFunc)) {
 9         pDict = PyObject_CallObject(pFunc, NULL);
10         if (pDict != NULL) {
11             // convert dictionary
12         } else {
13             PyErr_Print();
14             cout << "Call failed\n";
15         }
16     } else {
17         if (PyErr_Occurred()) {
18             PyErr_Print();
19         }
20         cout << "Cannot find function \"get_final_config\"\n";
21     }
22     Py_XDECREF(pFunc);
23 } else {
24     PyErr_Print();
25     cout << "Failed to load module\n";
26 }
27 Py_XDECREF(pTuner);
28 Py_XDECREF(pModule);
```

The conversion process for the returned dictionary is shown in listing 3.16. Line 1 declares an array as large as the number of defined pragmas. The datatype `AT_PragmaTuple` is able to hold a list of `AT_ValueNamePair`, a combination of parameter name and value. The while loop then iterates through the entries of the Python dictionary. In every iteration, it saves the key (the parameter name) in `pKey` and the value (the parameter value) in `pValue`. Line 5 and 6 show the conversion of Python strings to C++ strings. Line 7 adds the value name pair to the correct `AT_PragmaTuple` by using the information collected when creating the Python code for the trees. After reading all values from the dictionary the foor loop in lines 12 to 14 pushes the `AT_PragmaTuples` into a variable called `currentBestConfig` that is returned at the end of the `optimise` method.

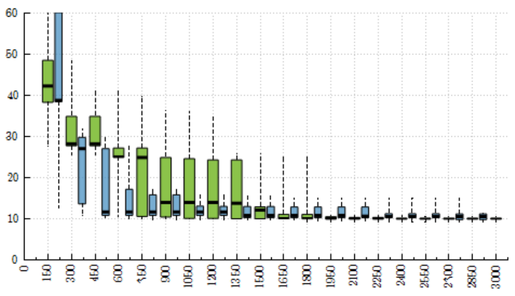
Listing 3.16: Converting a Python dictionary to C++ datatypes

```
 1 AT_PragmaTuple* pragmaTuple = new AT_PragmaTuple[index + 1];
 2 Py_ssize_t pos = 0;
 3 while (PyDict_Next(pDict, &pos, &pKey, &pValue)) {
 4     AT_ValueNamePair vnp;
 5     vnp.name = PyString_AsString(pKey);
 6     vnp.value = MultiValue(PyString_AsString(pValue));
 7     pragmaTuple[paramPragmaIndices[vnp.name]].insert(vnp);
 8 
 9     Py_DECREF(pKey);
10     Py_DECREF(pValue);
11 }
12 for (int i = 0; i < index + 1; i++) {
13     currentBestConfig.push_back(pragmaTuple[i]);
14 }
15 delete[] pragmaTuple;
16 Py_DECREF(pDict);
```

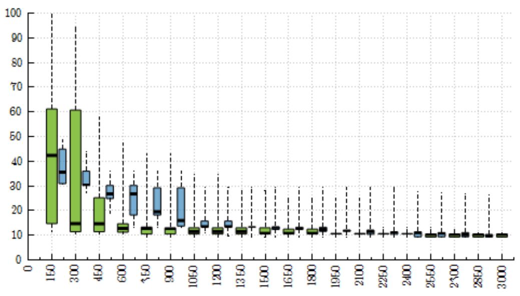
Reading the final configuration from the Python script finalizes the tuning process. As planned, the OpenTuner functionality was integrated into the ATF, combining the positive aspects of both solutions. Section 3.3.3 contains a more detailed evaluation of the modified ATF.

3.3.3 Evaluation

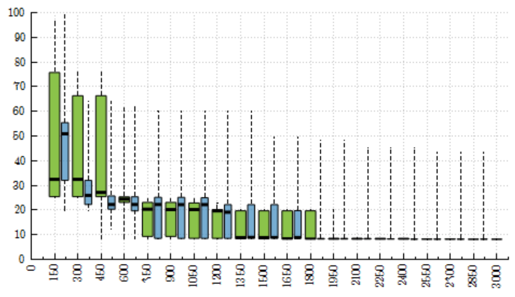
Richard:
As done with the OpenTuner, the GEMV application was tuned on the same three devices with the same input sizes. Figures 3.8 and 3.9 show the results of the tuning runs in comparison to the OpenTuner runs. The green box plots show the best results the ATF compiler found after the specified number of seconds. The smaller blue box plots to the right of each green boxplot show the results of the OpenTuner for a comparison with those values that were already presented in section 3.2.2.
The results on all devices are similar not only in terms of the final result, but also in the progress the tuner made over time. The box plots continuously shrink from left to right, almost reaching a single value at the end. Again the graphs flatten out at the end of the x-axis, a convincing sign that the ATF found a near optimal execution time.



(a) Input size: 1024 × 262144

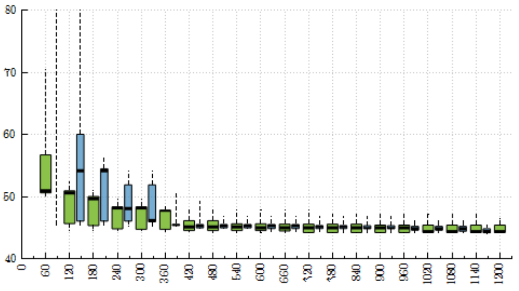


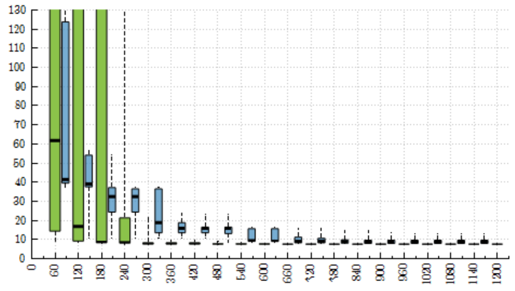
(b) Input size: 16384 × 16384

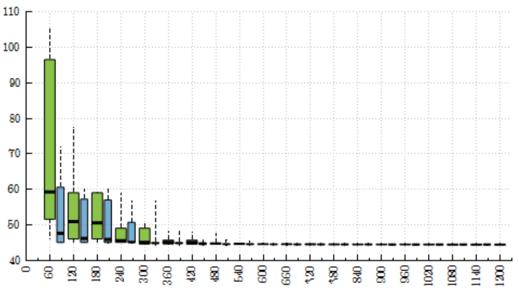


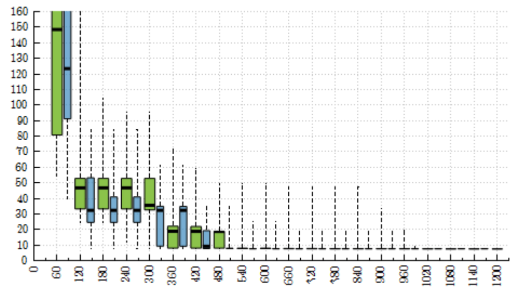
(c) Input size: 262144 × 1024

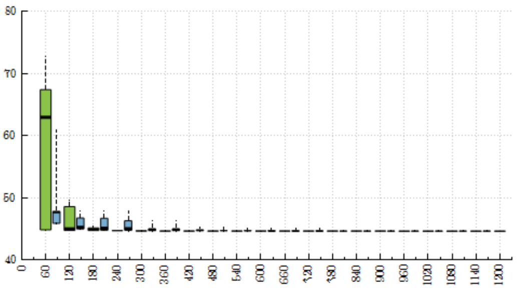
Figure 3.8: Tuning GEMV on a Intel Xeon Phi 5110P (x-axis shows tuning time in seconds, y-axis shows execution time in milliseconds.)


(a) Input size: 1024 × 262144


(b) Input size: 16384 × 16384


(c) Input size: 262144 × 1024


(d) Input size: 1024 × 262144


(e) Input size: 16384 × 16384

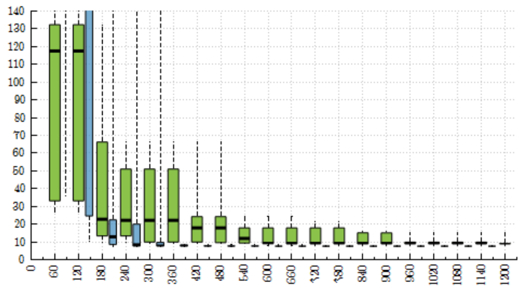

(f) Input size: 262144 × 1024

Figure 3.9: Tuning GEMV. Left: On an Intel Xeon CPU E5-1620 v2. Right: On a NVIDIA Tesla K20c (x-axis shows tuning time in seconds, y-axis shows execution time in milliseconds)

The goals when integrating OpenTuner into the ATF compiler were to maintain the performance of OpenTuner while simultaneously simplifying the tuning process. A direct comparison of the required code to tune GEMV with the OpenTuner and with the ATF compiler shows that these goals were achieved (figure 3.10). Tuning with the ATF compiler reduces the necessary lines of code to a few lines, as opposed to implementing a whole class in Python for tuning with OpenTuner.

```python
#!/usr/bin/env python
#
# Optimize number of work groups and work items of GEMV.cpp
#

import sys
sys.path.insert(0, '../ATFramework/')
import datetime
import math
import opentuner
from opentuner import ConfigurationManipulator
from opentuner import IntegerParameter
from opentuner import FloatParameter
from opentuner import MeasurementInterface
from opentuner import Result

DATA_SIZE_M_EXP = 14
DATA_SIZE_N_EXP = 14

class MatrixVectorMultiplicationTuner(MeasurementInterface):
  def manipulator(self):
    """
    Define the search space by creating a
    ConfigurationManipulator
    """
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(
      IntegerParameter('NUM_WG_0', 0, DATA_SIZE_M_EXP))
    manipulator.add_parameter(
      IntegerParameter('NUM_WI_0', 0, DATA_SIZE_M_EXP))
    manipulator.add_parameter(
      IntegerParameter('NUM_WG_1', 0, DATA_SIZE_N_EXP))
    manipulator.add_parameter(
      IntegerParameter('NUM_WI_1', 0, DATA_SIZE_N_EXP))
    return manipulator

  def run(self, desired_result, input, limit):
    """
    Compile and run a given configuration then
    return performance
    """
    cfg = desired_result.configuration.data

    num_wi_0 = cfg['NUM_WI_0'] % (DATA_SIZE_M_EXP - cfg['NUM_WG_0'] + 1)
    num_wi_1 = cfg['NUM_WI_1'] % (DATA_SIZE_N_EXP - cfg['NUM_WG_1'] + 1)

    gcc_cmd = 'g++ -std=c++11 gemv/gemv/GEMV.cpp'
    gcc_cmd += ' -DNUM_WG_1_GPU='+str(2 ** cfg['NUM_WG_1'])
    gcc_cmd += ' -DNUM_WI_1_GPU='+str(2 ** num_wi_1)
    gcc_cmd += ' -DNUM_WG_0_GPU='+str(2 ** cfg['NUM_WG_0'])
    gcc_cmd += ' -DNUM_WI_0_GPU='+str(2 ** num_wi_0)
    gcc_cmd += ' -lOpenCL -w -I/usr/local/cuda/include'
    gcc_cmd += ' -o ./tmp.bin'

    compile_result = self.call_program(gcc_cmd)
    assert compile_result['returncode'] == 0

    run_cmd = './tmp.bin'
    run_result = self.call_program(run_cmd)
    if (run_result['returncode'] != 0):
      return Result(time=sys.maxsize)
    else:
      f = open('results.csv', 'r')
      results = f.read()
      return Result(time=float(results.split(';')[0]))

  def save_final_config(self, configuration):
    """called at the end of tuning"""
    print "Optimal block size written to mvm_final_config.json:", configuration.data
    self.manipulator().save_to_file(configuration.data,
                                    str(datetime.datetime.now())+'mvm_final_config.json')


if __name__ == '__main__':
  argparser = opentuner.default_argparser()
  MatrixVectorMultiplicationTuner.main(argparser.parse_args())
```

(a) Python LOC: 75

```
#pragma tune NUM_WG_0 in int {1-16384},
        NUM_WI_0 in int {1-16384};;
        16384 % NUM_WG_0 == 0,
        16384 / NUM_WG_0 % NUM_WI_0 == 0;
        order = 0;
#pragma tune NUM_WG_1 in int {1-16384},
        NUM_WI_1 in int {1-16384};;
        16384 % NUM_WG_1 == 0,
        16384 / NUM_WG_1 % NUM_WI_1 == 0;
        order = 0;
```

(b) ATF LOC: 10

Figure 3.10: Comparison of the necessary code for tuning GEMV with OpenTuner on the left and ATF on the right

A problem still left in both OpenTuner and ATF is the rather long time needed for testing a single configuration in comparison to the measured execution time. For the input size of 16384 × 16384 on the GPU for example, testing a single configuration (excluding the tests of invalid configurations) took 12 seconds on average, while the average measured execution time was around 1 second and the final result just a few milliseconds. This indicates a large overhead in testing configurations.

3.4 Auto-Tuning Library

(Ber:) To speed up the tuning process the *Auto-Tuning Library* (hereafter simply called "library") was implemented. A secondary goal was to make the definition of the tuning parameters more intuitive for the user. In order to tune an application the tuning parameters are written as C++ code instead of using pragmas. Firstly one has to define the tuning blocks (TP blocks):

Listing 3.17: Library TP Blocks

```
// TP blocks
atf::TP_Block<1> tp_block_a, tp_block_b;
```

Dependent tuning parameters will be put into the same block. The template parameter `1` specifies the amount of tuning parameters that will be added to the block. Next, the tuning parameters are added:

Listing 3.18: Library tuning parameters

```
tp_block_a.insert<atf::tp_int>( "A_int", i_1, i_n );
tp_block_b.insert<atf::tp_float>( "B_float", f_1, f_m );
```

When inserting a tuning parameter, its name and value range have to be set. Constraints can also be applied once all tuning parameters are added. This will decrease the number of values each tuning parameter can be set to.

Listing 3.19: Library setting constraints

```
// predicates
tp_block_a.apply_constraint( []( int A_int ){
    return A_int % 2 == 0; } );
tp_block_b.apply_constraint( []( float B_float ){
    return ceil(B_float) % 4 == 0 ; } );
```

Once the constraints are applied, the tune function has to be called. This will invoke the tuning run. As parameters, the function to tune (requiring a configuration and returning a `double`) and the configurations for each tuning block have to be put in:

Listing 3.20: Library wrapper

```
atf::AbortCondition cond;
atf::Tuner tuner(cond) ;
auto best_config = tuner.tune(func,
        {tp_block_a.get_configurations(),
         tp_block_b.get_configurations()} );
```

The result of the tuning run is the best configuration, for example the configuration with the lowest run time.

3.4.1 Motivation

(Ber:) The main reason to use the library is to accelerate the tuning itself. As described in 3.3.3 on page 33, the tuning is slow because the Auto-Tuning Framework would search for a configuration, compile the application with the new values and run it. OpenCL has to be initialized and the data has to be copied from host to device, as well. The result - for instance, tuning time - of the kernel has also to be written into a file, so the Auto-Tuning Framework can read and save it. The library can bypass those weaknesses because it is an extension which is integrated into the application to be tuned. That way the kernel can be called directly via a method and OpenCL has to be initialized only once. This means the application will only be compiled once and a loop will call the kernel with new configurations and save the results, as well as report the best configuration once finished. The library allows for a much more intuitive use compared to the Auto-Tuning Framework. That's because the library can be used by writing C++ code into the application that is going to be tuned. No more pragmas have to be written and the installation of the Auto-Tuning Framework is skipped as well. To summarize, the library is an extension which allows a simpler way to tune an application. Furthermore the tuning itself is a lot faster compared to the Auto-Tuning Framework.

3.4.2 Implementation

(Ber:) When starting the implementation of the library a few modules were already implemented. Firstly the creation of the search space was already working, and secondly it was already possible to create tuning parameter blocks, as well as to insert tuning parameters into the blocks and set constraints upon the tuning parameters, just like shown above. When creating the search space, the library would iterate through each block to evaluate the constraints set upon the tuning parameters and save only those configurations that meet the requirements. One of the missing parts was a tuner itself, furthermore the wrapper function to call the OpenCL kernel was missing. It wasn't possible to set abort conditions for the tuner to have influence on the tuning run, since no tuner was integrated yet and a module to set conditions was missing completely.

Tuner class

(Ber:) In order to make the integration of other tuner into the Auto-Tuning Library as simple as possible, the *Tuner* class was implemented. When integrating a new tuner, it has to inherit the Tuner class (see Figure 3.11):

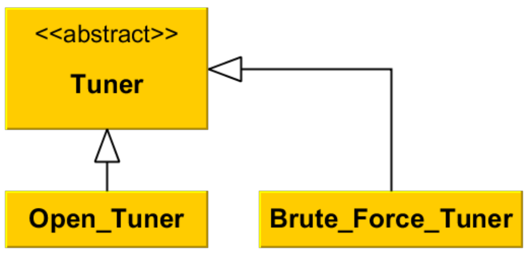


Figure 3.11: UML diagram for the Tuner class.

The Tuner class has a function called `tune` which describes the general order of events that happen during the tuning. It is implemented according to the template method pattern:

Listing 3.21: Implementation of the `tune` method.

```
virtual std::vector <TP_Value> tune(
        std::function<double(Configuration)> program,
        std::vector <Configuration_Tree> configs) {
 tuningStart = std::chrono::high_resolution_clock::now();
 bestResult = std::numeric_limits<double>::max();

  initialize(configs);
  double result;
  while (!abortCondition.checkCondition(*this)) {
  Configuration config = getNextConfiguration(configs);
  try {
   result = program(config);
  } catch (...) {
   if (abortOnError) {
    abort();
   } else {
    result = std::numeric_limits<double>::max();
   }
  }
  testCount++;
  if (result < bestResult) {
   bestResult = result;
   bestConfiguration = config;
   bestResultHistory.push_back(
     std::tuple < unsigned int,
     std::chrono::high_resolution_clock::time_point,
     double > (testCount,
       std::chrono::high_resolution_clock::now(),
       bestResult));
  }
  processResult(result);
 }
 finalize();
 return bestConfiguration;
}
```

When the tune function is called, it will set the `tuningStart` attribute by reading the date and time. The attribute `bestResult` is set to the maximum `double` value in case the tuner doesn't find any configuration that leads to a successful run of the kernel. The `initialize` function is an abstract function that has to be implemented by any subclass. This function needs to set up everything the tuner needs in order to run the `program` function inside the `while` loop. The `while` loop keeps running until the set condition returns `true`. More information about conditions is shown later, in chapter 3.4.2 on page 46. Inside the loop the tuner calls the `getNextConfiguration` method and then starts the kernel by calling `program` with that configuration. In case an error occurs when executing a configuration, the `Tuner` class can react in two different ways, according to `abortOnError`. If set to `true`, the tuning will abort completely. If set to `false`, the measured runtime will be set to the maximum double value. After the kernel execution, the result will be tested against the current best result. In case the new result is better (a shorter run time for example) it will replace the `bestResult` attribute and the configuration will be overwritten as well. After every kernel run the result will be evaluated by the `processResult` method, for example it will be written into a database so the tuner won't test the same configuration again. After the tuning is done, the `finalize` method will clean up everything that was used by the tuner and lastly the best configuration will be returned. The abstract methods `initialize`, `finalize`, `getNextConfiguration` and `processResult` are all used inside the `tune` method, which works as a skeleton to make sure a general structure during the tuning process is kept. `tune` is a template method, so different tuners can alter the concrete tuning steps to their liking. That makes it easy to integrate or implement more tuner at a later point in time, because only the specific aspects of the tuner have to be implemented.

OpenTuner class

Martin:
The `Tuner` class formed the foundation to integrate the OpenTuner into the library. It describes how the tuning roughly works by using template methods. The first step to integrate the OpenTuner is the creation of the `OpenTuner` class that implements the `Tuner` and its template methods. By implementing this class, `OpenTuner` has to implement the `initialize`, `finalize`, `getNextConfiguration` and `processResult` methods. Therefore it is required to control every step of the tuning process. It has to be possible to fetch next configurations and report the results afterwards step by step. OpenTuner already provides an API `TuningRunManager` with which it is possible to do just that. The methods of the `TuningRunManager` will be called out of these template methods.

Listing 3.22: Python script template for integrating the OpenTuner into the library.

```
 1 # imports
 2
 3 def get_next_desired_result():
 4     global desired_result
 5     desired_result = api.get_next_desired_result()
 6     while desired_result is None:
 7         desired_result = api.get_next_desired_result()
 8     return desired_result.configuration.data
 9
10 def report_result(runtime):
11     api.report_result(desired_result, Result(time=runtime))
12
13 def finish():
14     api.finish()
15
16 parser = argparse.ArgumentParser(
17     parents=opentuner.argparsers())
18 args = parser.parse_args()
19 manipulator = ConfigurationManipulator()
20 :::parameter:::
21 interface = DefaultMeasurementInterface(args=args,
22               manipulator=manipulator,
23               project_name='atflibrary',
24               program_name='atflibrary',
25               program_version='0.1')
26 api = TuningRunManager(interface, args)
```

The requirement to tune a program with the OpenTuner is a Python script. However the Python script which will be used in the library looks different than the one that was discussed in section 3.2. Listing 3.22 shows the Python script template. Lines 20-26 of that listing shows what is called on executing the Python script. First, it parses the arguments that were passed to the script. Afterwards the script creates a `ConfigurationManipulator` object, which is required to manage the search space. `:::parameter:::` is a placeholder which has to be filled with the tuning parameters with the given program in the fashion that was discussed in section 3.2. An object of `MeasurementInterface` is needed to use the OpenTuner. The aim is for the library controlling every step of the tuning. Therefore it is not required to override methods like `manipulator` and `run` and a default `MeasurementInterface` can be used. At the end, the script creates a `TuningRunManager` object with which the library can handle the stepwise access. Lines 3-14 are showing the methods that are realizing this access and will later be called out of the library class. The methods are called `get_next_desired_result` to fetch new configurations and `report_result` to save the configuration results in the database. To fetch a new configuration, `get_next_desired_result` calls the corresponding method in a while loop. This is necessary because the API method returns `None` if the OpenTuner search techniques produced a duplicate configuration. It is required to save the received configuration in a global variable because `report_result` needs that configuration to report its result to the database. `finish` finalizes the access to the API.

The OpenTuner class, that inherits from the Tuner class, needs to implement `initialize`, `finalize`, `getNextConfiguration` and `processResult` template methods. The following listings are showing how these methods were implemented.

Listing 3.23: Implementation of the `initialize` template method.

```
void OpenTuner::initialize(std::vector<Configuration_Tree>
                              &configs) {
    std::string pythonCode =
    #include"frameworktuner.py"
    ;

    std::string parameterCode;
    for (auto iter = configs.begin(); iter != configs.end();
        iter++) {
        size_t *maxChildren =
                new size_t[iter->num_params()]{0};
        calculateMaxChildren(maxChildren, 0, *iter);

        for (int i = 0; i < iter->num_params(); i++) {
            parameterCode += "manipulator.add_parameter" +
                    "(IntegerParameter('" +
                    iter->get_name(i) + "', 0, " +
                    std::to_string(maxChildren[i]) + "))\n";
        }
        delete[] maxChildren;
    }
    size_t start_pos = pythonCode.find(":::parameter:::");
    pythonCode.replace(start_pos,
                strlen(":::parameter:::"), parameterCode);

    std::vector<std::string> argVector;
    # add arguments and format to c-string argv

    Py_Initialize();
    PySys_SetArgv(argVector.size(), argv);

    PyRun_SimpleString(pythonCode.c_str());
    pModule = PyImport_ImportModule("__main__");
    pGetNextDesiredResult = PyObject_GetAttrString(pModule, "
        get_next_desired_result");
    pReportResult = PyObject_GetAttrString(pModule, "
        report_result");
    pFinish = PyObject_GetAttrString(pModule, "finish");
}
```

Listing 3.23 shows how the `initialize` method is implemented. `initialize` initializes everything that is needed to work with the OpenTuner out of the library. The first step (lines 3-5) is to read and save the python script template code as a string. Then the parameter code has to be created to fill the placeholder in the template. The following for loop (lines 8-20) implements that. Like already discussed in the ATF section, the tuner needs to know the maximum count of children for every parameter's tree level. This number will be the top of the parameters value range. To implement that, an array for the maximum count of children is created and filled with the helper method `calculateMaxChildren`. After it's filled, this method creates a string by reading the name of the parameters out of the configuration tree and specifying the range with 0 as the bottom of the range and the calculated maximum count of children as the top of the range. Then, in lines 21-23, the placeholder gets replaced by the created parameter code string. Afterwards the method defines a C string `argv` with arguments for the OpenTuner, which will not be discussed further in this section. To execute the created Python script code and later to call the methods from the Python script, *Embedded Python* is utilized. First (lines 28-29), Embedded Python is initialized and the arguments for the execution of the Python script are set. Then the Python script gets executed with Embedded Python by calling `PyRun_SimpleString`, which initalizes the `TuningRunManager`. The following four objects (lines 31-35) are all of the type `PyObject` and they import the methods from the Python script. So after the execution of `initialize`, it is possible to call the methods in the Python script to control the tuning with the `TuningRunManager`.

Listing 3.24: Implementation of the `getNextConfiguration` template method.

```
Configuration OpenTuner::getNextConfiguration(std::vector<
    Configuration_Tree> &configs) {
    PyObject *pDict = PyObject_CallObject(
            pGetNextDesiredResult, NULL);

    std::vector<long> indices;
    for (auto iter = configs.begin(); iter != configs.end();
        iter++) {
        for (int level = 0; level < iter->num_params();
              level++) {
            PyObject *pValue = PyDict_GetItemString(pDict,
                    iter->get_name(level).c_str());
            indices.push_back(PyInt_AsLong(pValue));
        }
    }
    Py_DECREF(pDict);
    Configuration config = applyProjection(indices, configs);
    return config;
}

Configuration OpenTuner::applyProjection(std::vector<long>
      indices, std::vector<Configuration_Tree> &trees) {
20      Configuration config;
21      TP_Value_Node currentNode(0, "");
22      for (auto iter = trees.begin(); iter != trees.end();
23           iter++) {
24          currentNode = *iter;
25          for (int level = 0; level < iter->num_params();
26               level++) {
27              currentNode = currentNode.child(int(indices.at(0) %
                    currentNode.num_childs()));
28              indices.erase(indices.begin());
29              config.push_back(TP_Value(iter->get_name(level),
                   currentNode.value()));
30          }
31      }
32      return config;
33  }
```

Listing 3.24 shows how the `getNextConfiguration` method is implemented. This template method implements how a configuration will be fetched. Since the top of the parameters value range is defined as the maximum count of children, it is required to project the configuration values to valid values. In lines 2-3 the method `getNextDesiredResult` will be called from the Python script with the help of Embedded Python. This call returns a `pDict`, which is a dictionary object with the parameter name as a key and the parameter value as the value. The following for loop (lines 6-17) converts the `pDict` object into a better readable vector with long objects. After that, the values in the vector have to be projected to valid values. `applyProjection` creates the projected configuration with the vector of parameter values and the configuration trees, which then will be tested by the tuner. One configuration tree represents one tuning parameter block in the library and one tree level represents one tuning parameter. To apply the projection, the library has to find the corresponding node with the projected tuning parameter value in every tree level. The first loop (lines 22-23) iterates over the configuration trees and saves the root node in `currentNode`. The second loop(25-26) iterates over the tree level. For every level the value in the vector and the count of the children of the current node are taken and the rest of the division between these two values is calculated. That calculates the projected, final value of the tuning parameter. Then this value is saved alongside the parameters name and the value of the current node in a configuration object. Ultimately, `applyProjection` has calculated every projected tuning parameter value and returned the configuration. Therefore the tuner can start the tuning process with that configuration.

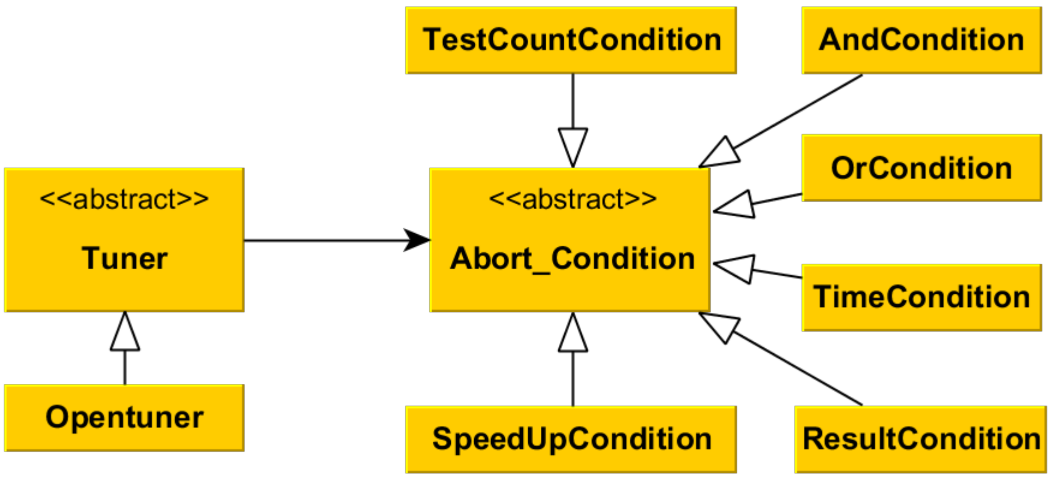
Listing 3.25: Implementation of the `processResult` and `finalize` template method.

```
1  void OpenTuner::processResult(double result) {
 2     PyObject *args = PyTuple_New(1);
 3     PyTuple_SetItem(args, 0,
 4   PyFloat_FromDouble(result));
 5     PyObject_CallObject(pReportResult, args);
 6     Py_DECREF(args);
 7 }
 8
 9 void OpenTuner::finalize() {
10     PyObject_CallObject(pFinish, NULL);
11
12     Py_XDECREF(pFinish);
13     Py_XDECREF(pReportResult);
14     Py_XDECREF(pGetNextDesiredResult);
15     Py_DECREF(pModule);
16     Py_Finalize();
17 }
```

Listing 3.25 shows how the `processResult` and `finalize` method are implemented. The final steps are describing how the result will be reported back to the database and how the tuning process is finalized. `processResult` (lines 1-7) creates new `PyTuple args`, with which it is possible to save a single value. Then the value of `result` is saved into the created `PyTuple`. After that, the `reportResult` method from the Python script is called with `args`, which saves the value of `result` in the database. `Py_DECREF` is needed to decrement the reference count. In `finalize` (lines 9-17) the `finish` method in the Python script is called to finalize the API access. Then the reference count again needs to be decremented for the `PyObjects`. Finally, the `Py_Finalize` method of Embedded Python is called which is mandatory to finish the use of Embedded Python.

Abort Conditions

(Ber:) In order to control how long the tuning of an application should run, certain conditions can be set before starting the tuning run. `Abort_Condition` is an abstract class with a method called `checkCondition` and designed after the composite pattern. `AndCondition` and `OrCondition` act as the composite and can combine the rest of the conditions. The method should return `true` in case the condition is met and `false` if not.



As of now there are four general conditions and all of them have to implement the `checkCondition` method:

- `TestCountCondition`, a condition that is used to control how many configurations are going to be tested by the tuner before stopping. Once the number of tested configurations reaches the counter set in this condition the tuning run will stop. Even configurations that lead to errors by the kernel count towards this counter.
- `TimeCondition`, this condition controls how long a tuning run will take. Once the timer set in this condition is reached the `checkCondition` method returns true and the tuning will stop.
- `ResultCondition` will tune until a specified result, for example a certain runtime, is reached.
- `SpeedUpCondition` will stop the tuning if a certain speed up is not reached. This class can be instantiated by using one of the following constructors. The tuning will either stop if the submitted speed up is not reached after a certain amount of configurations tested, or if the time submitted is passed before the speed up is reached.

Listing 3.26: SpeedUpCondition constructors

```
SpeedUpCondition(float speedup,
                 unsigned int configs) :
                    speedup(speedup),
                    configs(configs),
                    type(CONFIGS){}
SpeedUpCondition(float speedup,
                 std::chrono::milliseconds time) :
                    speedup(speedup),
                    time(time),
                    type(TIME){}
```

Once enough configurations are tested or enough time has elapsed, the `checkCondition` method will start to test if the speed up is reached every time a new configuration is tested.

Listing 3.27: Speed Up Condition `checkCondition`

```
bool SpeedUpCondition::checkCondition(
        const Tuner &tuner) const {
 double lastBestResult;
 double bestResult;
 bestResult = tuner.getBestResult();
 if (type == CONFIGS) {
  if (tuner.getTestCount() < configs)
   return false;
   lastBestResult = tuner.getBestResult(
              tuner.getTestCount() - configs);
 } else {
  if (std::chrono::high_resolution_clock::now()
      - time < tuner.getTuningStart())
   return false;
  lastBestResult = tuner.getBestResult(
          std::chrono::high_resolution_clock::now()
          - time);
 }
 return lastBestResult < bestResult * speedup;
}
```

The `checkCondition` method will check if the `lastBestResult` which was reached either a certain time or a certain number of configurations ago is smaller than the new `bestResult` multiplied by the `speedup`. In case the `lastBestResult` is smaller than the new `bestResult * speedup` the condition is met, which means the speed up was not reached and the tuning will stop.

- `AndCondition` is a composite condition that is used to combine two or more of the conditions described above.

Listing 3.28: `AndCondition checkCondition`

```
bool AndCondition::checkCondition(const Tuner &tuner)
    const {
 for (auto iter = conditions.begin(); iter != conditions.
     end(); iter++) {
  if (!iter->get().checkCondition(tuner)) {
   return false;
  }
 }
 return true;
}
```

The `checkCondition` method of this class iterates through the `checkCondition` methods of all conditions that are inside the `vector conditions` inside the `AndCondition`. If one condition evaluates to `false` the tuning run will stop.

- `OrCondition` is equal to `AndCondition` with the exception that one condition has to return `true` in order for the tuning run to stop.

Kernel Wrapper

Richard:
A main goal when developing the ATF library was to be able to tune OpenCL and CUDA kernels efficiently. As described in the motivation section, the main problem of the ATF compiler was repeatedly compiling and starting the program to tune and thus creating a significant overhead. The kernel wrapper for OpenCL kernels described in this section was able to reduce this overhead by minimizing compilation and allocation work. An analogous CUDA kernel wrapper was also created. It follows the same principles as the OpenCL kernel wrapper and only differs in the syntax used for implementing it. It will therefore not be described in greater detail.

The basic structure for the kernel wrapper is to move the OpenCL initializiation, the creation of buffers and the data transfer to the creation of the wrapper. This reduces the effort of testing a configuration to simply compiling and calling the kernel. Section 3.4.2 describes how the `Tuner` class tunes a function that takes a `Configuration` and produces a `double` value. When combining these requirements with the afore mentioned movement of OpenCL specific preparation, it makes sense to implement the wrapper as a class, where the constructor is responsible for initializing the kernel execution. To be able to call that object from the `Tuner` class like a method, the parenthesis operator needs to be overridden. Listing 3.29 shows the interface of the `ocl_wrapper` class. The template parameter `T` specifies the data type for the buffers used by the kernel.

Listing 3.29: Interface of the kernel wrapper class

```
template <typename T>
class ocl_wrapper {
public:
    ocl_wrapper(const std::string &vendor_name,
                const int &device_type,
                const int &device_number,

                const std::string &source,
                const std::vector <size_t> &input_size,

                cl::NDRange global_size,
                cl::NDRange local_size,

                    const std::function<void(void *, size_t)> data_generator = float_generator
    );

    double operator()(const Configuration &configuration);
};
```

The constructor expects a vendor name, device type and device number. These parameters are used to determine the device used by the wrapper. Next, the source of the kernel is submitted. To be able to tune a kernel via this wrapper, it has to fulfill two conditions. First, the name of the kernel has to be „func“. This is necessary because the wrapper will look for a kernel named func when compiling the submitted source code. Second, the kernel can only take buffers as parameters. The size of these buffers is defined in the vector `input_size`. The following two parameters are used to define the global and local size of the kernel (more information on global and local sizes can be found in section 2.1). The last parameter `data_generator` is optional. It is a function used for generating input data for the buffers created in the constructor. By default, a method generating random float values is used, as this is necessary in most cases. Generating data for the buffers is an important task to be able to know what type of data the buffers are initialized with. Initializing the buffers with zeros may have an impact on kernel execution time, because the compiler can simplify calculations that contain zeros. Giving the user the ability to specify the data used for initializing the buffers prohibits unwanted effects of zero-initialized buffers.

The overridden parenthesis operator is responsible for testing a single configuration. Because creation of the buffers and data transfer were already handled by the constructor, the only thing left to do is compiling and executing the kernel. The tuning parameters contained in the submitted configuration are handed to the OpenCL compiler as a flags string as seen in listing 3.30 in lines 2 to 9. The kernel then gets built (lines 11 to 22), the kernel arguments (the buffers created in the constructor) are being set (lines 24 to 26) and the kernel is executed (lines 28 to 30). Lines 32 to 37 show the measurement of the execution time via OpenCL events.

Listing 3.30: Interface of the kernel wrapper class

```
1  double operator()( const Configuration& configuration ) {
2      // create flags
3      std::stringstream flags;
4
5      for( auto& param : configuration )
6          flags << " -D " << param.name() << "=" + param.value();
7
8      for( size_t i = 0 ; i < _input_size.size() ; ++i )
9          flags << " -D " << "N_" << i+1 << "=" << _input_size[i];
10
```

```
    // build kernel
    try {
        _program.build( std::vector<cl::Device>( 1, _device ),
            flags.str().c_str() );
    } catch(cl::Error& err) {
        if( err.err() == CL_BUILD_PROGRAM_FAILURE ) {
            auto buildLog = _program.getBuildInfo<
                CL_PROGRAM_BUILD_LOG>( _device );
            std::cout << "\nBuild failed! Log:\n" << buildLog
                << std::endl;
        }
        throw err;
    }

    auto kernel = cl::Kernel( _program, "func", &err );
        check_error( err );

    // set kernel arguments
    for( cl_uint i = 0 ; i < _input_buffers.size() ; ++i )
        kernel.setArg( i, _input_buffers[i] );

    // start kernel
    cl::Event event;
    auto error = _command_queue.enqueueNDRangeKernel( kernel,
        cl::NullRange, _global_size, _local_size, NULL, &event )
        ; check_error( error );

    // profiling
    cl_ulong start_time;
    cl_ulong end_time;
    event.wait();
    event.getProfilingInfo( CL_PROFILING_COMMAND_START, &
        start_time );
    event.getProfilingInfo( CL_PROFILING_COMMAND_END,   &
        end_time   );

    return end_time - start_time;
}
```

With this wrapper a basic tuning process is possible. However, part of the requirements for tuning the stencil application developed in this project seminar is to be able to tune the amount of work groups (equal to global size divided by local size) and amount of work items per work group (equal to local size) started. A subclass of `ocl_wrapper` was implemented to make such tunings possible.

The class `ocl_md_hom_wrapper` overrides the parenthesis operator defined in `ocl_wrapper`. Before executing the kernel it looks for parameter names starting with „NUM_WG“ and „NUM_WI“ to calculate the appropriate global and local size. It then calls the original parenthesis operator to make use of the already implemented kernel execution.

3.4.3 Tuning Matrix-Vector Multiplication

(Ber:) When tuning GEMV with the library everything needed by the tuner will be written into the application as C++ code. When defining the tuning blocks:

Listing 3.31: Tuning Blocks in GEMV Library

```
atf::TP_Block<2> tpm, tpn;
```

in GEMV, two blocks need to be created, since the input matrix has two dimensions. Next the tuning parameters have to be set:

Listing 3.32: Tuning Parameters in GEMV Library

```
tpm.insert<atf::tp_int>("NUM_WG_1",
    atf::int_set(1, 2, 4, 8, 16, 32, 64,
                 128, 256, 512, 1024, 2048,
                 4096, 8192, 16384));
tpm.insert<atf::tp_int>("NUM_WI_1",
    atf::int_set(1, 2, 4, 8, 16, 32, 64,
                 128, 256, 512, 1024, 2048,
                 4096, 8192, 16384));
tpn.insert<atf::tp_int>("NUM_WG_2",
    atf::int_set(1, 2, 4, 8, 16, 32, 64,
                 128, 256, 512, 1024, 2048,
                 4096, 8192, 16384));
tpn.insert<atf::tp_int>("NUM_WI_2",
    atf::int_set(1, 2, 4, 8, 16, 32, 64,
                 128, 256, 512, 1024, 2048,
                 4096, 8192, 16384));
```

All four tuning parameters have a range from 1 to 16384, and only values that are powers of two are used. The powers of two are submitted as a set in order to speed up the tuning (more details in section 3.4.5 on page 56). The following constraints restrict the tuning parameters even more, as needed by the GEMV application:

Listing 3.33: Constraints in GEMV Library

```
tpm.apply_constraint([=](int NUM_WG_1, int NUM_WI_1) {
    return M / NUM_WG_1 % NUM_WI_1 == 0;});
tpn.apply_constraint([=](int NUM_WG_2, int NUM_WI_2) {
    return N / NUM_WG_2 % NUM_WI_2 == 0;});
```

The constraints are the same as described in 3.1 on page 14. Next the input sizes have to be calculated and initialized. Then the wrapper has to be initialized:

Listing 3.34: Initialize wrapper in GEMV Library

```
std::vector<size_t> kernel_input_size = { M*N , N , M };
```

```
std::vector<size_t> mda_input_size    = { M   , N     };
std::string vendor = "NVIDIA";
atf::ocl_md_hom_wrapper<float, 1, 1> wrapper(
        vendor, CL_DEVICE_TYPE_GPU, 0,
        source, kernel_input_size, mda_input_size);
```

The wrapper needs to know the input sizes of the matrix, the vector and the result vector. Furthermore the type of device has to be specified, in this case the tuning will run on a NVIDIA GPU. Lastly the wrapper needs to know the kernel code, here submitted via the `source` variable. Once the wrapper is initialized the conditions for the tuner and the tuner for the tuning run need to be set:

Listing 3.35: Initialize tuning in GEMV Library

```
atf::TestCountCondition cond(100);
atf::OpenTuner tuner(cond);
```

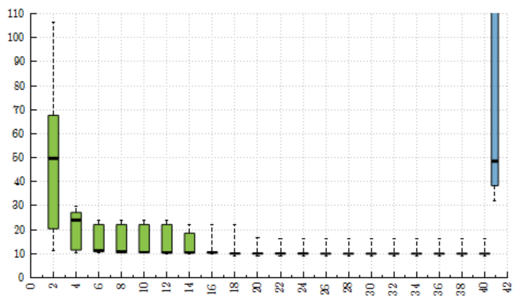
The tuning run will test 100 configurations and as a tuner, the OpenTuner is going to be used. The tuner needs to know the conditions. Next the tuning will be invoked and once done the best configuration is returned. The best result can be obtained by calling `getBestResult` on the `tuner`:

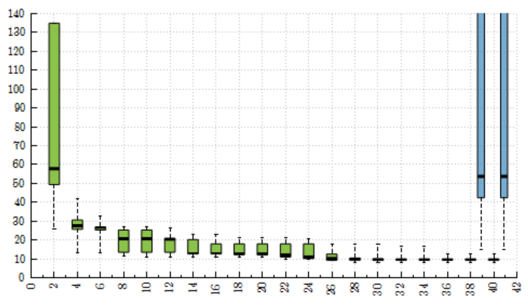
Listing 3.36: Tuning and result in GEMV Library

```
auto best_config = tuner.tune(wrapper, searchSpace);
...
double runtime = tuner.getBestResult();
```

3.4.4 Evaluation

(Ber:)
  Evaluation zu oben später.
The six graphs in figure 3.13 show the tuning of the library compared to the tuning of the compiler. For each input size and device, five tuning runs were made with the library and the compiler. The graphs show a tuning time of 20 seconds on the x-axis and the runtime of the GEMV kernel in milliseconds on the y-axis. Important to note is that the tuning curve drawn by the library in green resembles the tuning curve of the compiler, but is way faster. Looking at graph (a) for example, the library has already reached a good run time of the GEMV kernel at 20 seconds tuning time, while the compiler just finished the first configuration test. Eventually the compiler will reach the same run time of the GEMV kernel as the library, but it takes a lot longer. Sometimes, as shown in graph (d), the compiler did not test a single valid configuration in the first 20 seconds, resulting in graphs with only one boxplot series. Measuring the memory usage of the library compared to the compiler when tuning GEMV with an input size of 16384×16384 shows that the library uses 1812 KB


(a) Input size: 1024 × 262144


(b) Input size: 16384 × 16384

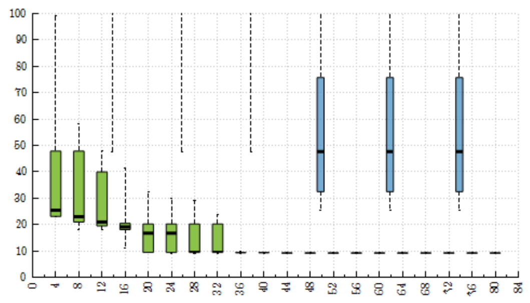
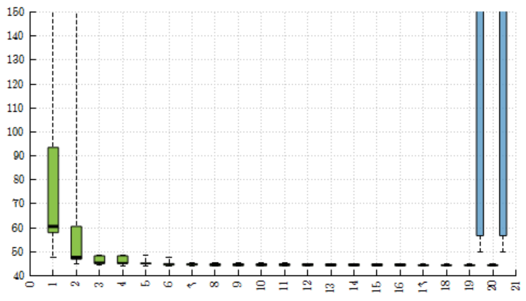
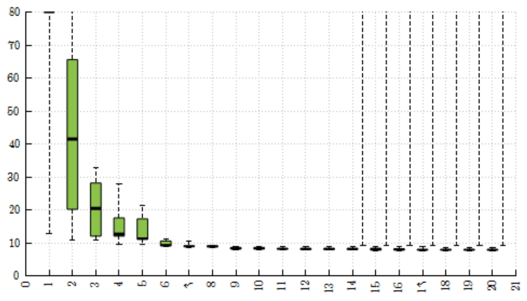
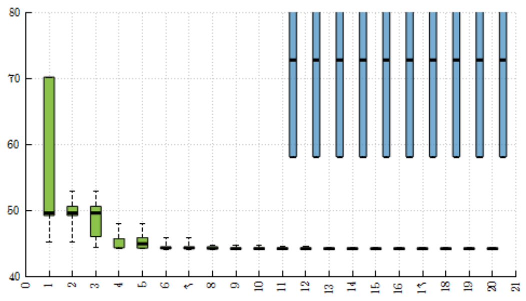

(c) Input size: 262144 × 1024

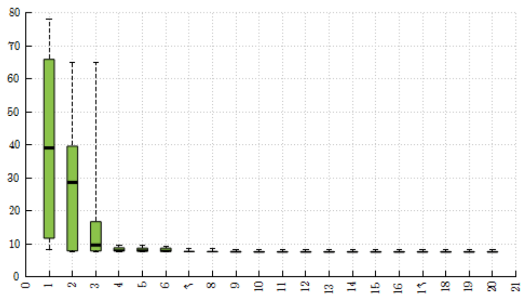
Figure 3.12: Tuning GEMV on a Intel Xeon Phi 5110P (x-axis shows tuning time in seconds, y-axis shows execution time in milliseconds.)

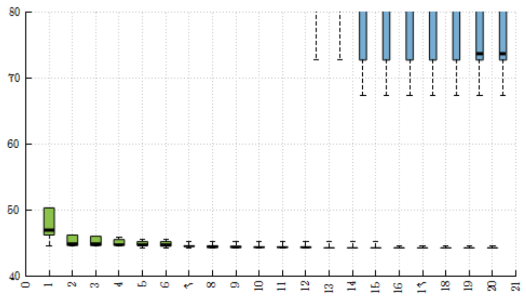
compared to the 187 MB the compiler uses. The reason behind this difference is the way the compiler saves the search space: The compiler stores the search space as a tree of configurations, where every node contains the parameter name and value. The library on the other hand stores the search space tree and the parameter names separately. Therefore the nodes only contain the parameter value.


(a) Input size: 1024 × 262144


(b) Input size: 16384 × 16384


(c) Input size: 262144 × 1024


(d) Input size: 1024 × 262144


(e) Input size: 16384 × 16384

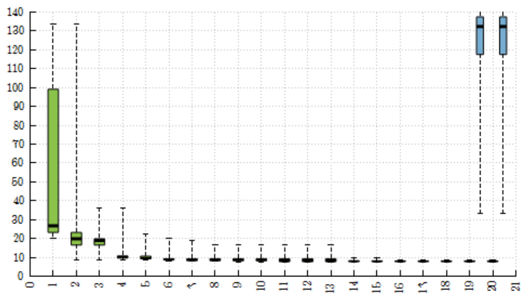

(f) Input size: 262144 × 1024

Figure 3.13: Tuning GEMV. Left: On an Intel Xeon CPU E5-1620 v2. Right: On a NVIDIA Tesla K20c (x-axis shows tuning time in seconds, y-axis shows execution time in milliseconds.)

3.4.5 Outlook

Richard:

Despite the good results produced by the ATF library, there are still some things that can be improved, but could not be realized in the time span of this project seminar.

The creation of the search space, or in more detail, the evaluation of the constraints defined on the tuning parameters is rather slow for large spaces. For the GEMV example and an input matrix size of 2^{18}x2^{10}, the creation of the search space took around 14 minutes. The cause for these long times is based on the way the constraints are applied. An input size of 2^{18}x2^{10} means there are two parameters per dimension, `NUM_WG` and `NUM_WI`. In dimension 1 these two parameters have a range from 1 to 2^{18}. Because the constraints, as defined in section 3.1, contain both tuning parameters, two nested for loops have to iterate over the complete ranges of the parameters to evaluate the constraints. This results in $2^{18} * 2^{18}$ evaluations of the constraint. To reduce this amount of evaluations, a division of the constraint into multiple constraints can be helpful. If the constraints were defined like in listing 3.37, the first constraint on NUM_WG_1 could be evaluated independently in 2^{18} iterations. This would already reduce the valid values for NUM_WG_1 and thus also reduce the iterations needed to evaluate the second constraint. Implementing these concepts is rather difficult, because constraints would no longer be defined on built-in datatypes like `int`, but also on an abstract tuning parameter type representing a whole range of values. To overcome the long preparation times when tuning the GEMV or stencil applications, defining the tuning parameters as a set of valid values as opposed to a range of values with lots of invalid ones, reduced the preparation time to a few seconds.

Listing 3.37: Alternative definition of constraints

```
auto NUM_WG_1 = atf::tp_int( "NUM_WG_1",
    atf::int_range(1,M), [=]( int NUM_WG_1 ){
        return M % NUM_WG_1 == 0; } );
auto NUM_WI_1 = atf::tp_int( "NUM_WI_1",
    atf::int_range(1,M), [=]( int NUM_WI_1 ){
        return M / NUM_WG_1 % NUM_WI_1 == 0; } );
```

Another restriction when using the ATF library is that only C++ code can be tuned. The ATF compiler on the other hand was able to tune any type of programming language because the pragmas were language independent. By combining the pragma functionality of the ATF compiler and the actual tuning capabilities of the ATF library, this weakness could be solved without really implementing anything new. The compiler would be responsibly for parsing the pragmas and generating sufficient tuning parameter declarations for the library. The library would then take on the tuning process itself.

4 Stencils

4.1 The Stencil Concept

(**Phil:**)

A *stencil* is a mathematical function of the form $stencil(M, size, func)$, where M depicts an input matrix, $size$ is a vector that defines the neighborhood of a value in M, and $func$ is a function that is applied for each value in M and uses its neighborhood values to calculate a new value for the current position.

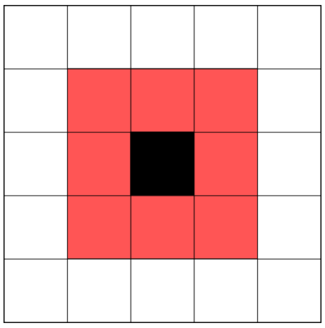


Figure 4.1: A 3×3 stencil (in red).
Here, M is a 5×5 matrix and $size = \{1, 1, 1, 1\}$. The black square in the center is the value whose stencil value is currently being calculated.

M can be of an arbitrary dimension d, although in the following work we assume $d = 2$, with most real-life usages of stencils being of the two- or three-dimensional kind.

$size$ defines the 2D-neighborhood around the current value by supplying four integer values, namely up, $down$, $left$ and $right$, which measure the extent of the neighborhood into each direction, starting from the origin value. This definition of $size$ initially only allows defining rectangular neighborhood shapes, but if non-rectangular shapes are desired, these can be achieved by ignoring a number of values in the rectangle during the calculation of the next value so the arrangement of the values from the neighborhood that are actually used in the calculation matches the desired shape.

$func$ is the core of the stencil function and defines the way new values are derived from their neighborhood values. For example, func could be

$$func = \sum_{i=0}^{n-1} i$$

with i being a value in the neighborhood (including the current value itself) and n being the number of items in the neighborhood. Thus, $func$ would simply assign the sum of all neighborhood elements to the current position.

Some stencils need to run multiple times, using the output of the first iteration as the input for the second one, and so forth - for example when calculating heat being distributed through materials. In our case, however, we only look at one iteration, where the output of the stencil calculations is saved to a separate output matrix.

The particular stencils we used in this paper are the *box blur* and *Laplace* filter stencils. The box blur is a simple blurring filter with the following *func* definition:

$$func_{(box\,blur)} = \frac{1}{n}\sum_{i=0}^{n-1} i$$

The Laplace filter on the other hand calculates an approximation of the spatial second derivative and is often used in image processing applications for edge detection means. Often, this is combined with a Gaussian filtering of the image for clearer results, leading to the “Laplacian of Gaussian” (*LoG*) kernel which we'll hereafter be referring to simply as “Laplace”. Its *func* definition is given by

$$func_{(Laplace)} = \sum_{i=0}^{n-1} factor(x_i, y_i) * i$$

where x_i and y_i refer to the column and row position of the value i relative to the neighborhood. The function $factor$ is defined as

$$factor(x, y) = -\frac{1}{\pi\sigma^4}(1 - \frac{x^2+y^2}{2\sigma^2})\, e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where σ is the smoothing constant of the Gaussian blur.

To summarize, the box blur kernel calculates the arithmetic mean of all neighborhood values, while the Laplace kernel multiplies each neighborhood value with the appropriate *LoG* factor and then computes the sum of all values.

4.2 Stencils in OpenCL and CUDA

4.2.1 Algorithm Design

(**Alex:**)

In the following we will describe the methods of the algorithms we implemented in our kernels during the project seminar. We applied most of these methods in order to gain a good kernel run time. An exception is out-of-bounds handling which was not implemented to further gain speed, but to allow the usage of untreated input matrices.

Most of our implementations are written without out-of-bounds handling in order to gain a better kernel run time. These implementations require an input matrix which is extended by a frame of 'neutral' elements (see Figure 4.2). These elements must all have the same value, which can be chosen by the programmer. The reasons for adding this specific frame to the input matrix are as follows:

The frame itself is needed to avoid out-of-bounds handling. Since every element of the input matrix needs the values from its neighborhood to compute its new value, the elements on or near the borders of the input matrix will have a neighborhood which in parts violates the borders of the input matrix. To avoid the resulting attempts of out-of-bounds accesses during the calculation of a new value, the input matrix is extended by a frame which matches the values of the vector *size*. This means that the input matrix is extended on its left side by the value $left$, on its right side by the value *right* and so on. In this way every data access which exceeds the borders of the original input matrix returns a value of the frame instead of resulting in an out-of-bounds access error.

This itself does not make the specific frame of neutral elements necessary. The uniformity of the neutral elements is required to enable the separation of the kernels. A frame of random elements would falsify the result of the box blur calculations. The reason for this is, that the corners of the frame, meaning for example for the upper left corner every element of the frame that is located left and above the input matrix itself, are excluded from the calculations. This exclusion leads to a falsification of the overall result, as long as the frame consists of random values and no explicit treatment of the corners of the frame - which would lead to a reduction of the kernel speed - takes place. The problem does not occur if the elements of the frame all have the same value. The exact reasons for this behaviour will be explained in the respective part of this chapter (4.2.1).

Blocking

The first method we applied in our kernels is blocking. The idea of blocking is that instead of assigning every entry of the input matrix to a work item,

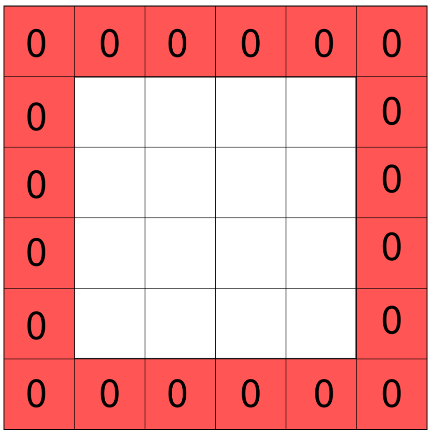


Figure 4.2: The *frame* that allows avoiding explicit out of bounds-handling.

each work item gets a whole block of data from the input matrix (see Figure 4.3). In this way, less work items are needed and additional overhead which would be the result of creating new work items can be avoided. This also adds two additional tuning parameters - *block height* and *block width* - to the kernel which allows a more accurate tuning.

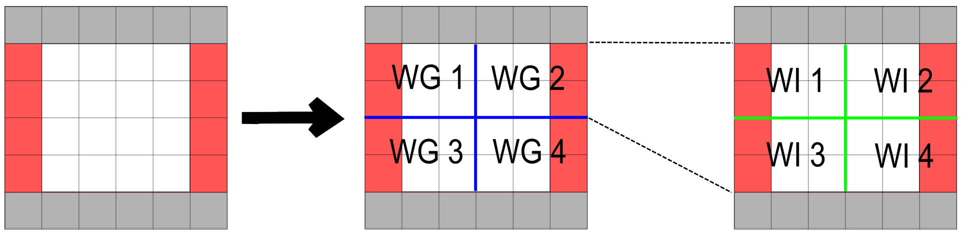


Figure 4.3: The concept of blocking. Work group borders are shown in blue, work item block borders in green.

L1 Cache and Local Memory

Every entry in the input matrix needs data from its neighborhood to calculate its new value. As a result there are overlaps regarding the needed data for the calculation of the new values for the input matrix. We can make use of this circumstance by storing the data that is needed by a work group in the faster shared memory of the GPU (respectively the cache of the CPU). In this way the required data does not need to be loaded from the slow global memory every time a work item of that work group needs it, but can be accessed in a much faster way.

There are two ways to make use of the shared memory of a GPU. One can either make explicit use of the local memory or implicitly use the L1 cache. In both approaches the data should be loaded in such a way that every work item of a work group can load the data it needs directly from the shared memory.

When making explicit use of the local memory, the scheme of the kernel algorithm is as follows: First local memory has to be allocated. This has to be done once for each work group and the memory allocated has to have as much size as needed to store the data needed by the work items of the single work groups to calculate the new values of their assigned entries of the input matrix. This data includes the data blocks assigned to the work items as well as a frame of data from the neighboring work groups which is defined by $size$. When a work group requests data from without the input matrix, the according entries of the local memory are filled with neutral values. The exact way of how the data - especially that from the neighborhood of a work group - is loaded into the local memory depends on whether or not out-of-bounds handling is implemented. In this way, all data which is needed by any entry within a work group is stored in the local memory. After the local memory is filled with the required data, the work items of the work group can calculate the new values of their entries. They can now access the data directly in the shared memory instead of accessing it in the much slower global memory every time the data is needed. This leads to a significant speedup. The problem with the local memory approach is that it might result in a loss of speed compared to an algorithm using L1 cache, when executed on a CPU. The reason is that the memory hierarchy on CPUs does not have local memory. The usage of local memory on CPU devices would only lead to an unnecessary copying of data within the RAM (see 2.1).

Because of this reason we implicitly made use of the L1 cache in many of our kernels, to gain good performances on GPUs as well as CPUs. But implementations which explicitly use local memory were also implemented in order to verify the impact of its use regarding the runtime of the kernels on CPUs. We used the auto-tuner in order to find configurations which allow a good usage of the L1 cache.

Kernel separation

The last aspect that we implemented in our kernels in order to gain speed is the separation of the kernel into a x- and y-kernel which are executed consecutively.

The idea is that instead of calculating the arithmetic mean of all values of a rectangular neighborhood, each work item calculates for each of its entries only the mean of a vector (see figure 4.4). This vector is determined by setting either the values for up and $down$ or $left$ and $right$ in the vector $size$ to zero, depending on whether the x- or the y-kernel is executed.

When the program is executed, the host first executes one of the kernels, saves the intermediate result in a matrix of the size of the input matrix (which can differ from the size of the output matrix) and then executes the second kernel on that intermediate result to finally return the output matrix. The size of the input and intermediate matrix can differ from the size of the output matrix, depending on if the input matrix is expanded by a frame. The order of the execution of the two kernels can be freely chosen.

As was already mentioned at the beginning of this section, the calculation runs into problems, if the input matrix is embedded in a frame of random values. As can be seen in figure 4.4, the corners of the frame remain unused (grey) in both kernel runs. This is problematic since their elements are part of the neighborhood of entries of the input matrix which lie near the corners and should therefore influence the new values of these entries. In the separated kernels they ought to influence the intermediate values of the nearby elements on the frame that are not used during the first kernel run. Since these elements which influence the new values of the input matrix during the second kernel run are then not correctly calculated, the result is falsified. The problem can either be handled with an explicit handling of the corners of the extended input matrix which leads to additional prompts and slower calculation or by making use of a frame of unified neutral elements. With such a frame the missing calculations on the frame itself do not matter since their result would again be the values of the neutral elements. Thus the calculation can be executed without additional prompts.

The mathematical background of kernel separation is that those kernels whose matrix rank is equal to 1 can be written as the outer product of two vectors, expressed as y- and x-kernels. This outer product is equivalent to performing the calculations with a 2D kernel, but leads to a faster computation of the same result since less calculations are necessary.

In the case of a $m \times n$ matrix and a $x \times y$ neighborhood, the unseparated version takes $mn * xy$ multiplications and additions. The separated version first runs the y-kernel, taking $mn * y$ instructions, and then runs the x-kernel, which takes $mn * x$ instructions leading to the following equation:

$$mn * (x + y) \leq mn * xy \quad \text{for } x, y > 1$$

This shows that for neighborhoods $\geq 2 \times 2$, separation provides a computation time benefit.

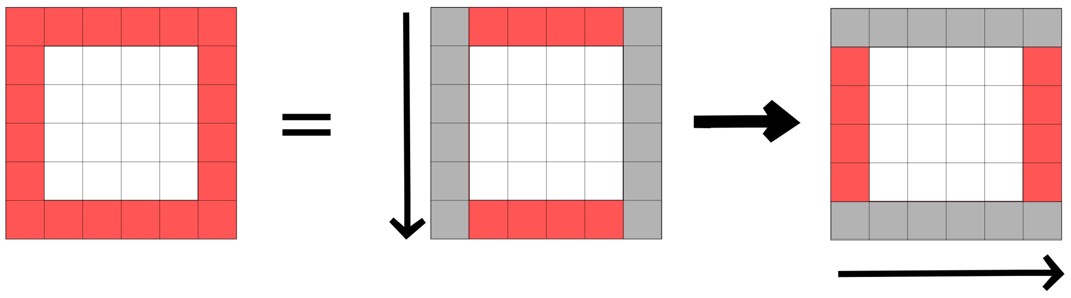


Figure 4.4: y- and x-kernels for the separated box blur.

Out-Of-Bounds Handling

In some implementations we used explicit out-of-bounds handling. This allows us to work with input matrices without an additional frame of neutral elements. It also allows us to verify which impact explicit out-of-bounds handling has on the run time of a kernel (see section 4.4.3).

The out-of-bounds handling itself works in a way that each work item checks if it is located at one of its work group's borders. If that is the case, it loads the data needed from neighboring work groups into the local memory or fills it with neutral elements if the range of the input matrix is exceeded. The concrete implementation of the out-of-bounds handling depends on the methods used in the respective program. It is very simple in its basic form, but grows more complex if local memory is used. However, the complexity of out-of-bounds handling in programs which use local memory is reduced if additionally separation of the kernels takes place.

4.2.2 Kernel implementations

(**Alex:**)

We will now give an overview and short descriptions of the implementations we have done. The implementations can be categorized as naive, blocking and separated. The separated kernels also include blocking. Of all three categories exist versions with and without out of bound handling. The blocking and separated implementations were additionally implemented in variants with and without local memory.

Naive Implementation

Our first implementations were simple box blur implementations without any of the methods mentioned in 4.2.1. The algorithm of the kernel simply computes a new value for a single entry of the input matrix. This is done by summing up the values of all entries of the neighborhood of this specific entry, as well as its own value, and dividing the result with the size of the rectangle which includes this neighborhood. The neighborhood itself is defined by the vector *size*. One can see below in the extract of the naive kernel code with an extended input matrix (listing 4.1) that ~~at~~ first *col* and *row* have to be specified which define the position of the element treated by the actual work item in the extended input matrix (lines 2 and 3). The position depends on the global ID of the work item in x- and y-direction as well as the values of *left* and *up* from *size*, which are added to the values of the global IDs to cover the offset produced by the additional frame of the input matrix. The summing up of the values of the neighborhood takes place in lines 12 to 16. There the program runs through two for loops which define the positions of the elements of the neighborhood and sums the values of these elements ~~up~~.

Finally, in line 23 the result which is gained by dividing the sum of the values of the neighborhood with the number of elements within the neighborhood is written to the output matrix.

Listing 4.1: Algorithm of the naive kernel with extended input matrix

```
 1 // retrieve this work item's global work item id in x and y
      dimensions
 2 int col = get_global_id(0) + left;
 3 int row = get_global_id(1) + up;
 4
 5  ...
 6
 7 int sum = 0; // sum of all mask elements
 8 int val;
 9
10 // get sum of all elements inside the mask
11 // centered at the (col, row)
12 for(int c_row = row - up; c_row <= row + down; c_row++)
13     for(int c_col = col - left; c_col <= col + right; c_col++)
14     {
15     sum += image[c_col + c_row * imageSize[0]];
16     }
17
18 // divide by size of mask
19 int masksize = (left + 1 + right) * (up + 1 + down); // +1
      because of "middle" element
20 int pixelValue = sum / masksize;
21
22 // write new pixel intensity value to output image
23 output[get_global_id(0) + get_global_id(1) * get_global_size(0)
      ] = pixelValue;
```

We have implemented two variants of the naive kernel. One of them with out of bound handling and the other without. As can be seen in listing 4.2, the version with out of bound handling checks for every element of the neighborhood if it is located within the input matrix. If so, the value of that entry is summed up with the values of the other entries. If not, a neutral value is added to the sum. The version without out of bound handling needs an input matrix with a frame of random elements - not necessarily neutral elements as described in 4.2.1 - to work.

Listing 4.2: Out of bound handling in the naive kernel

```
val = c_row < 0 ||
   c_row >= imageSize[1] ||
   c_col < 0 ||
   c_col >= imageSize[0] ?
   0 : image[c_col + c_row * imageSize[0]];
```

Versions with Blocking

Our first step to improve the run time of our kernels was to add blocking to the kernel algorithm. The kernel gets two additional parameters *blockHeight* and *blockWidth* (lines 7 and 8 in listing 4.3), which are used as limiters for two additional for-loops (lines 11 to 22) and define a specific area of the input matrix, a block. Each iteration of the two new loops identifies the position of an entry within the block (lines 16 and 17). After identifying the position of such an entry, its new value is calculated in the same way as in the naive kernels and the result is written to the output matrix (line 19).

Listing 4.3: Structure of the blocking algorithm

```
 1 ...
 2
 3 int blockX = get_global_id(0) * blockSize[0]; // x position of
       first element in block (internal block coordinates (0,0))
 4 int blockY = get_global_id(1) * blockSize[1]; // y position of
       first element in block
 5
 6 // get block sizes
 7 int blockWidth = blockSize[0];
 8 int blockHeight = blockSize[1];
 9
10 // calculate all positions in block
11 for (int i = 0; i < blockHeight; i++)
12 {
13   for (int j = 0; j < blockWidth; j++)
14   {
15     // find position of global matrix entry to calculate
16     int col = blockX + j;
17     int row = blockY + i;
18
19     //calculate new value for every entry within the block and
           write it to the output matrix
20
21   }
22 }
```

In versions with out of bound handling depends the implementation of the out of bound handling on whether or not an explicit usage of local memory takes place. As long as the explicit usage of local memory is not implemented, out of bound handling works as in the naive algorithm (see listing 4.2).

As can be seen in listing 4.4, the version which implements the usage of local memory first checks if the work item is not placed at any border of the work group at all (lines 1-4). If the work item is located at a border, the algorithm first checks if its located at one of the corners of the work group and after that if it is located at one of the edges (lines 5 to 53). The search is ended as soon as the fitting location is found. As can be seen in the code of the lower left and upper right corner (lines 10 to 33), the range of entries which are loaded into the local memory by the work item is extended according to the position of the work item at the border of the work group. In this way the additional data beyond the range of the work group is addressed and stored to the local memory as well as the data within the block of the work item. After the work item has written its share of data into the local memory, it waits at a barrier for the other work items of the work group (line 55) before resuming the calculation of the new values for its assigned elements (line 57). The barrier is necessary to ensure that all necessary information is available, when the work items start to calculate the new values for the entries of their blocks.

There is no version with local memory without out of bound handling. The reason is that not implementing out of bound handling results in the unnecessary copying of data which depends on the parameters of *size* and results in more speed loss, the larger the neighborhood is defined.

Listing 4.4: Out of bound handling when using local memory

```
if (block is at no border)
{
  only load data from input matrix which is specified by the
     block itself
}
else
  if (upper left corner)
  {
  ...
  }
  else if (lower left corner)
  {
    for (int i = 0; i < BLOCKHEIGHT + up; i++)
    {
      for (int j = 0; j < BLOCKWIDTH + left; j++)
      {
      check if entry is within the borders of the input matrix
      if so: add value from according input matrix entry to the
           local memory
      else: add neutral element to the local memory
      }
    }
  }
  else if (upper right corner)
  {
    for (int i = 0; i < BLOCKHEIGHT + up; i++)
    {
      for (int j = 0; j < BLOCKWIDTH + right; j++)
      {
      check if entry is within the borders of the input matrix
      if so: add value from according input matrix entry to the
           local memory
      else: add neutral element to the local memory
      }
    }
  }
  else if (lower right corner)
  {
    ...
  }
  else if (left edge)
  {
    ...
  }
  else if (upper edge)
  {
    ...
  }
  else if (right edge)
  {
    ...
  }
  else if (lower edge)
  {
    ...
  }

  barrier (CLK_LOCAL_MEM_FENCE);

  calculation of the new values
```

Versions with separated Kernels

Our final step in order to achieve a good kernel run time was to separate the kernel into a x- and a y-kernel. The kernel algorithm itself is basically the same as in the blocking implementations. Only the out-of-bounds handling is altered, since it is only necessary to check for the left and right border in the x-kernel and the upper and lower border in the y-kernel, when out-of-bounds handling is implemented. The corners and both other borders need no treatment. As stated in 4.2.1, only the values of vectors instead of arbitrary rectangles are summed up and divided by the number of entries of the vectors. In order to do so, the values for *up* and *down* in the x-kernel, respectively $left$ and *right* in the y-kernel are set to zero in the *size* vector.

The result of the first kernel which is executed has to be written into an intermediate matrix which has the same measures as the input matrix. This size can be larger than that of the output matrix if no out of bound handling takes place. The reason is the frame of neutral elements, which then surrounds the input matrix. This intermediate matrix is then used as input for the second kernel run which calculates the final result and writes it into the output matrix. The order in which the kernels are executed can be chosen freely.

Laplace

The Laplace kernel uses the blocking technique as described above. We have implemented two variants, one with and one without out of bound handling and local memory. The structure is in both cases the same as in the box blur kernels. Only the calculation of the new value of an entry is different, as can be seen in listing 4.5. For that purpose the kernel needs an additional matrix which in its measures corresponds to the neighborhood, defined by *size*. In the code below it is named *laplace_mask*. It contains the weights of the entries in the neighborhood. Instead of summing up the neighborhood and dividing the result as in the box blur kernels, the values of the entries are multiplied with their corresponding weights before adding them up (line 6). No division takes place.

Listing 4.5: Calculation of a new value in Laplace kernels without out-of-bounds handling.

```
1 float sum = 0;
2 for (int a = 0; a <= up + down; a++)
3 {
4   for (int b = 0; b <= left + right; b++)
5   {
6     sum += laplace_mask[b + a * SHAPE_SIZE_X] * localmem[(
          get_local_id(0) * BLOCKWIDTH + j + b) + (get_local_id(1)
           * BLOCKHEIGHT + i + a) * LOCALMEM_WIDTH];
7   }
8 }
```

4.3 Halide and SkelCL

(**Chris:**)

In order to have references of other works and to compare our implementation with real life examples of stencil implementations, we implemented and compared the results of our stencils with implementations using SkelCL and Halide as backends. These will be briefly discussed here and advantages and disadvantages will be outlined.

4.3.1 SkelCL

For SkelCL, we modified the `GaussianBlur` code which by default comes with the SkelCL repository as an example. To do so we changed the input parameters from the default “Lena” sample to a matrix fitting the dimension (8192×8192) of our tests. Further we overwrote the supplied user function in the file `MapOverlap1D.cl` once with an box blur and once with a Laplace stencil and then proceeded to measure the execution time of the generated kernel.

Performance Issues

After solving numerous issues with compiling and using SkelCL we found that there are some expected and some surprising results with the *MapOverlap* and *Stencil* skeletons. First of all as to be expected the MapOverlap performs faster then the Stencil skeleton and will therefore be the main focus of further discussion and will not be explicitly marked in the following comparison graphics. This is most likely attributed to the fact that we have mainly tested using stencils with square neighborhoods. Therefore the simplicity of the MapOverlap skeleton outweighs the gains of the Stencil skeleton, which greatest benefit would be to effectively support non-square stencils.

But contrary to our expectations, our SkelCL implementation performs far worse compared to even our naive, auto-tuned OpenCL implementation. We believe there are multiple reasons for that. First of all one reason is that the global and local sizes used for starting the SkelCL kernel are not directly modifiable and constant on any given testing system. This denies SkelCL the massive performance gains through auto-tuning a specific OpenCL Kernel.

The probably greater problem is the way in which work items load values used for computation in the local memory. For testing purposes we generated a SkelCL kernel and instead of directly executing it, saved the result to the file `skelcl_mapoverlap_kernel.cl`, that may be found together with the other kernel implementations in the source folder associated with this paper. By step-wise analysis of the generated kernel we found that not execution of the user algorithm is the bottleneck, but loading values from the global memory into the local memory slows down the algorithm. Only a few work items load the border regions while all other items are stuck at a synchronization barrier. In the case of the box blur kernel, calculating the result of the computation roughly takes $\approx 20\%$ of the time, while loading the values for the left and right border region takes $\approx 75\%$ of the execution time. Loading of any other values only takes the remaining $\approx 5\%$.

4.3.2 Halide

For Halide, we wrote a small program allowing us to rapidly test different Halide algorithms and schedules in order to find a configuration that allows for maximum efficiency. In conjunction with support from the Halide developers themselves, we arrived at a final configuration we then used as a reference.

Implementation Details

The Halide implementation we used for evaluating the best configuration, algorithm and schedule is divided into three parts:

Host: Defined in `main.cpp` is the code for initializing, measuring the runtime, executing and checking the result of the program.

Values: Defined in `parameter.h` are constants that can be modified by a tuner for increased performance. In `config.h` there are any other constants that affect program execution such as input size, stencil range, debug information and the combination of schedule, algorithm and device that should be used for execution.

Pipeline: Defined in `algorithm.h` and `algorithm.cpp` are blocks for Halide algorithms and schedules. These can be combined using the values in the configuration file to produce a complete Halide pipeline, which will then be executed after the host compiles it.

Furthermore the construction of the pipeline is separated into roughly four segments that may take different routes depending on the overall pipeline defined by the `USE` directives as set in the `config.h` file.

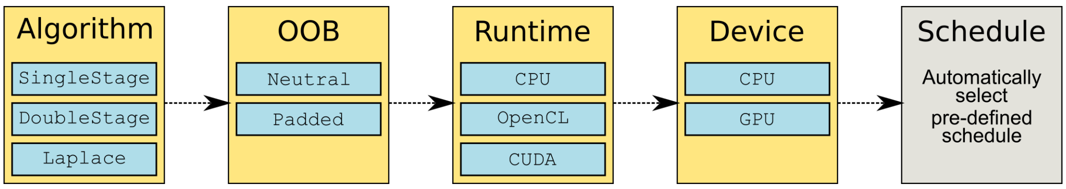


Figure 4.5: Illustration of the options for creating a Halide pipeline. In each segment one option may be chosen, resulting in 24 valid configurations.

It is possible to influence the out-of-bounds handling, the device used for execution, the algorithm and the runtime backend used, which then will determine the schedule used by the respective algorithm, as illustrated in 4.5. This is achieved by passing these values as parameter to the constructor of the `Stencil` class, where they will then be used for building the pipeline.

Upon construction of a Stencil object as defined in `algorithm.h` the constructor will first of all deterministically generate an input matrix of the size defined in `config.h`. At this point, in case `NEUTRAL_PADDING` has been selected as the out-of-bounds handling type, the input will also be padded and filled with a neutral value, so as to avoid invalid access of memory. Otherwise out-of-bounds handling will be performed at runtime through index checks which return a neutral value in case they fail.

Next, it is necessary to select the device to use for executing the pipeline. Here Halide restricts the possibilities, because support for multi-device execution of a single pipeline has not yet been implemented. For GPUs there is only a single shared context managed internally by Halide, making it impossible to switch between multiple GPU devices during program execution without full re-initialization of Halide itself. Therefore we restricted the device selection process to a simple switch, allowing to either select a CPU or GPU as the underlying device.

At this point the Halide backend to use for generating the pipeline is selected. By default the standard target is the CPU and LLVM is used as a backend for lowering Halide directives to machine code. It is also possible - and in case a GPU was selected as the target device, even necessary - to select either CUDA or OpenCL as alternative backends.

After selecting the runtime backend we proceed to define the Halide algorithm. In essence the mathematical formula of what it actually is that we are going to calculate. For the purpose of this assignment we implemented three different algorithms.

SingleStage: A box blur stencil using the enhancements described in 4.2.1, except for separation.

DoubleStage: Same as the SingleStage algorithm with the difference that the calculation has been separated in x- and y-dimensions. This is not applicable for all stencils, see section 4.2.1.

Laplace: Implementation of the Laplace Blob detection as described in section **??**.

Once all these settings have been applied, the schedule for the selected algorithm will be implicitly selected. Depending on whether a CPU or GPU has been selected as the target device it is necessary to modify the schedule appropriately, because on GPUs Halide needs to know how to divide the calculation into work groups and work items and also may not use the `vectorize` schedule statement.

The general outline of what the schedule does, however, is always the same. The calculation is tiled three times allowing to tune the work group, work item and work item block sizes. On CPUs additionally `vectorize` is used for optimizing performance.

The parameters that usually need to be passed to the schedules to define the exact division of the tiles and so forth, are the access point for tuning. Any freely modifiable parameter for any schedule has been set up for easy tuning in the `parameter.h` file, thus also separating execution code from tuning directives and increasing readability.

As the last step before executing, the Halide `compile_jit` routine is called, for Just-In-Time (JIT) compilation of the pipeline.

Profiling

It is important to note that for correct measurement of the execution time, the Halide `realize` function has to be called at least twice. This is because on the very first execution Halide will first initialize any necessary context that might be needed. Depending on the selected runtime this can take a significant amount of time. For example with OpenCL as the runtime this includes compiling the OpenCL kernel generated by Halide, copying any needed data from host to device memory and setting up the context. The cost of these operations usually dwarfs the execution time of the kernel itself for our specific test case.

Once this has been done, any subsequent call to `realize` will execute the calculation as directly as possible. Therefore we measure the execution time by first executing the pipeline once and then a second time enclosed by timing statements. On GPUs it's necessary to wait for the completion of the kernel with a `device_sync` statement.

Pros and cons

(TODO: Move this?)

As with all high-level approaches to low-level optimization problems Halide has a number of advantages but also problems resulting from them.

One of the clear advantages is the ease of use and the speed with which it is possible to generate, construct and modify a pipeline. This enables fast testing of different approaches to a problem and also makes it very easy to auto-tune. It is also possible to completely automate generation of schedules, thus alleviating the need for knowledge about the low-level workings of the underlying hardware. This is an yet ongoing part of research by numerous Halide contributors.

But this also causes some problems. For example in the case of our *DoubleStage* pipeline we tuned the tile size in which to separate the calculation. This however does not have a one-to-one correlation to global and local sizes in OpenCL. More exactly it is not possible to directly tune global and local sizes at all, but instead only to divide the calculation into tiles so that Halide is able to infer that a specific division is desirable. This then leads to the problem that the perhaps optimal division of workgroups to work items is not achievable because of limitations set by the Halide division algorithm. How much of a problem this poses in reality cannot be said without in-depth analysis of the inner workings of Halide, but tuning results suggest that the impact is relatively minor.

4.4 Results

4.4.1 Test System Hardware Specifications

(**Phil:**)

We tested the aforementioned implementations on two systems: "*Agamemnon*" and "*Palma50*", which possess different hardware (see Figure 4.6). *Agamemnon* was used for CPU and GPU tests, while *Palma50* was only used for GPU

	Agamemnon	**Palma50**
GPU	NVIDIA Tesla K20c	NVIDIA GeForce GTX480
GPU Architecture	Kepler	Fermi
Compute Capability	3.5	2.0
GPU Memory	5120MB	1536MB
Shared Memory	64KB	64KB
CUDA cores	2496	480
CPU	Intel Xeon E5-1620 v2	-
CPU cores	4 (8 via hyperthreading)	-
Clock Speed	3.70GHz	-
L1 Cache size	256KB	-
L2 Cache size	1MB	-

Figure 4.6: Hardware specifications of the two test systems *Agamemnon* and *Palma50*.

testing.

The main difference between the two systems lies in the GPU generation: While *Agamemnon* uses a NVIDIA Kepler Tesla K20c, on which L1 caching for global memory is disabled, *Palma50* uses the older NVIDIA Fermi GeForce GTX480, which allows dividing up shared memory via CUDA API calls: either L1 cache or local memory can be preferred, resulting in 48KB of memory being allocated for the preferred choice, while the other one is given 16KB of memory.

As a necessary compromise because of the differering GPU memory sizes, the size of the input matrix was chosen as 8192×8192 values because this is the largest common matrix size both GPUs can load into memory at once. The mask size was set at 15×15 because this was deemed a good compromise between a mask size that would be used in real life and reasonably high computation times that allow observing runtime differences between the different kernels more reliably.

4.4.2 Kernel Runtimes for Optimal Parameters

(**Phil**:)

The figure 4.7 shows the runtimes for all CUDA GPU kernels on the *Palma50* and *Agamemnon* systems, using the configurations the auto-tuning deemed fastest for each kernel (shown in section 6.4) for a 15×15 box blur on a 8192×8192 input matrix. The results only show the actual running time of the kernels, excluding memory transfers from and to the device in question.

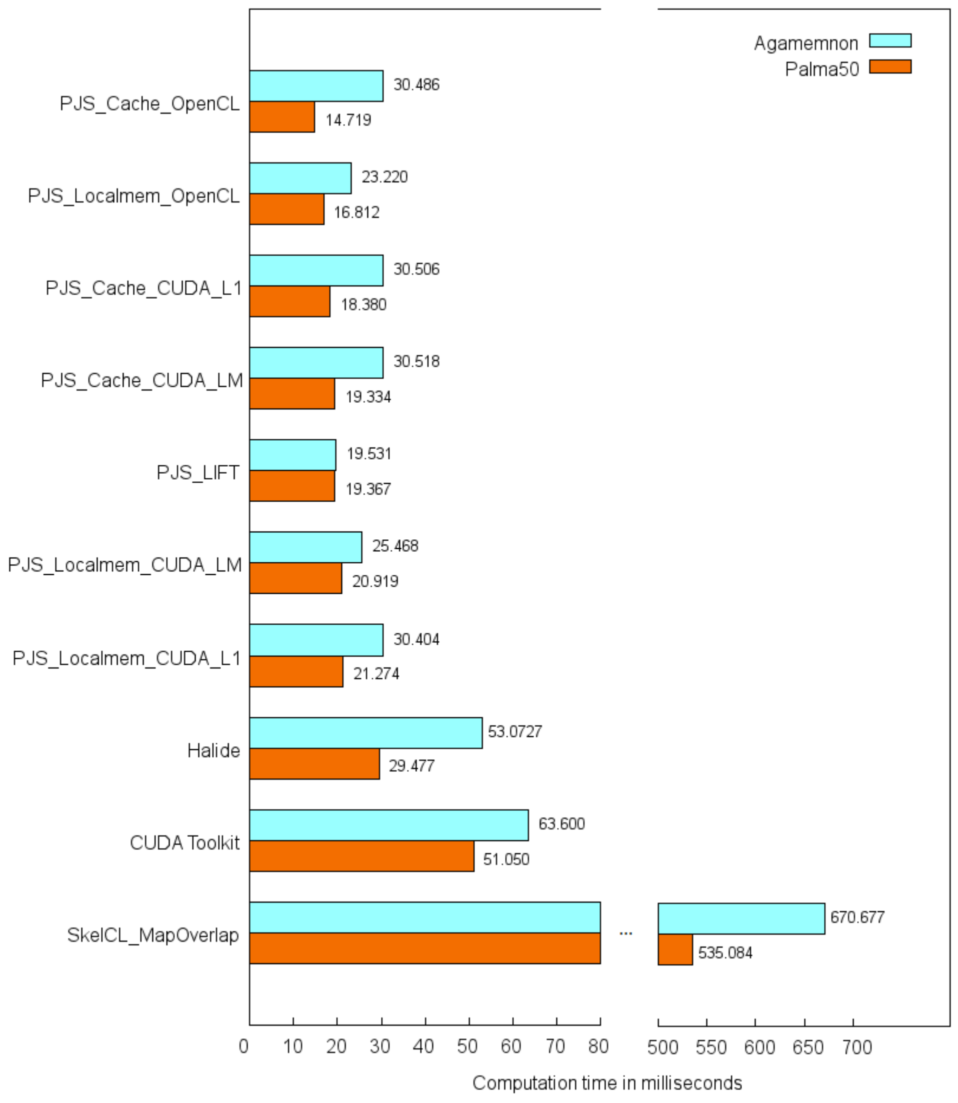


Figure 4.7: Box blur GPU kernel runtimes for a 8192 × 8192 input matrix and 15 × 15 mask. Note that despite using a newer GPU, *Agamemnon* runtimes are slower than the ones on *Palma50*. We attest this to automatic global memory caching being unavailable on the Tesla K20c.

The staggeringly high runtime of the *SkelCL* kernel is explained in section 4.3.1: Inconvenient work load divisions that prevent other work items from starting their calculations make it the worst kernel in the test run.

The official CUDA box blur example performs better, but is still slow overall. The reason for this is found easily: While the kernel is fairly optimized and uses separation, texture memory and local memory, it employs static parameters for the number of work items per work group and the number of work groups regardless of what GPU it is running on. While the assumed parameters might be ideal for the specific GPU the example was written for, the same can't be said about our GPUs, leading to a significantly worse results than the ones the kernels which were auto-tuned achieved.

Halide does perform better than the CUDA example - however Halide was originally intended for parallel programs running on CPUs. Halide's compiler's optimizations seem to be aimed more at the CPU use case. (**TODO:** The current speculation entails that Halide is making use of architecture specific optimizations the OpenCL compiler (GCC) does not and also that LLVM may make more optimizations since it is specifically geared towards CPUs and their inner workings. Specifics here once the Halide developers respond to the email.) [2]

Next up is a reference OpenCL implementation of the box blur kernel that was generated using the LIFT framework and is subsequently called "PJS_LIFT".The framework allows writing kernels in the Scala language and then automatically generating OpenCL C kernels as described in [3]. The generated kernel is hardly human-readable and is generated for a specific parameter set. If other parameters are chosen, the kernel is not guaranteed to work fast or to work at all. The kernel performs almost as well as the OpenCL versions of the following kernels. **TODO: Where exactly do the parameters come from? Separation of kernels?**
The *PJS_Localmem* and *PJS_Cache* CUDA kernels were assumed to perform the same in regards to execution speed because while they use different approaches, both kernels use the physical shared memory segment of the GPU, as both L1 cache and local memory are located there.
Interestingly, in all cases the shared memory configuration makes no large difference on *Palma50*'s Fermi GPU. This is because larger local memory is not necessarily useful: Although 48KB can be used with the appropriate configuration, the kernel with the optimal parameters doesn't take up that much memory because the amount of used local memory determines how many work groups can be executed in parallel, as local memory is shared among all work groups on a GPU computation unit. If the used amount is high, only few work groups can run, forcing the other work groups to idle.

On *Agamemnon* on the other hand, the *PJS_Localmem_LM* kernel benefits from the additional local memory, while the *PJS_Cache* kernels remain unaffected due to due L1 caching being disabled on the Tesla K20c GPU.

The OpenCL variants of both local memory and cache kernels performed faster than their OpenCL counterparts overall, despite being exact translations of another.

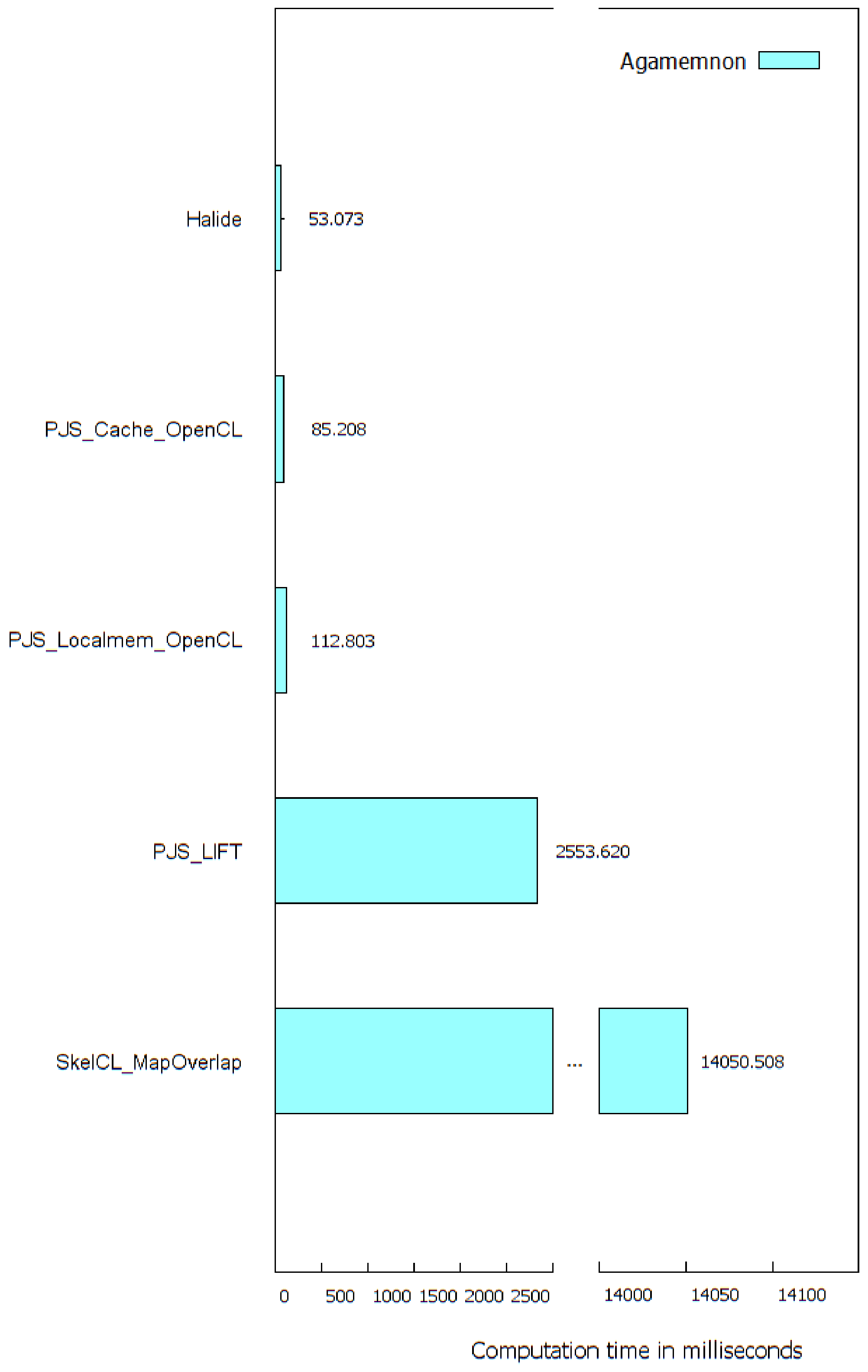


Figure 4.8: Box blur CPU kernel runtimes for a 8192×8192 input matrix and 15×15 mask.

Figure 4.8 shows the OpenCL CPU runtimes of all kernels save for the CUDA example which only runs on NVIDIA GPUs.

TODO: Bastian-Kernel on CPU here.

SkelCL, again the kernel with the longest runtime, was never advertised to work well on CPUs and in fact just uses its GPU kernel version for CPU runs, which amplifies the aforementioned detrimental effects.

On the CPU, *PJS_Localmem* and *PJS_Cache* differ in their execution times significantly: *PJS_Cache* was $\approx 25\%$ faster than *PJS_Localmem*. The reason for this is that *PJS_Localmem* copies values into local memory in the expectation that querying those values later will be faster than loading them from global memory - but on the CPU, the local memory is mapped to RAM, which is the same as global memory. Therefore, copying the values provides no advantage and just slows down the kernel in comparison to the *PJS_Cache* kernel which relies on global memory caching only (see section **??**).

Halide, on the other hand, now performs best, because of its explicit CPU optimization (**TODO**: Also wait for email response here before continuing.) [2]

4.4.3 Runtime Impact of Out-of-Bounds Handling

(**Phil:**)
All of the results in figures 4.7 and 4.8 for kernels which manage local memory explicitly were achieved by employing programs that neglected out-of-bounds handling on purpose by assuming a larger input matrix while only calculating blocks of values whose neighborhood was guaranteed to completely lie within the bounds of the matrix. This was done to allow the kernels to remain relatively simple because otherwise, blocks on the edge of a work group would need special treatment to coordinate loading neighborhood values into local memory, while also taking care of using neutral values in the case of blocks that lie not only on the edge of a work group, but also on the edge of the input matrix.
However, we were interested in the runtime advantage of not using out-of-bounds handling in our kernels, and thus compared the execution times of "*PJS_Localmem*" with those of "*PJS_Localmem_OOB*". Surprisingly, the kernel that used out-of-bounds handling performed better than the one that did not, despite it having more instructions overall (see Figure 4.9).

The reason for this counter-intuitive result is that when out-of-bounds handling is eliminated, every block in a work group has to be handled the same way; it is no longer clear if the block is on the edge of a work group or an interior block. Because of this circumstance, every block needs to load its entire neighborhood into local memory, which causes numerous loads that overlap

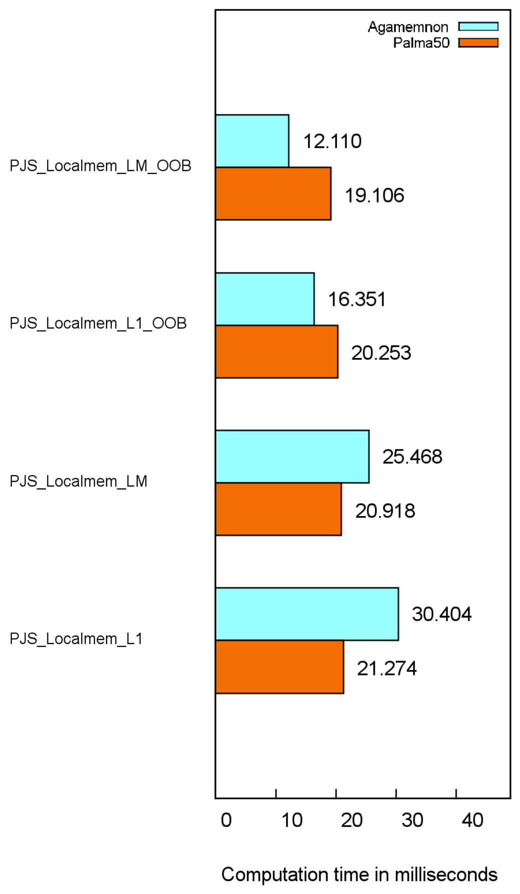


Figure 4.9: Runtimes of explicit local memory management kernels with and without out-of-bounds handling. Note that the normal kernels perform slower than their OOB counterparts.

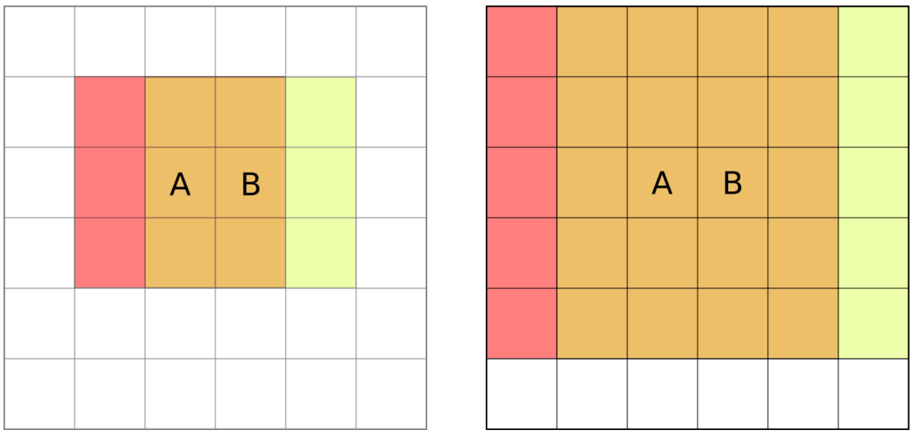


Figure 4.10: Effects of neglecting out-of-bounds handling in the PJS_Localmem kernel variations.

Left: Two values, A and B, and their respective neighborhoods in red and yellow with a mask size of 3×3.
Right: The same values but with a mask size of 5×5 instead.
The overlapping values (orange) are loaded twice. In the case of the 3×3 mask, six values are overlapping, whereas with a 5×5 mask, their number increases to 20.

each other and are therefore unnecessary because each value only has to be loaded once, whereas when using out-of-bounds handling, the border blocks have to load less values than interior blocks. The larger the dimension of the kernel mask, the worse this effect becomes as the overlapping area increases (see Figure 4.10).

Hence, neglecting out-of-bounds handling only proves beneficial for kernel runtimes when the kernel mask is relatively small, so the effect of overlapping loads does not overshadow the advantage of not having to check the position of the current block.

Interestingly, the effect of multiple, overlapping loads on the kernel runtime is weaker on the *Palma50* platform. Most likely this is because Fermi GPUs have global memory caching enabled, allowing the loading operations for the "explicit caching" to be done faster by using implicit caching.
We then decided to test this effect using the Laplace kernel, enabling out-of-bounds checking for the kernels that use local memory. As the Laplace kernel is inseparable, this also granted us higher runtimes overall, leading to clearer differences in speed.

Surprisingly, even though both local memory and L1 cache are both located in shared memory, explicitly managing local memory is faster by $\approx 25 - 50\%$ when compared to automatic caching, depending on the chosen system (see Figure 4.11).

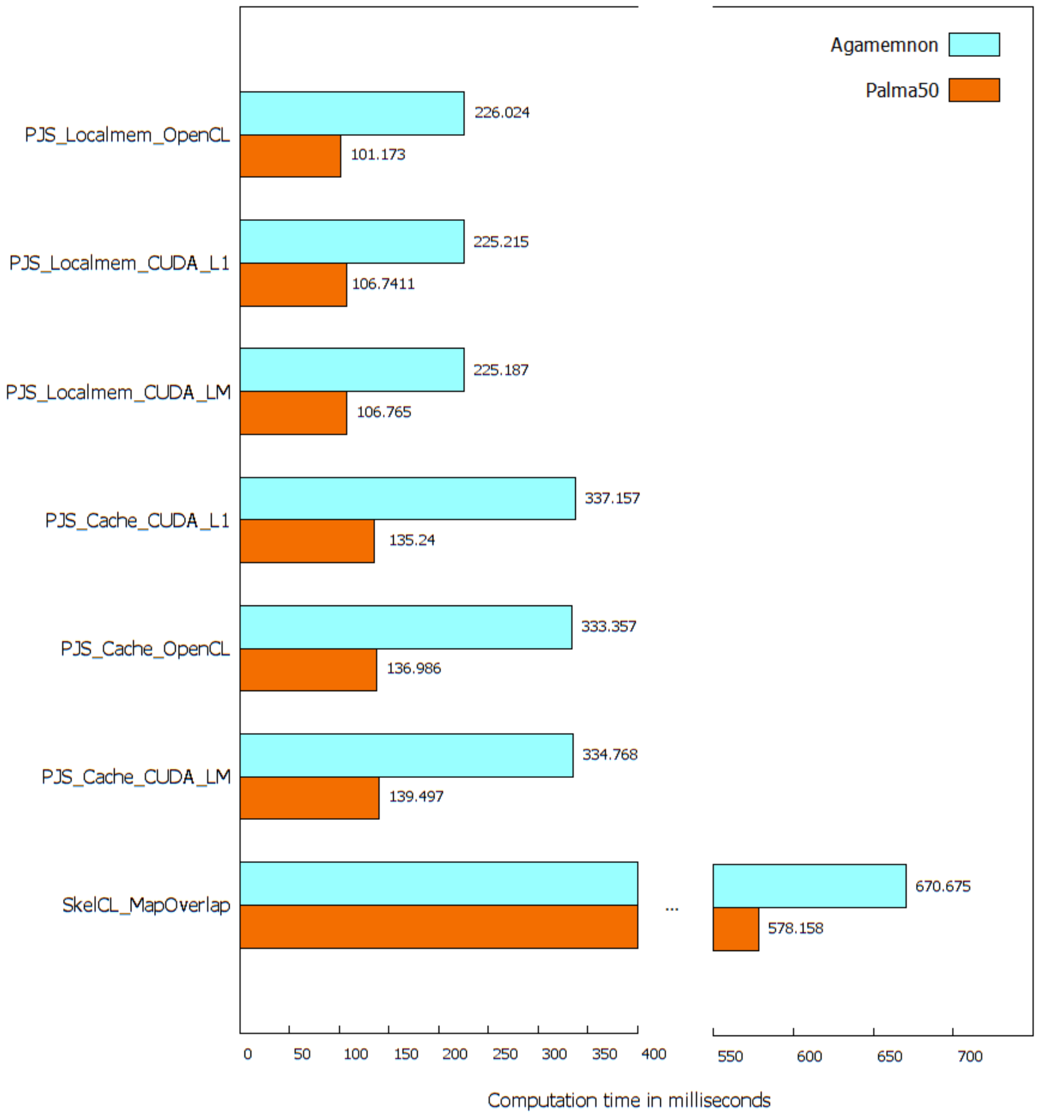


Figure 4.11: Laplace GPU kernel runtimes for a 8192×8192 input matrix and 15×15 mask, with $\sigma = 1$. Local memory kernels use out-of-bounds handling.

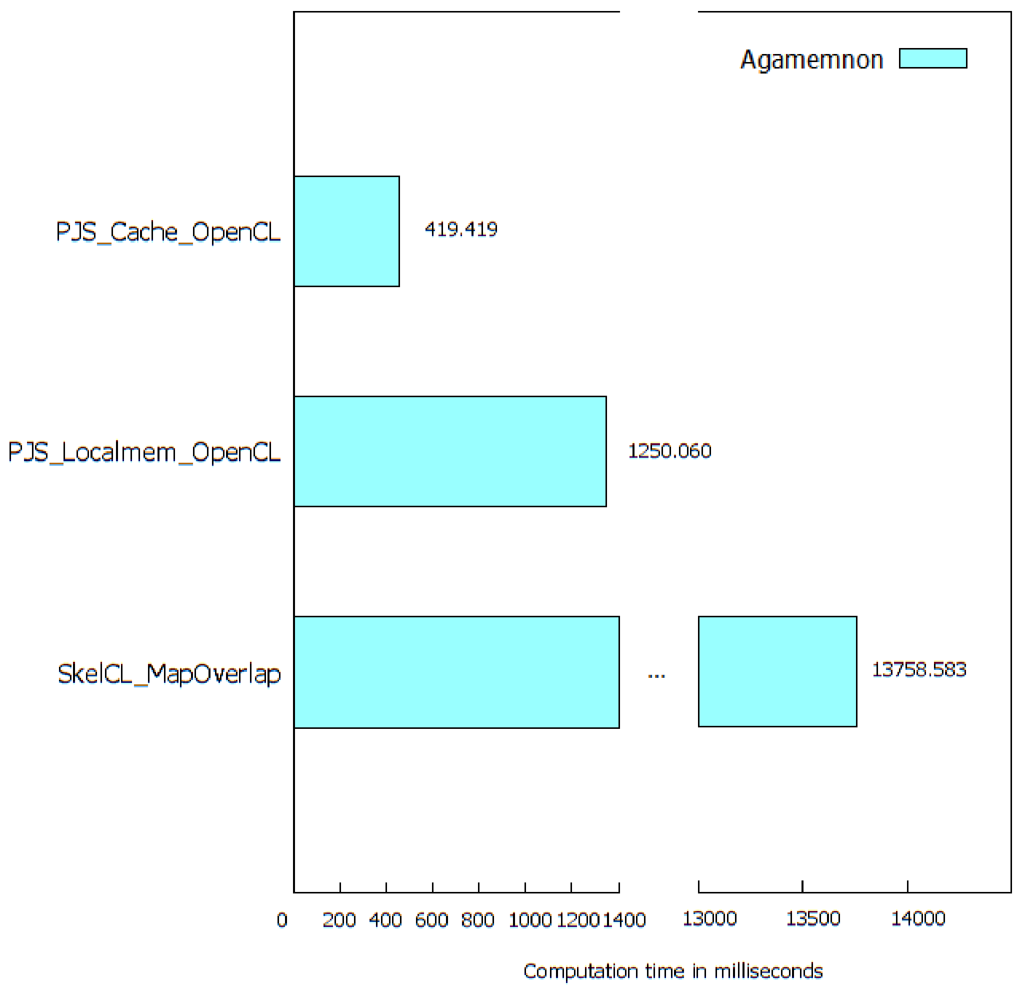


Figure 4.12: Laplace CPU kernel runtimes for a 8192 × 8192 input matrix and 15 × 15 mask, with $\sigma = 1$. Local memory kernels use out-of-bounds handling.

A possible explanation for this behavior is proposed in [1]: Depending on the caching algorithm and the access pattern, the order in which the algorithm accesses the input values, values might be evicted from the cache prematurely, causing cache misses and subsequently higher runtimes.

On CPUs, the effects are the same as for the box blur kernel. Using local memory introduces unnecessary copying because local memory is mapped to RAM, making caching much faster.

5 Conclusion

Phil:

Based on the results of the previous chapter, it is evident that auto-tuning our generic stencil kernels with dynamic parameters leads to huge performance gains compared to their static alternatives: Our tuned box blur kernel for instance ran $\approx 50\%$ faster than the CUDA Toolkit example. This improvement could be seen on both test platforms as the tuning parameters were separately found for each of them, and is expected to be achievable with all kinds of kernels, becoming more useful the longer and more complex kernels become as it would be cumbersome to tune them manually.

Therefore, we have come to the conclusion that writing platform-independent kernels and then auto-tuning them is a time-saving means to achieve maximum speed on any device when compared to the time one would need to spend to manually tune a kernel, which requires a lot of time and detailed knowledge about the hardware the kernel is supposed to run on. Especially after the tuning library implementation, tuning became doable in a relatively short amount of time even for large parameter spaces.

It remains questionable if OpenTuner's "intelligent" searching strategies net an ideal speedup if the tuning time is limited, although the achieved speedups were already very high. Halide optionally can use an alternative, built-in tuner for finding schedules that compile to fast code, employing a genetic and approximative algorithm [2]. The results weren't as good as when we tuned the kernel ourselves, but delivered a reasonably fast schedule in much shorter time than our tuning application did.

The OpenCL and CUDA run times of the same kernels, translated from OpenCL to CUDA one-to-one, initially expected to be the same, actually differ by quite an amount. Finding out why this is is still subject to testing and remains an issue to solve in future work.

Overall, manually written OpenCL / CUDA kernels such as the *PJS_Cache* and *PJS_Localmem* variants performed the fastest after having been tuned, but took a lot of time to implement, analyze and debug. If one is willing to sacrifice some performance, a (self-)tuned Halide kernel will perform almost as fast while being much easier to implement.

6 Addendum

6.1 Auto-Tuning Framework guide

The following section describes the installation and usage of the Auto-Tuning Framework.

6.1.1 Requirements

- OS: Linux
- libreadline-dev
- Python 2.7+ with the following packages installed:
  - argparse>=1.2.1
  - fn>=0.2.12
  - numpy>=1.8.0
  - pysqlite>=2.6.3
  - SQLAlchemy>=0.8.2

6.1.2 Installation

Checkout the git branch ”ATFramework” from the link:
git@IVV5GITHUB01.uni-muenster.de:a_rasc01/pjs_ss_16.git

Script

1. Execute the ”install” file found in your ATFramework folder. Make sure you have installed Python beforehand. You should get the message ”AT-Framework was successfully installed”.

Manual

1. Install Lua:
   a) in directory ”ATFramework/lua-5.3.1” execute ”make linux”
   b) check correct build with ”make test”; you should get ”Lua 5.3.1 Copyright (C) 1994-2015 Lua.org, PUC-Rio”

c) install locally with "make local"; you should get "make[1]: Leaving directory "/path/to/lua-5.3.1"

d) visit http://www.lua.org for troubleshooting

e) If nothing went wrong Lua has been installed correctly

2. If Lua has been successfully installed:

   a) Execute the commands "python-config –cflags" and "python-config –ldflags" and append the outputs to the second line of the makefile in your ATFramework folder.

   b) Run "make" in the directory "ATFramework".

6.1.3 Modifying your program

1. Set up pragmas in your source code:
   #pragma tune TP in TYPE {RANGE}(INCREMENT*); DEFAULT*; PREDICATE; ORDER;

TP	Name of the parameter you want to tune. Dependent parameters have to be declared in one pragma. Independent parameters should be declared in separate pragmas.
TYPE	The type of the parameter. Valid types: int, char, double, short, float, string
RANGE	The range of values for the parameter.
INCREMENT (optional)	Defines the step size of the evaluation of the parameter interval. Mostly used in combination with float parameters.
DEFAULT	Sets the default value of the parameter which will be used during the evaluation of other parameters. If nothing is set the first value of RANGE will be used. default(TP) = DEFAULTVALUE
PREDICATE	Side effect free boolean expressions that only contain tuning parameters as variables. Only true-evaluated configurations will be added to the search space.
ORDER	Sets the tuning order of the parameters.

   More information about pragmas in the diploma thesis: "Implementing a Generic Auto-Tuning Framework for OpenCL" from Markus Damerau.

2. To be able to tune your program it has to write the measured tuning values of one run into the second line of a csv file. The first line has to be used for labeling. For example:

Listing 6.1: Writing the measured values to a csv file

```
#include <fstream>
std::ofstream file("results.csv");
file << "Kernel time" << ";" << "Memory usage"        << "
    ;" << ... << std::endl;
file << measured_time << ";" << measured_memory_usage << "
    ;" << ...;
file.close();
```

This will write the measured tuning values into a file called ”results.csv”. You can use the ”tuning-objective” flag to select the metric whose values will be tuned. The default is to tune according to the first metric written into the file.

6.1.4 Running the ATF

1. Go into the directory of the program you want to tune.

2. Create a new file named ”config_AT” with the following content:

Listing 6.2: Configuration file for the ATF

```
COMPILER_CALL=/*call to compiler with absolute paths to
    source files*/
EXECUTABLE=/*absolute path to executable file*/
FILES=/*relative paths to the files with pragmas*/
```

Example: For example, if you want to tune the program main.cpp located in ”home/user_name/example/main.cpp” your ”config_AT” file would look like this:

Listing 6.3: Configuration file for the ATF

```
COMPILER_CALL=g++ /home/user_name/example/main.cpp -o /
    home/user_name/example/tmp.bin
EXECUTABLE=/home/user_name/example/tmp.bin
FILES=main.cpp
```

3. Run ATF with /path/to/atframework /path/to/folder-with-config_AT/ [-flag=value]

   If value contains blank spaces, it has to be surrounded with quotation marks.

   Flags:

-reps=	Determines the number of repetitions for the benchmarks, default is 3.
-tuning-objective=	The index of the metric to read from the csv file.
-results-file=	The name and path of the csv file explained in 2.2, default name is ”results.csv”. If you are using the exhaustive algorithm you have to specify the runtime in microseconds.
-abort-on-compile-error=	If submitted, the tuning process aborts on compiling errors. If not submitted, compiling errors result in maximum runtime for the tested configuration.
-abort-on-runtime-error=	If submitted, the tuning process aborts on runtime errors. If not submitted, runtime errors result in maximum runtime for the tested configuration.

Flags per order group:

-algorithm-args=	Further flags specific for the search algorithm. If you are using OpenTuner for example you can use: -algorithm-args=”--improvement-threshold 600 1.1” This will stop the tuning run if the speedup during the last 600 seconds was lower than 10%. More flags for OpenTuner can be found at ”opentuner.org”.
-stop-after=	The maximal tuning time in seconds, default is 30.
-search=	Name of search algorithm to be employed, default is ”opentuner”.

You can use these flags per order group. If you submit only one value for one of these flags it will be used for all order groups. If you submit multiple values the number of order groups and values per flag have to match. Multiple values have to be separated by a comma. You can change the separator by using the flag ”-separator=”.

For example if you have two order groups:

/path/to/atframework /path/to/folder-with-config_AT/ -search=opentuner -stop-after=180,20 -separator=; -algorithm-args=”--improvement-threshold 600 1.1”;--no-dups

This will use the OpenTuner for both order groups, stops the tuning for the first order group after 180 seconds and the tuning for the second order group after 20 seconds. Now the separator gets changed to ”;” for the following flags. The first set of algorithm arguments written in quotation marks will be used for the first order group, the ”--no-dups” will be used for the second one.

6.2 Auto-Tuning Library guide

The following section describes the installation and usage of the Auto-Tuning Library.

6.2.1 Requirements

These requirements are only necessary if you are using the OpenTuner class. If you do not have the rights to install these packages globally, you can install them in a local directory and add the directory to the environment variable `PYTHONPATH`.

- Python 2.7+ with the following packages installed:
  - opentuner>=0.5.1
  - argparse>=1.2.1
  - fn>=0.2.12
  - numpy>=1.8.0
  - pysqlite>=2.6.3
  - SQLAlchemy>=0.8.2

6.2.2 Installation

Checkout the git branch ”atf_library” from the link:
git@IVV5GITHUB01.uni-muenster.de:a_rasc01/pjs_ss_16.git

6.2.3 Modifying your program

You have to take the following steps to tune your program with the ATF library:

1. Include the `ATF_Library.h` header file.
2. Define tuning parameter blocks:

   ```
   // TP blocks
   atf::TP_Block<1> tp_block_a, tp_block_b;
   tp_block_a.insert<atf::tp_int>( "A_int", i_1, i_n );
   tp_block_b.insert<atf::tp_float>( "B_float", f_1, f_m );
   ```

   Dependent tuning parameters will be put into the same block. The template parameter `1` specifies the amount of tuning parameters that will be added to the block.
3. Apply constraints to the parameters:

```
// predicates
tp_block_a.apply_constraint( []( int A_int ){
    return A_int % 2 == 0; } );
tp_block_b.apply_constraint( []( float B_float ){
    return ceil(B_float) % 4 == 0 ; } );
```

4. Invoke the tuning process:

```
atf::AbortCondition cond;
atf::Tuner tuner(cond) ;
auto best_config = tuner.tune(func,
        {tp_block_a.get_configurations(),
         tp_block_b.get_configurations()} );
```

In this listing the abstract classes `AbortCondition` and `Tuner` are used. Make sure to substitute them with a concrete subclass in your program. The available abort conditions are:

- `TestCountCondition`, a condition that is used to control how many configurations are going to be tested by the tuner before stopping. Once the number of tested configurations reaches the counter set in this condition the tuning run will stop. Even configurations that lead to errors by the kernel count towards this counter.
- `TimeCondition`, this condition controls how long a tuning run will take. Once the timer set in this condition is reached the `checkCondition` method returns true and the tuning will stop.
- `ResultCondition` will tune until a specified result, for example a certain runtime, is reached.
- `SpeedUpCondition` will stop the tuning if a certain speed up is not reached. This class can be instantiated by using one of the following constructors. The tuning will either stop if the submitted speed up is not reached after a certain amount of configurations tested, or if the time submitted is passed before the speed up is reached.
- `AndCondition` is a composite condition that is used to combine two or more of the conditions described above.
- `OrCondition` is equal to `AndCondition` with the exception that one condition has to return `true` in order for the tuning run to stop.

6.3 Reading information from the OpenTuner database

Richard:
The OpenTuner saves all the relevant tuning data in a database. If not con-

figured otherwise, this database will be created in the working directory in a subfolder called `opentuner.db`. Being able to read information from this database is very useful when more detailed information about a tuning run is needed. Because the OpenTuner was integrated in the ATF and ATF library, this section is applicable not only to tuning runs done by the OpenTuner, but also to those done by the ATF or ATF library.

The OpenTuner database is a sqlite3[1] database, that can be accessed with the sqlite3 executable. It is possible to access virtually all information about the tuning runs contained in that database manually via SQL queries. However, because the configuration data is stored in a compressed format, reading configurations is not possible by query. To read single configurations and also a few other useful statistics of a tuning run, a Python script was developed.

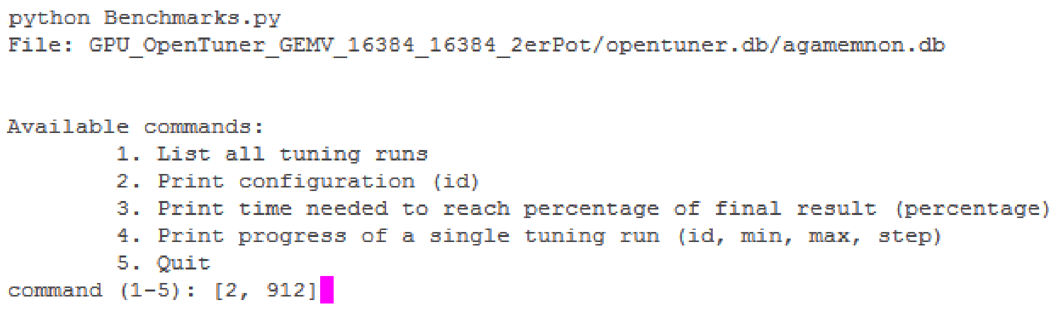
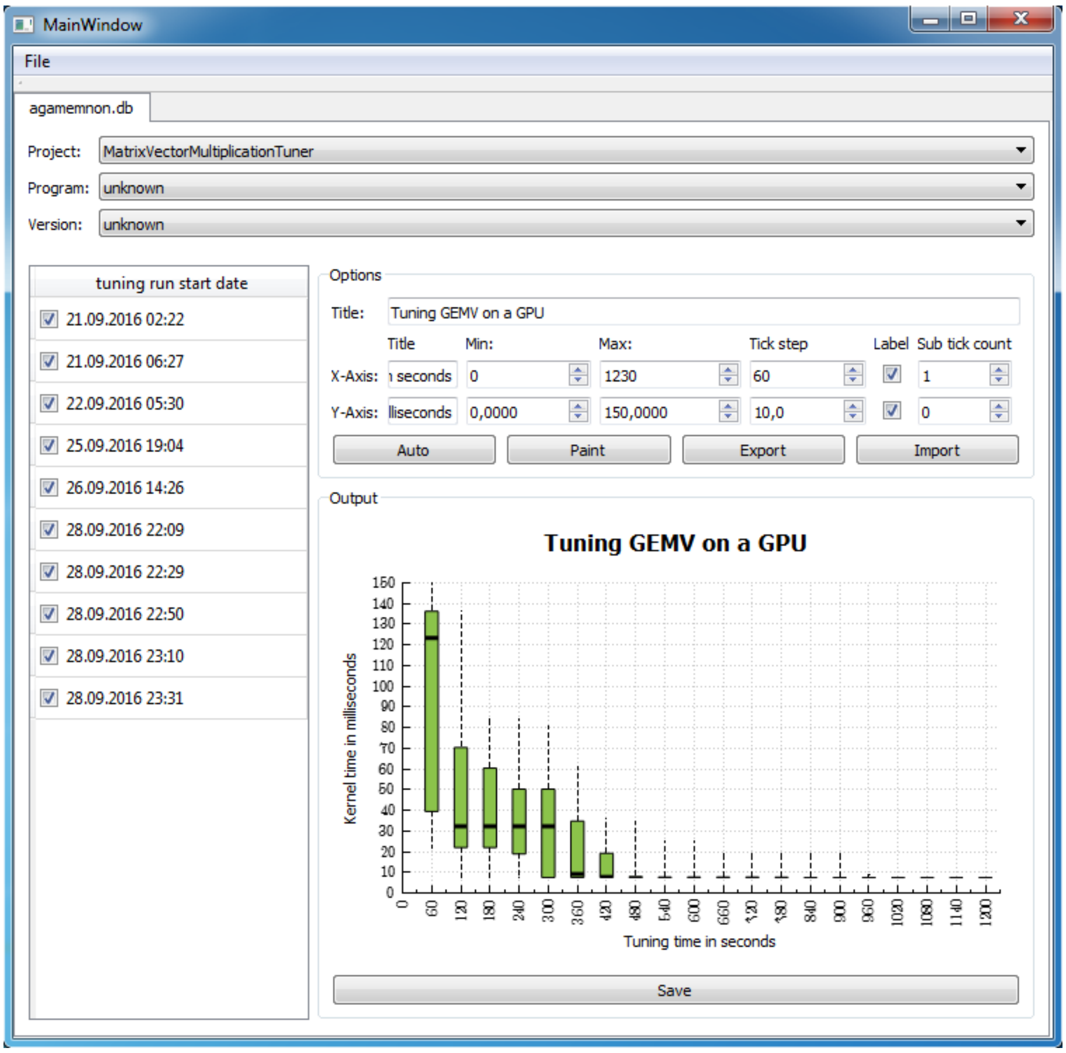

```
python Benchmarks.py
File: GPU_OpenTuner_GEMV_16384_16384_2erPot/opentuner.db/agamemnon.db


Available commands:
        1. List all tuning runs
        2. Print configuration (id)
        3. Print time needed to reach percentage of final result (percentage)
        4. Print progress of a single tuning run (id, min, max, step)
        5. Quit
command (1-5): [2, 912]
```

Figure 6.1: Reading a single configuration from an OpenTuner database

The script needs the OpenTuner package installed to access the database and can be started without any command line parameters. It first prompts the user for the path to the database to read from. After that the main menu is shown (figure 6.1). The user can select any of the menu items by typing the number and pressing `Enter`. For menu items that need additional parameters, such as the second item for example, the item number and arguments have to be enclosed in brackets. Entering [`2, 912`] for example would print the configuration with the id 912.

Additionally an application for creating graphs was created. It is based on the Qt framework and is able to display a single tuning run or a list of tuning runs as a box plot with the 25th and 75th percentile. When displaying multiple tuning runs at once, all of them have to be contained in the same database. Figure 6.2 shows the interface of the plotting application when a database was opened via the `File` menu. When plotting a single run, the best runtime is queried at every major and minor tick of the x-axis. When plotting multiple runs as a box plot, a box plot is created for every major tick of the x-axis.
Tuning runs can be exported as a text file to use the extracted data in other applications. Exported box plots can also be imported again into the plotting application. This will lead to graphs like the ones shown in figure 3.9, display-

[1]https://www.sqlite.org/



Project: MatrixVectorMultiplicationTuner

Program: unknown

Version: unknown

tuning run start date
✓ 21.09.2016 02:22
✓ 21.09.2016 06:27
✓ 22.09.2016 05:30
✓ 25.09.2016 19:04
✓ 26.09.2016 14:26
✓ 28.09.2016 22:09
✓ 28.09.2016 22:29
✓ 28.09.2016 22:50
✓ 28.09.2016 23:10
✓ 28.09.2016 23:31

Title: Tuning GEMV on a GPU

	Title	Min:	Max:	Tick step	Label	Sub tick count
X-Axis:	ı seconds	0	1230	60	✓	1
Y-Axis:	lliseconds	0,0000	150,0000	10,0	✓	0

Figure 6.2: Plotting multiple runs from an OpenTuner database

ing a secondary series of box plots for comparison. The `Save` button will save the currently plotted graph as an image file.

6.4 Auto-tuning results

(**Phil:**)

This section shows the optimal parameters found by the auto-tuning library (see section 3.4) for each tuneable implementation and graphical representations of the running times. “Optimal” means that the program ran in the shortest time with the given configuration in respect to all other configurations.

The parameters for the PJS_Localmem kernel variations are as follows:

NUM_WI_X/Y refers to the number of work items in a work group in the X- or Y-dimensions, NUM_WG_X/Y is the number of work groups in the X- or Y-dimensions and X/Y-BLOCKHEIGHT and -BLOCKWIDTH refer to the size of the blocks of values a work item works on. The parameters for the PJS_Cache kernels are named analogously.

The Halide parameters are explained in section 2.4.1.

Explicitly separated kernels have a prefix for their parameters indicating which kernel dimension they are meant for. "Y_NUM_WI_X" for instance refers to the number of work items in a work group in the x-dimension of the y-dimension kernel. Determining two separate sets of parameters is necessary because a kernel exhibits different memory access patterns depending on the order in which the input values are processed. Y-dimension kernels "jump around" in memory because the C programming language stores arrays in row-major order, whereas X-dimension kernels' memory accesses can be coalesced because they are close to each other in memory. Hence, different parameters can be optimal for kernels that work in different dimensions.

	Y_NUM_WI_Y	Y_NUM_WI_X	Y_BLOCKHEIGHT	Y_BLOCKWIDTH	X_NUM_WI_Y	X_NUM_WI_X	X_BLOCKHEIGHT	X_BLOCKWIDTH
Agamemnon								
PJS_Localmem_OOB_LM	4	32	8	1	2	64	2	4
PJS_Localmem_OOB_L1	16	32	1	1	2	128	1	2
PJS_Localmem_LM	4	32	8	1	2	64	2	4
PJS_Localmem_L1	2	256	2	1	2	64	1	4
Palma50								
PJS_Localmem_OOB_LM	4	32	8	1	2	64	2	2
PJS_Localmem_OOB_L1	8	32	2	1	2	64	2	2
PJS_Localmem_LM	4	32	8	1	8	32	1	4
PJS_Localmem_L1	8	32	2	1	2	64	1	4

Table 6.1: PJS_Localmem CUDA box blur GPU parameters.

	Y_NUM_WI_Y	Y_NUM_WI_X	Y_BLOCKHEIGHT	Y_BLOCKWIDTH	X_NUM_WI_Y	X_NUM_WI_X	X_BLOCKHEIGHT	X_BLOCKWIDTH
Agamemnon								
PJS_Localmem	2	32	8	1	2	32	2	4
Palma50								
PJS_Localmem	2	32	16	1	8	16	1	4

Table 6.2: PJS_Localmem OpenCL box blur GPU parameters.

	Y_NUM_WG_Y	Y_NUM_WG_X	Y_NUM_WI_Y	Y_NUM_WI_X	Y_CACHE_BLOCK_Y	Y_CACHE_BLOCK_X	X_NUM_WG_Y	X_NUM_WG_X	X_NUM_WI_Y	X_NUM_WI_X	X_CACHE_BLOCK_Y	X_CACHE_BLOCK_X
Agamemnon												
PJS_Cache_LM	4096	16	1	128	2	2	4096	64	2	128	1	1
PJS_Cache_L1	8192	4	1	128	1	4	4096	64	2	128	1	1
Palma50												
PJS_Cache_LM	64	64	16	64	1	2	2048	1	1	256	4	16
PJS_Cache_L1	16	128	8	64	2	1	1024	8	4	32	2	16

Table 6.3: PJS_Cache CUDA box blur GPU parameters.

	Y_NUM_WG_Y	Y_NUM_WG_X	Y_NUM_WI_Y	Y_NUM_WI_X	Y_CACHE_BLOCK_Y	Y_CACHE_BLOCK_X	X_NUM_WG_Y	X_NUM_WG_X	X_NUM_WI_Y	X_NUM_WI_X	X_CACHE_BLOCK_Y	X_CACHE_BLOCK_X
Agamemnon												
PJS_Cache	8192	16	1	128	1	4	4096	64	2	128	1	1
Palma50												
PJS_Cache	32	64	16	64	16	2	128	16	4	64	8	8

Table 6.4: PJS_Cache OpenCL box blur GPU parameters.

	GPU_WORKGROUP_X	GPU_WORKGROUP_Y	GPU_WORKITEM_X	GPU_WORKITEM_Y
Palma50				
Halide	128	64	64	4

Table 6.5: Halide box blur GPU parameters.

	Y_NUM_WI_Y	Y_NUM_WI_X	Y_BLOCKHEIGHT	Y_BLOCKWIDTH	X_NUM_WI_Y	X_NUM_WI_X	X_BLOCKHEIGHT	X_BLOCKWIDTH
Agamemnon								
PJS_Localmem_OOB_CPU	4	16	8	1	2	32	4	1
PJS_Localmem_CPU	2	128	4	1	2	16	16	1

Table 6.6: PJS_Localmem box blur CPU parameters.

	Y_NUM_WG_Y	Y_NUM_WG_X	Y_NUM_WI_Y	Y_NUM_WI_X	Y_CACHE_BLOCK_Y	Y_CACHE_BLOCK_X	X_NUM_WG_Y	X_NUM_WG_X	X_NUM_WI_Y	X_NUM_WI_X	X_CACHE_BLOCK_Y	X_CACHE_BLOCK_X
Agamemnon												
PJS_Cache_CPU	256	32	32	4	1	16	128	1	16	4	4	32

Table 6.7: PJS_Cache box blur CPU parameters.

	CPU_VECTORIZE_1	CPU_VECTORIZE_2	CPU_TILES_X1	CPU_TILES_X2	CPU_TILES_Y1	CPU_TILES_Y2
Agamemnon						
Halide_CPU	8	8	128	16	32	1

Table 6.8: Halide box blur CPU parameters.

	NUM_WI_Y	NUM_WI_X	BLOCKHEIGHT	BLOCKWIDTH
Agamemnon				
PJS_Localmem_OOB_LM	4	256	1	1
PJS_Localmem_OOB_L1	4	256	1	1
Palma50				
PJS_Localmem_OOB_LM	2	256	2	1
PJS_Localmem_OOB_L1	2	256	2	1

Table 6.9: PJS_Localmem CUDA GPU Laplace parameters.

	NUM_WI_Y	NUM_WI_X	BLOCKHEIGHT	BLOCKWIDTH
Agamemnon				
PJS_Localmem_OOB	8	64	1	2
Palma50				
PJS_Localmem_OOB	2	64	8	1

Table 6.10: PJS_Localmem OpenCL GPU Laplace parameters.

	NUM_WG_Y	NUM_WG_X	NUM_WI_Y	NUM_WI_X	CACHE_BLOCK_Y	CACHE_BLOCK_X
Agamemnon						
PJS_Cache_LM	128	8	8	128	2	2
PJS_Cache_L1	256	1	4	256	2	8
Palma50						
PJS_Cache_LM	1024	128	8	128	1	1
PJS_Cache_L1	32	64	4	128	1	1

Table 6.11: PJS_Cache CUDA GPU Laplace parameters.

	NUM_WG_Y	NUM_WG_X	NUM_WI_Y	NUM_WI_X	CACHE_BLOCK_Y	CACHE_BLOCK_X
Agamemnon						
PJS_Cache	512	2	4	256	2	4
Palma50						
PJS_Cache	32	64	8	128	32	1

Table 6.12: PJS_Cache OpenCL GPU Laplace parameters.

	NUM_WI_Y	NUM_WI_X	BLOCKHEIGHT	BLOCKWIDTH
Agamemnon				
PJS_Localmem_OOB_CPU	2	8	64	1

Table 6.13: PJS_Localmem Laplace CPU parameters.

	NUM_WG_Y	NUM_WG_X	NUM_WI_Y	NUM_WI_X	CACHE_BLOCK_Y	CACHE_BLOCK_X
Agamemnon						
PJS_Cache_CPU	2048	128	1	64	4	1

Table 6.14: PJS_Cache Laplace CPU parameters.

6.5 List of kernel implementations

6.5.1 OpenCL

Naive

boxblur_naive.cpp

Blocking

altern_host.cpp

boxblur_blocking.cpp

Blocking-Cache

OpenCL_Ari_Cache.cpp

OpenCL_Ari_Cache_sep_CPU_x/y.cpp

OpenCL_Ari_Cache_sep_x/y.cpp

PJS_Localmem

OpenCL_PJS_Localmem.cpp

OpenCL_PJS_Localmem_noOOB.cpp

OpenCL_PJS_Localmem_CPU.cpp

OpenCL_PJS_Localmem_LAPLACE.cpp

OpenCL_PJS_Localmem_LAPLACE_CPU.cpp

PJS_Cache

OpenCL_PJS_Cache.cpp

OpenCL_PJS_Cache_CPU.cpp

OpenCL_PJS_Cache_LAPLACE.cpp

OpenCL_PJS_Cache_LAPLACE_CPU.cpp

PJS_LIFT

OpenCL_PJS_LIFT.cpp

6.5.2 CUDA

PJS_Localmem

CUDA_PJS_Localmem_L1.cu

CUDA_PJS_Localmem_LM.cu

CUDA_PJS_Localmem_OOB_L1.cu

CUDA_PJS_Localmem_OOB_LM.cu

CUDA_PJS_Localmem_LAPLACE_L1.cu

CUDA_PJS_Localmem_LAPLACE_LM.cu

PJS_Cache

CUDA_PJS_Cache_L1.cu

CUDA_PJS_Cache_LM.cu

CUDA_PJS_Cache_LAPLACE_L1.cu

CUDA_PJS_Cache_LAPLACE_LM.cu

CUDA-Example

CUDA_example.cu + boxFilter_gold.cpp

6.5.3 Halide

6.5.4 SkelCL

List of Figures

List of Tables

Bibliography

[1] C. Li et. al. **Understanding the Tradeoffs between Software- Managed vs. Hardware-Managed Caches in GPUs**. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[2] J. Ragan-Kelley et. al. **Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines**. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 519–530, 2013.

[3] M. Steuwer et. al. **Generating Performance Portable Code using Rewrite Rules**. 2015.