UPPSALA
UNIVERSITET

Auto-testing efficiency of signal processing algorithms on embedded GPUs

Olle Frisberg
Gunnlaugur Geirsson
Vilhelm Urpi Hedbjörk

Project in Computational Science: Report

January 2019

Institutionen för informationsteknologi

Abstract

When comparing GPU and CPU performance in various applications it is common to look at speedup as the single defining performance metric. Such an approach can be highly subjective and is very much dependent on how optimized the algorithm implementations are on the CPU and GPU. It also does not take into account the energy efficiency of the computation. This project presents an extendable framework to benchmark GPU modules, more specifically efficiency metrics for radar and signal processing related applications, with more objective metrics such as FLOPS, byte throughput and power consumption. Three different algorithms commonly used in signal processing; QR-factorization, bicubic interpolation and pattern matching were implemented and benchmarked on a Jetson Tegra TX1 and TX2 and the performance metrics described above were gathered. The TX2 was about 20% more efficient than the TX1 at pattern matching, and was the most energy efficient in lower power modes.

Contents

Terminology **5**

1 Background **6**
1.1 Project goal 6
1.2 Summary of GPGPU history 6
1.3 The Nvidia Jetson TX1 and TX2 6

2 The CUDA Programming Model **7**
2.1 Thread Hierarchy 7
2.2 Xtream: A CUDA C++ productivity framework 7

3 Auto-testing Framework **8**
3.1 Measurement tools 8
3.1.1 Computation time 8
3.1.2 Power 8
3.1.3 Other metrics 9
3.2 Generating performance metrics 9
3.3 Generating plots 9

4 Benchmarking Algorithms **10**
4.1 Description of the Algorithms 10
4.1.1 QR-factorization 10
4.1.2 Bicubic interpolation 11
4.1.3 Pattern matching 12
4.2 Implementation 12
4.2.1 Porting old work 12
4.2.2 QR-factorization 12
4.2.3 Bicubic interpolation 13
4.2.4 Pattern matching 14

5 Performance Results **15**
5.1 QR-factorization 15
5.2 Bicubic interpolation 18
5.2.1 Varying input size 18
5.2.2 Varying the scale factor 20
5.3 Pattern matching 20

6 Discussion **24**
6.1 QR-factorization 24
6.2 Bicubic interpolation 25
6.2.1 Hermite and B-spline comparison 25
6.2.2 Interpreting the benchmarks 25
6.2.3 Transpose versions 26
6.3 Pattern matching 26

7 Conclusions **27**

Terminology

API - Application Programming Interface. A set of software tools to allow defined communication between various components.

Block - A unit of threads which can share fast memory and can communicate with each other. Multiple blocks make up a grid. A block can threads in three dimensions (X, Y & Z).

CPU - Central Processing Unit.

CSV - Comma Separated Values. Denotes a data file type format with file extension .csv.

CUDA - Parallel computing API provided by Nvidia.

Device - A separate GPU with separate hardware and memory, as considered by CUDA.

FLOPS - Floating point operations per second.

GFLOPS - 10^9 FLOPS.

GPU - Graphics Processing Unit.

GPGPU - General Purpose computing on Graphics Processing Units.

Grid - A unit of blocks.

Host - A separate CPU with separate hardware and memory, as considered by CUDA.

JSON - JavaScript Object Notation.

nvprof - Command line version of the Nvidia Visual Profiler.

Kernel - A set of instructions designated and commenced by the host, but executed on the device. An important aspect of the kernel is that the host continues on its own without waiting for the kernel to finish.

Thread - The smallest unit of the CUDA block/grid abstraction. A thread performs instructions designated by a kernel. Threads are arranged into blocks. 32 threads make up a warp.

TX1/TX2 - Shorthand for the Nvidia Jetson TX1 and TX2 embedded GPU modules.

Warp - 32 threads which all execute the same instruction.

1 Background

As demand for high performance computing is ever-increasing, manufacturers are constantly trying to improve hardware and software in order to increase the computational power and efficiency of processors. The high performance market has historically been dominated by Central Processing Units (CPUs), with many supercomputers consisting of hundreds or thousands of CPUs running in parallel, and connected by high speed communication bands. However, the architecture of CPUs is not inherently parallel, but rather focused on low latency achieved through pipelining and smart cache structures. A modern CPU typically consists of four to eight cores. Graphics Processing Units (GPUs) on the other hand can be composed of thousands of cores, but with a much lower clock frequency than a CPU core. This makes GPUs particularly suitable for highly parallelizable algorithms. The parallel design of the GPU stems from its main purpose: rendering graphics, which requires a large number of very simple and similar computations. The structure of the GPU however can be superior to that of the CPU for tasks other than graphics processing. One such area where the GPU architecture can outperform CPU is in signal processing and radar-related algorithms [13], although making a fair comparison between the performance of CPUs vs GPUs is not always trivial. The most obvious is way to compare the computation time for various tasks, but power and energy consumption needs to be taken into account as well. Besides these factors, metrics such as floating point operations per second (FLOPS) or the overall data throughput can be of great interest to evaluate the performance of an algorithm, especially when attempting to locate bottlenecks.

1.1 Project goal

The aim of this project is to design an easy to extend framework which provides users with important performance metrics when running an algorithm on the GPU. The project also includes the task of implementing from scratch and benchmarking three algorithms which are common in signal processing and/or included in the HPEC suite [7], namely QR-factorization, pattern matching and bicubic spline interpolation.

1.2 Summary of GPGPU history

General purpose computing on graphics processing units (GPGPU) is the catch-all term for using dedicated GPUs to perform tasks usually done by CPUs. Historically GPUs have only been used for graphical computation (hence the "G" in GPU), but around the early 2000s programmers began to implement algorithms to solve problems easily translatable to the graphical nature of a GPU. These algorithms included operations on objects such vectors and matrices [5]. A great hurdle back then was to translate the problem from a computational form to a form supported by graphics APIs (Application programming interface), usually DirectX or OpenGL, which were of course designed solely with graphics rendering in mind. Nowadays there are multiple API options available designed for GPGPU, such as OpenCL by Khronos Group [12] and CUDA by Nvidia [4], with CUDA being the API of choice for this project. Most modern Nvidia GPUs are now CUDA-enabled, i.e. they support the CUDA API which allows for GPGPU on any modern Nvidia card.

1.3 The Nvidia Jetson TX1 and TX2

The GPU modules used for this project are the Nvidia Jetson TX1 and TX2, which will henceforth usually be referred to as the TX1 and TX2, or simply the Jetson modules. These are marketed by Nvidia as "supercomputers on a module" [9]. They combine high performance GPU capabilities

with ARM CPU architecture [2] and are designed with embedded efficiency in mind. Specification-wise the TX2 is an upgraded TX1, delivering even greater performance overall. Unlike regular dedicated GPU setups where GPU and CPU memory is completely separate, the Jetson modules share RAM between GPU and CPU. This allows equal memory access between host and device and removes the need to move data between the two parts before and after GPU execution.

This project was developed and tested on a TX1 module and then benchmarked on the TX2. The TX2 was considered more interesting to benchmark as it provides more detailed power consumption metrics than the TX1. Pattern matching was however also benchmarked on the TX1 to give a comparison between the two modules.

2 The CUDA Programming Model

CUDA is the programming model and API developed by Nvidia. It allows programmers to write code for CUDA-enabled GPUs, using well-known languages such as C/C++ or FORTRAN [1]. The idea of CUDA is to have an intuitive way of utilizing the highly parallel architecture of the GPU with high performance computing in mind as well as allowing for good scalability and future compatibility with both low- and high-end graphics cards. For a more in-depth guide to CUDA, see the CUDA programming guide [11].

2.1 Thread Hierarchy

Threads are the lowest level of the abstract CUDA representation of GPU cores which execute the code. A set of instructions (or a function) performed by the threads is called a kernel, with the execution of code referred to as launching a kernel. Threads are organized in three ”dimensions” as blocks, which in turn form a three-dimensional grid. As the blocks can be of one, two or three dimensions, the programmer can intuitively visualize the parallel execution and work partitioning of a particular algorithm.

Synchronization of the threads within a block can be done with a function call, and each block executes independently from other blocks - and in any order. The independency of the block execution can present a challenge when writing code as there is no simple inter-block communication; it does however give better code scalability as increasing the problem size will often only linearly affects execution time (this depends on the implementation, of course). The block independency makes CUDA code highly backwards compatible. Code written for an older or lower end GPU should work just as well on a modern or higher end GPU, as they are executing the same blocks but at different rates and number of blocks at once. In most cases however old code can be optimized as GPU architecture improves.

Global memory is accessible by all threads and persistent over kernel multiple kernel launches, but is much slower than most other alternatives. Shared memory is accessible by all threads within a block and is 100 times faster than the global memory but persists only as long as the block does. Each thread also has its own register memory, which is the fastest of the three. Smart utilization of shared memory and registers is imperative to create efficient GPU code. There are also other memory types available in CUDA, such as constant and texture memory, which have their particular use but are not discussed here.

2.2 Xtream: A CUDA C++ productivity framework

Xtream is a header-based CUDA C++ productivity library provided by High Performance Consulting (HPC) [6]. It allows for convenient transfer and allocation of data between host (CPU) and device (GPU) as well as many other handy functions, such as array slicing operators and simplified ways of allocating shared memory. The benchmarking framework and algorithms written for

this project are heavily dependent on Xtream, which substantially assisted the authors in writing efficient CUDA code during the short time frame available to them.

3 Auto-testing Framework

The auto-testing framework is designed to easily be extendable to include any GPU or CPU algorithm written in CUDA C++. It measures the computation time and power consumption while the GPU is performing computations, and also counts the total number of floating point operations performed and load/store size (i.e. the byte size of the data written to and read from global memory), which must be provided by the algorithm implementation. The framework algorithms are written in CUDA C++ with Xtream, while the measurement of the various metrics is handled using Python scripts. The parameters to benchmark an algorithm with the framework are provided by JSON [8] configuration files. These are used to describe what input parameters to use for the algorithm, as well as additional information required by the framework, such as which timing type to use (GPU or CPU). An example configuration file is presented in Section 3.2. A Python script runs the algorithm for a specific configuration and returns the performance metrics in easy to read comma separated value files (CSV).

The framework is structured so that all algorithms must extend an abstract class *BaseAlgorithm* at the top of the hierarchy. This class provides the measurement method while defining pre-run, run and post-run methods which the attached algorithms override. Two other abstract classes, *GPUAlgorithm* and *CPUAlgorithm*, inherit from *BaseAlgorithm*. This allows algorithms to inherit the appropriate timing function whether it is written for a GPU or CPU. The processor type of an algorithm is then defined in the JSON configuration file. Any future algorithms implemented into the framework must be structured to extend these abstract classes appropriately.

3.1 Measurement tools

3.1.1 Computation time

The Nvidia visual profiler (nvprof) is a useful tool to view performance metrics for a specific CUDA program. Initially the framework was designed to utilize this to time the computation of an algorithm, but it proved difficult to neatly sum up the times provided by nvprof in cases when the algorithm contained multiple kernels, or if the number of kernel launches was determined dynamically at run time. This problem could be solved by dramatically increasing the complexity of the configuration files, but it was decided that it would be simpler to instead implement software timer functions into the *BaseAlgorithm* class, which are then read by the framework. This also decreases the external dependencies of the framework. A consequence of this is that software timers are not as reliable as nvprof: to solve this each algorithm is run multiple times and the computation time averaged.

3.1.2 Power

Both the TX1 and TX2 contain current and voltage monitors which provide the board with the current power consumption of the CPU, GPU and RAM over an I^2C [10] bus. The board then writes this information to data files in real-time. This allows reasonably accurate power measurements without an external measurement setup. In earlier work a Python script solution has been developed which reads these files over multiple executions of the algorithms and averages the measured power. This approach is also used in this project, but generalized to work on both the TX1 and TX2 and to be extendable to future platforms.

3.1.3 Other metrics

Theoretical FLOP and load-store sizes are counted "by hand" in the implemented algorithms and compared to the measurements of nvprof to assure that the measurements are accurate. Since the GPUs are quite fast, the FLOP measurements are usually on an order of magnitude over 10^9, i.e. as Giga-FLOP (GFLOP). Other performance metrics such as FLOPS, load-store efficiency and energy consumption are then derived using the four basic metrics.

3.2 Generating performance metrics

To generate the metrics into CSV-files using the framework, one simply runs the following commands in a terminal window:

```
python src/main.py src_app/example-algorithm/example.config.json
```

Suppose the JSON configuration file had the following structure (with completely arbitrary values):

```
{
        "executable" : "/path/to/executable",
        "processor" : "GPU",
        "kernel_name" : "functionName",
        "output_file_prefix" : "testOutput",
        "static_args" : [
                ["--length",    32],
                ["--version",   4]
        ],
        "run_arg" : {
                "one" : ["--runs", 1],
                "many" : ["--runs", 150]
        },
        "sweep_arg" : {
                "name" : "--inputsize",
                "values" : [10,1000,10000]
        }
}
```

Then the created metric files in the current working would be:

- testOutput_flops.csv
- testOutput_loadstoresize.csv
- testOutput_power.csv
- testOutput_time.csv

3.3 Generating plots

After the files have been generated, the framework also provides a handy way of plotting the results using R-functions that are tailor made for the auto generated output files.

4 Benchmarking Algorithms

4.1 Description of the Algorithms

4.1.1 QR-factorization

QR-factorization is the process of decomposition an $m \times n$ matrix A into an orthogonal $m \times n$ matrix Q and an upper triangular n x n matrix R, such that $A = QR$. There are different ways of constructing the Q-matrix. The one used in this project was Modified Gram-Schmidt orthogonalization (MGS) [3]. The reason for using MGS as opposed to other methods (e.g. Householder reflections or Givens rotations) was due to the authors' previous experience with implementing MGS on CPUs.

The Q-matrix is constructed by applying Gram-Schmidt orthogonalization to the column vectors of A. Let a_j be a column vector of A and $proj_{q_i}(a_j) = \frac{\langle q_i, a_j \rangle}{\langle q_i, q_i \rangle} q_i$, i.e. the projection of a_j onto the vector q_i. The Gram-Schmidt method is then defined as follows:

$$q_1 = \frac{a_1}{||a_1||}, \tag{1}$$

$$q_2 = \frac{a_2 - proj_{q_1}(a_2)}{||a_2 - proj_{q_1}(a_2)||}, \tag{2}$$

$$q_3 = \frac{a_3 - proj_{q_1}(a_3) - proj_{q_2}(a_3)}{||a_3 - proj_{q_1}(a_3) - proj_{q_2}(a_3)||}, \tag{3}$$

$$\vdots \tag{4}$$

$$q_j = \frac{a_j - \sum_{i=1}^{j-1} proj_{q_i}(a_j)}{||a_j - \sum_{i=1}^{j-1} proj_{q_i}(a_j)||}. \tag{5}$$

This produces a matrix Q consisting of n orthonormal column vectors q_j. The problem with the classical Gram-Schmidt method is that it is numerically unstable. Therefore the Modified Gram-Schmidt method (MGS) is used instead, where instead of as in 5, q_i is computed like so:

$$q_i^{(1)} = a_i - proj_{q_1}(a_i), \tag{6}$$

$$q_i^{(2)} = q_i^{(1)} - proj_{q_2}(q_i^{(1)}), \tag{7}$$

$$q_i^{(3)} = q_i^{(2)} - proj_{q_3}(q_i^{(2)}), \tag{8}$$

$$\vdots \tag{9}$$

$$q_i^{(i-1)} = q_i^{(i-2)} - proj_{q_{i-1}}(q_i^{(i-2)}), \tag{10}$$

$$q_i^{(i)} = \frac{q_i^{(i-1)}}{||q_i^{(i-1)}||} \tag{11}$$

which in exact-precision arithmetics is identical to the classical method, but is numerically stable.

Since q_i is a unit vector $proj_{q_i}(a_j) = \frac{\langle q_i, a_j \rangle}{\langle q_i, q_i \rangle} q_i = \langle q_i, a_j \rangle q_i$. The R-matrix can be viewed as consisting of n column vectors r_j of length n. The i:th element of r_j gives the weight to be multiplied with q_i in order to create a_j. For $i < j$, this means that $r_{ij} = \langle q_i, a_j \rangle$. Note that in the expression for r_{ij} for the MGS method, a_j will only actually be the column vector a_j for $i = 1$. For $i = 2$, it will be the updated vector $q_j^{(1)}$ etc. as shown in 6 - 10. For $i = j$, $r_{ij} = ||q_i||$ as seen in 11. For $i > j$, $r_{ij} = 0$ which makes R an upper triangular matrix.

4.1.2 Bicubic interpolation

A common algorithm for interpolating data, bicubic interpolation is an extension of one-dimensional cubic spline interpolation to two dimensions. By performing four cubic interpolations in one direction (for example, the column space of a matrix) and then a fifth interpolation perpendicular to the previous four interpolations (along the matrix row space) one can interpolate any point in a two-dimensional plane. For this project the only application of bicubic interpolation considered is for interpolating data when upscaling matrices.

While many may be familiar with bicubic interpolation from image processing, for example when scaling an image, it can be used in any field which requires interpolating data, like in signal processing. As bicubic interpolation is rather computationally heavy it is a useful algorithm for benchmarking. In this project we do not discuss the application of interpolating data using splines, but rather the computational efficiency of the algorithm implemented with CUDA. As such, the various mathematical properties of different bicubic spline interpolations are not considered in great detail, beyond how they can be implemented.

The one dimensional cubic splines can be constructed using four basis polynomials and a combination of the two points to interpolate in between, and the next set of points outside the interval defined by the central two points. The basis functions can be constructed in various ways, giving the spline different properties. In this project we consider Hermite basis splines and B-splines. When scaling matrices the spline types are cardinal, i.e. the input points are equally spaced. The choice of spline type when interpolating depends entirely on the problem at hand, but both splines are identical in how they are parallelized and of similar computational cost.

The general algorithm to calculate a cubic spline is as follows: We denote the original data points p_0, p_1, p_2 and p_3 as control points and want to interpolate the value p of some point x between p_1 and p_2. The intervals between controls points are normalized, so x is in the unit interval $x \in [0, 1]$ between $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$.

$$
\begin{aligned}
p_H(x) &= h_{11}(x)\boldsymbol{p_1} + h_{21}(x)\boldsymbol{m_1} + h_{12}(x)\boldsymbol{p_2} + h_{22}(x)\boldsymbol{m_2}, \\
p_B(x) &= w_0(x)\boldsymbol{p_0} + w_1(x)\boldsymbol{p_1} + w_2(x)\boldsymbol{p_2} + w_3(x)\boldsymbol{p_3}.
\end{aligned}
$$

Here p_H and p_B are the interpolated values using the corresponding spline types, Hermite and B-spline, and m_1 and m_2 are the tangents in points p_1 and p_2. The basis functions w and h are defined as

$$
\begin{aligned}
h_{11}(x) &= 2x^3 - 3x^2 + 1, & w_0(x) &= \frac{1}{6}(-x^3 + 3x^2 - 3x + 1), \\
h_{21}(x) &= x^3 - 2x^2 + x, & w_1(x) &= \frac{1}{6}(3x^3 - 6x^2 + 4), \\
h_{12}(x) &= -2x^3 + 3x^2, & w_2(x) &= \frac{1}{6}(-3x^3 + 3x^2 + 3x + 1), \\
h_{22}(x) &= x^3 - x^2, & w_3(x) &= \frac{1}{6}(x^3).
\end{aligned}
$$

The main difference between the two spline types, beside their basis polynomials, are how the control points are used. Hermite interpolates requires the two points to interpolate between and the derivative in those points while B-splines simply use the four control points directly. The Hermite splines in this project are Catmull-Rom splines: the tangents m_1 and m_2 are calculated using central differences.

4.1.3 Pattern matching

Pattern matching compares a search vector to template vectors in a library. This is done by taking the sum of the mean square errors (MSEs) between each element in the search vector a and template vector t_k, where k is one of K patterns in the library. Due to the fact that a search vector (or signal) can correspond to a correct pattern but with a misalignment along the horizontal axis, the best shift also needs to minimize the MSE. This is done by "brute force": trying the S possible shifts from $-S/2$ to $S/2$. A signal can also correspond to a pattern but with the wrong amplitude scaling. The best scaling (or gain) for a pattern can be found by the same brute force method as the shift.

The first step in the algorithm is to find the optimum shift s so that Equation 12 is minimized for some pattern k,

$$\sum_{s=-S/2}^{S/2} \sum_{n=1}^{N} \left(w_n * (a_n - t_{k,n+s})^2 \right), \tag{12}$$

where w is a input parameter which corresponds to a weight vector of size N.

The second and last step in the algorithm is to find a gain value in a vector v of size G which minimizes Equation 13 for a pattern k, with the correct shift cs found in step 1:

$$\sum_{g=1}^{G} \sum_{n=1}^{N} \left(w_n * (a_n - t_{k,n+cs} * v_g)^2 \right). \tag{13}$$

When this is done, the pattern with the lowest MSE for the best shift and gain values corresponds to the best match. Observe that if $\sum_{n=1}^{N} w_n$ is not equal to one, equation 12 and 13 need to be divided by that value. In this project it is assumed that the input is already normalized.

4.2 Implementation

4.2.1 Porting old work

In addition to implementing the algorithms proposed in the section above, four algorithms developed by a previous project done at High Performance Consulting were also ported to work with the framework. This was primarily done as a learning experience for the CUDA workflow, but also to include older work to extend the framework. The process of fitting these algorithms into the framework proved to be very simple, as very little code related to the actual algorithm needed to be altered. In the future we hope that the framework can be extended to include even more algorithms (or other implementations of current ones), to make for an even more accurate benchmark.

4.2.2 QR-factorization

QR-factorization (both full and reduced) is implemented on the GPU by looping two kernels: the first being the normalization of one column vector of A and the other one being the projection of the remaining vectors onto the space orthogonal to the normalized vector. One block is assigned to each vector of A and the projection of vectors are executed in parallel. The vector operations within the kernels; the scalar product, the normalization and projection are straightforward to parallelize. The summation involved in the scalar product is obtained through parallel reduction. To further increase the efficiency of the algorithm a second version is implemented, where the vectors are loaded into shared memory in each kernel. In this way many global read and writes can be avoided. In an attempt to utilize the register memory instead of having to read and write

Table 1: Overview of the 8 different implementations of QR-factorization

Version	Method
v1	Global read & write
v2	Shared memory
v3	Stack variables
v4	Global read & write (64x64)
v5	Shared memory (64x64)
v6	Global writes at end (64x64)
v7	Warp functions (64x64)
v8	Different work partitioning (64x64)

to either global or shared memory, a third version where data is stored as stack variables when possible is also implemented,

Additional versions are created specifically for 64 x 64 matrices, which is a common problem size in radar signal processing. This enables the use of warp functions which further decreases the demand for data loads and stores. To simplify communication and to require less overhead for thread synchronization these implementations only use one block, since multiple kernels would not need to be launched to execute the algorithm. Different schemes for distributing the work among the threads to further decrease the need for thread synchronization and minimize the effect of load and store latency are also implemented. In total five versions for 64 x 64 matrices are created; the first one using only global memory, the second with shared memory, the third moving all global writes to the end of the kernel, the fourth utilizing warp functions and the fifth version using different workload partitioning to try to reduce the number of synchronization calls and shared memory reads and writes. Table 1 shows an overview of the different versions.

4.2.3 Bicubic interpolation

The first implementation is made by letting each thread represent a point in the scaled up data. This means that each thread loads 16 control points from the original data and calculates the five required 1D spline interpolations to interpolate an output point. This method utilizes no block or thread cooperation and does many unnecessary computations when the row scaling is large, as many threads in the same column recalculate the same column interpolations for different rows. The same is true for rows and row scaling. It also loads any control points unnecessarily, as nearby threads have no way of sharing control points between them. Implementing bicubic interpolation this way is very simple, especially if a CPU implementation is available, since it essentially consists of replacing a loop iterator with a thread and block index. Unfortunately improving this to reduce unnecessary loads and spline computations using shared memory proved difficult, as CUDA blocks in a kernel must all be the same size, but the interpolating points do not always line up so nicely. This could be solved by making the blocks larger than necessary but this would likely result in very poor utilization and therefore poor performance. A better solution is found by looking at alternative implementation methods.

The second approach is to let each thread handle a point in the initial, non-scaled up data. This makes the threads much "heavier", with each thread performing multiple 2D spline calculations, especially for large scale factors. This implementation can more easily utilize shared memory by loading the control points required by the threads of a block into shared memory. Three different versions of this approach are presented. The first interpolates all points along the rows (Y-axis) and then along the columns (X-axis). The second transposes the input array, interpolates along the former row axis (which is now an X-axis, hopefully improving data locality and thus the fetching speed), transposes the array back and interpolates along the original X-axis. The third version

Table 2: Overview of the 4 different implementations of bicubic interpolation

Version	Method
v1	Light threads
v2	Heavy threads
v3	Heavy threads & transpose
v4	Heavy threads, transpose & kernel splitting

is the same as the second but also splits the array into chunks when interpolating, to launch in multiple kernels. This is written in such a way that the algorithm can handle very large problem sizes without overloading the GPU memory.

The first approach is henceforth described as using "light" threads, and denoted as Version 1. Versions 2, 3 and 4 are the approaches described using the "heavy" thread parallelization. An overview of the methods is shown in Table 2.

4.2.4 Pattern matching

Five different versions of pattern matching are implemented, and are described here in order of implementation:

Version 0 is a trivial single core CPU implementation used for testing, in comparison and error checking.

For Version 1 the threads were divided across three dimensions in each block. Y-threads handle different patterns k, X-threads different search vector elements n, and Z-threads make up the different shifts s in equation 12 and different gains g in equation 13. All $K \times N \times (S+G)$ error terms were computed by one single thread. $K \times N \times S$ by one kernel with the thread dimension (K, N, S) and $K \times N \times G$ by another kernel with another thread dimension (K, N, G). The resulting terms are saved to global memory and then summed up by the first s-thread for the first kernel and the first g-thread for the second kernel.

Version 2 is implemented in the same way as Version 1 but with the resulting MSE terms written to shared instead of global memory.

In Version 3 the template vector sizes are limited to 32 elements, so that the CUDA-function __shfl_down_sync could be used to more effectively reduce the sums inside a warp.

Version 4 has the same assumption about the vector sizes as Version 3. However the thread mapping is changed so that the first kernel only has threads in two dimensions instead of three. Y-threads still handle patterns, X-threads handle vector terms; however, instead of using Z-threads the X-threads loop over all the possible shifts. This leads to more work done by each thread. The shifted template values are read from neighbouring threads in the same warp using __shfl_sync.

An overview of all implemented pattern matching versions is shown in Table 3.

Table 3: Overview of the 5 different implementations of pattern matching

Version	Method
v0	Trivial single core CPU implementation
v1	Global memory & 3D thread mapping
v2	Shared memory & 3D thread mapping
v3	Shared memory, 3D thread mapping & warp functions (limited input size)
v4	Shared memory, 2D thread mapping & warp functions (limited input size)

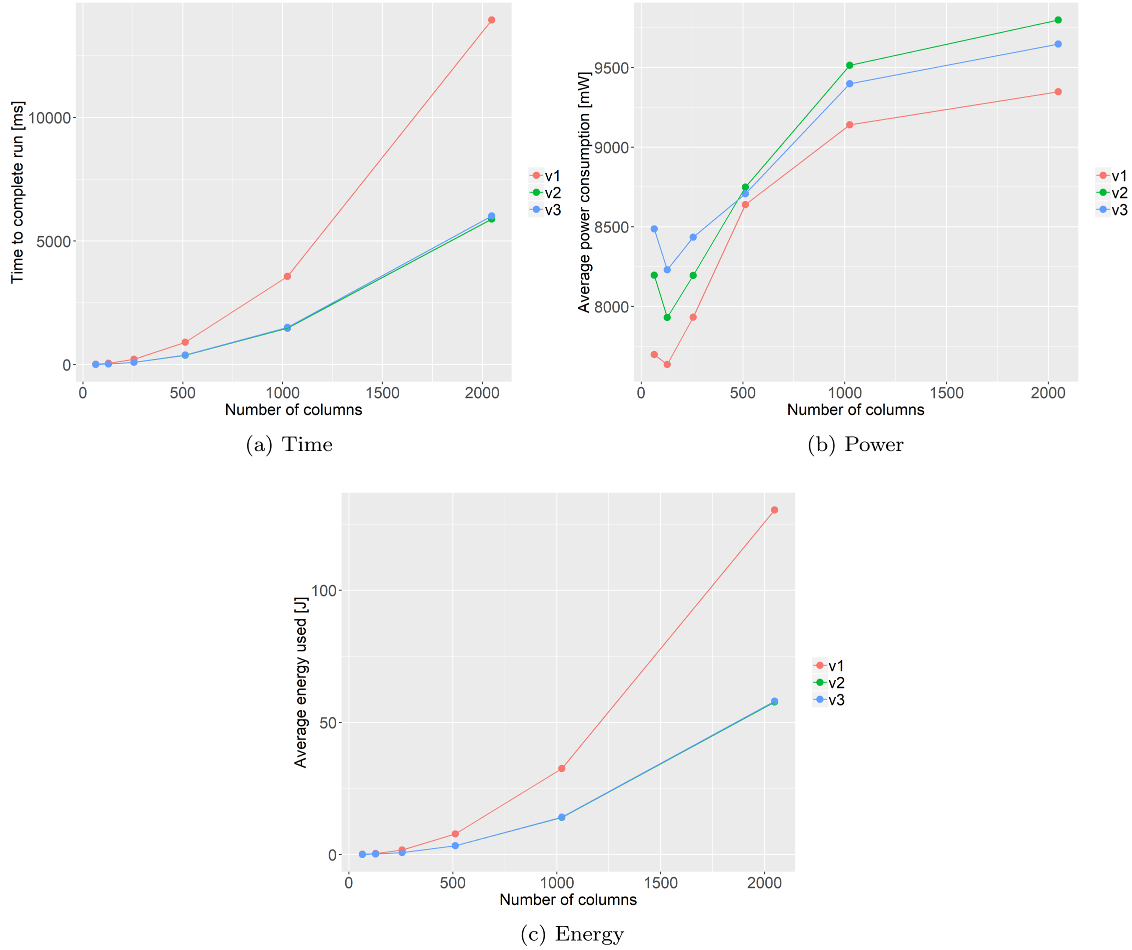


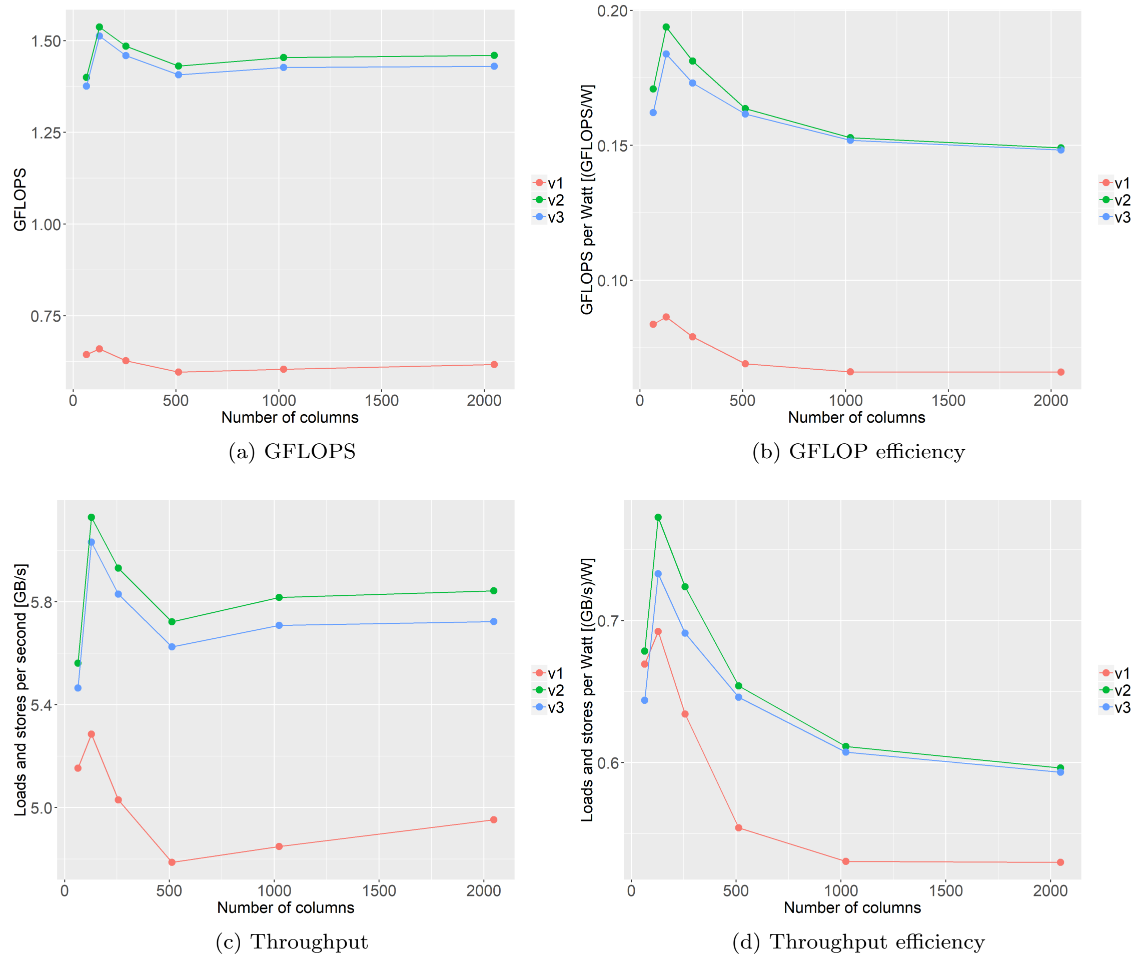
Figure 1: Execution time, power consumption and total energy usage for the global (v1), shared memory (v2) and register versions (v3) for different problem sizes (1024 rows, x columns).

5 Performance Results

In this section the different performance metrics for the three implemented algorithms are presented.

5.1 QR-factorization

Significant performance improvements are achieved by utilizing shared memory. Figures 1 and 2 show various performance metrics for various problem sizes for the three versions designed for arbitrary problem sizes. Additional increases in performance is achieved when optimizing for the specific problem size of 64×64 matrices. A comparison of different performance metrics between all versions is presented in Figure 3. The benchmarks for varying problem sizes (Figures 1 and 2) were done on the TX2, while the benchmark for 64×64 matrices (Figure 3) was done on the TX1.



(a) GFLOPS

(b) GFLOP efficiency

(c) Throughput

(d) Throughput efficiency

Figure 2: QR-factorization performance metrics for the global (v1), shared memory (v2) and register versions (v3) for different problem sizes (1024 rows, x columns).

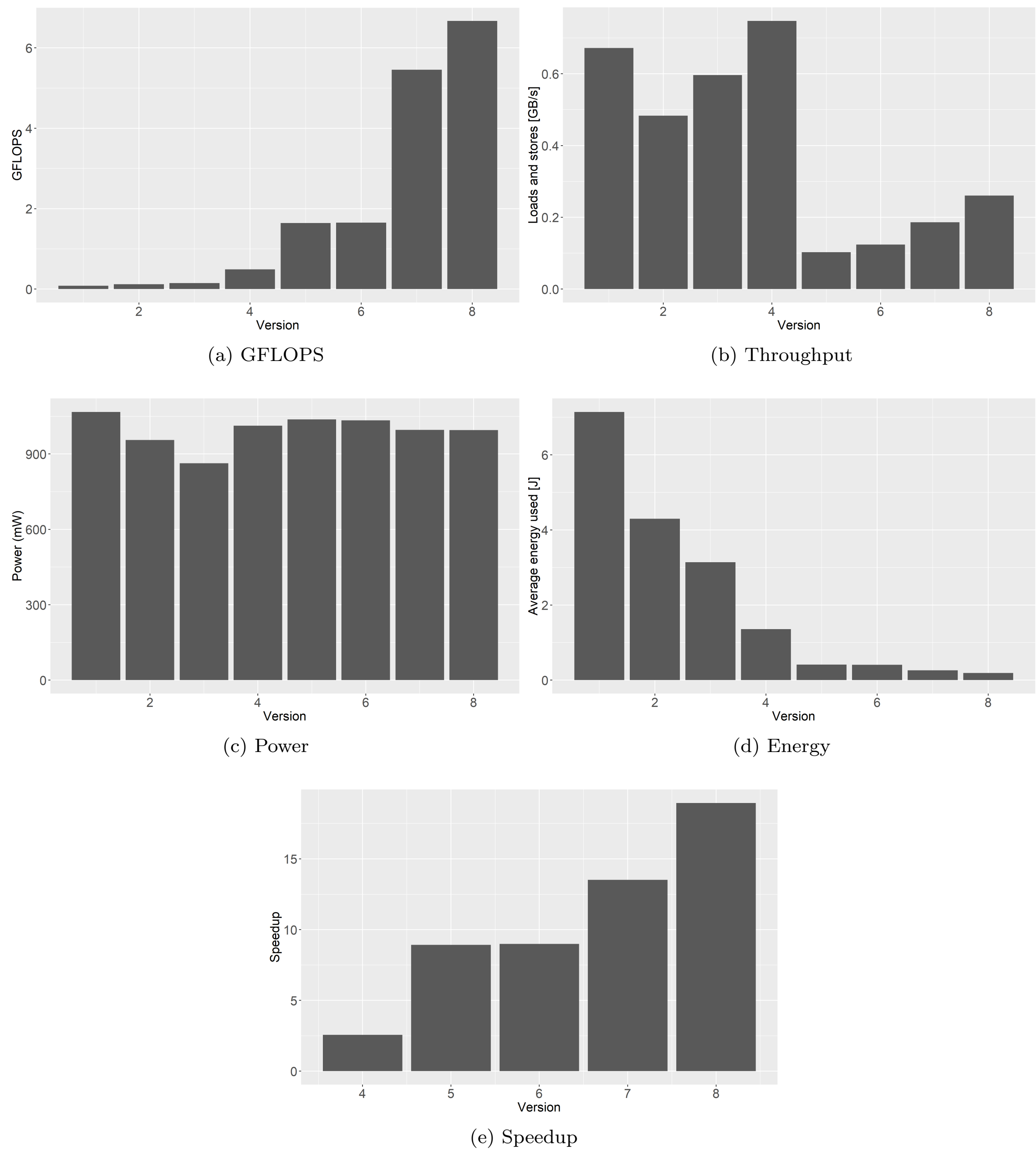
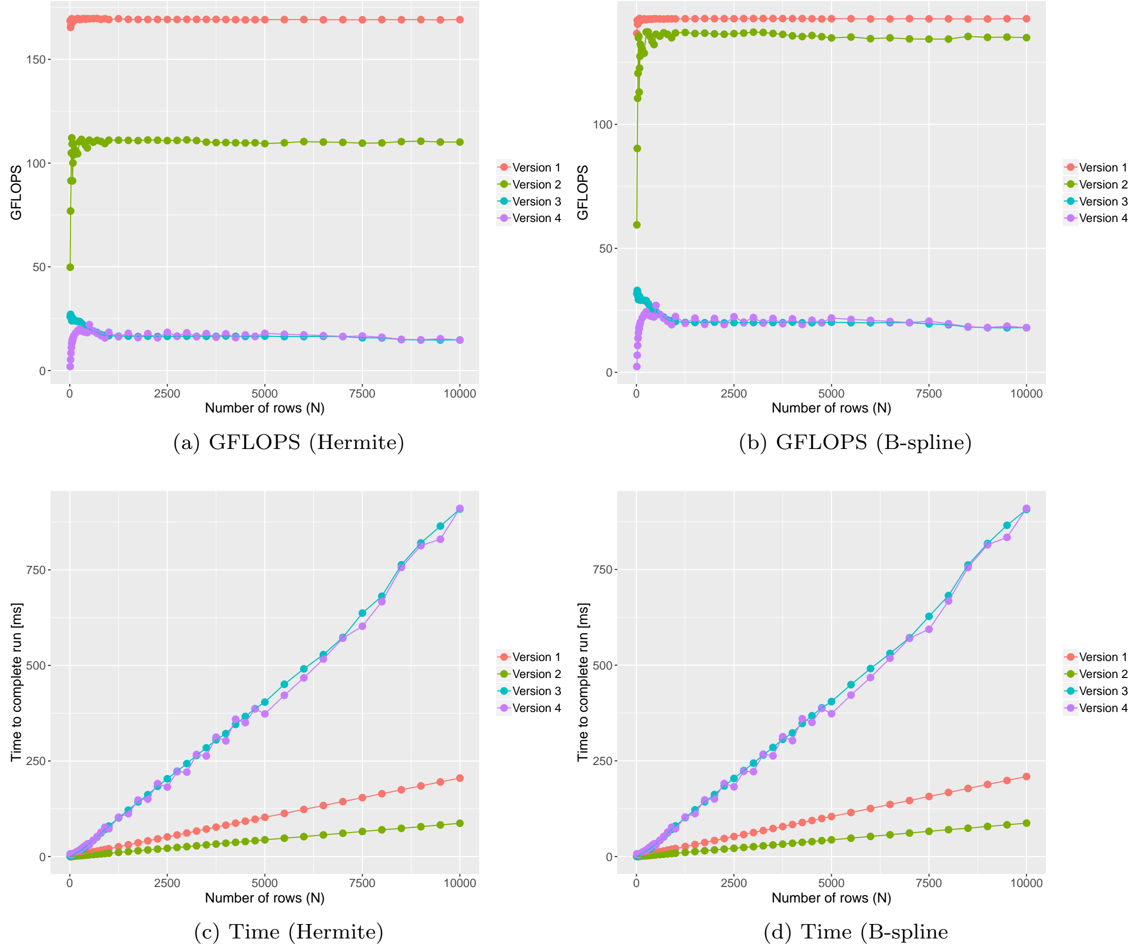


Figure 3: A comparison of the average (from 1000 runs) GFLOPS, throughput, power, total energy and speedup between different versions on 64 x 64 matrices. Versions 1-3 work for arbitrary problem sizes while Versions 4-8 are designed specifically for this problem size. Speedup is relative to the fastest version designed for arbitrary problem sizes (Version 3).



(a) GFLOPS (Hermite) (b) GFLOPS (B-spline)

(c) Time (Hermite) (d) Time (B-spline

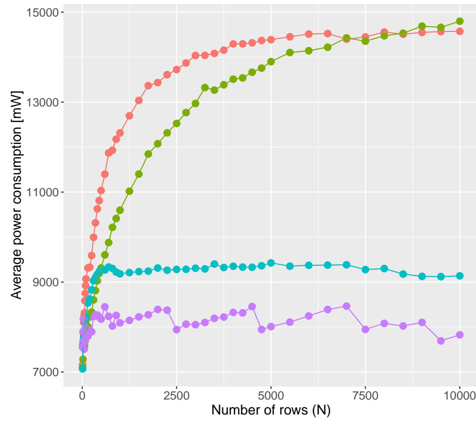
Figure 4: Bicubic interpolation performance metrics when varying the input size, for the different versions and spline types.

5.2 Bicubic interpolation

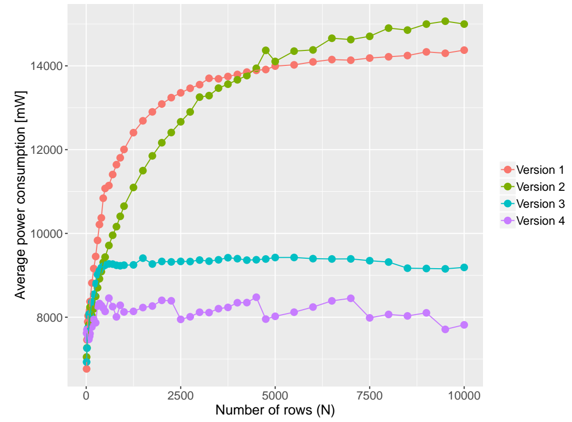
All the results shown in this section are from benchmarks on the TX2. Each benchmark is run 50 times and the results are averaged to allow for more accurate power consumption metrics. The versions are as described in Table 2 in Section 4.2.3.

5.2.1 Varying input size

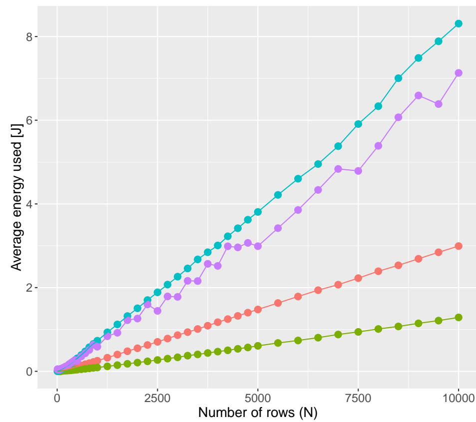
The input size is varied only in one dimension, namely we consider only different number of rows. The other parameters are kept constant: 2048 columns, 2 times column scaling and 8 times row scaling. Figure 4 shows how the theoretical FLOPS and execution time change for different sizes of input matrices. Power- and energy consumption, and FLOP efficiency is shown in Figure 5.



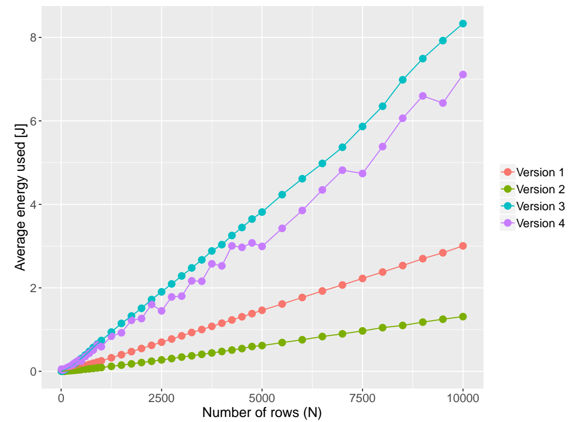
(a) Power (Hermite)



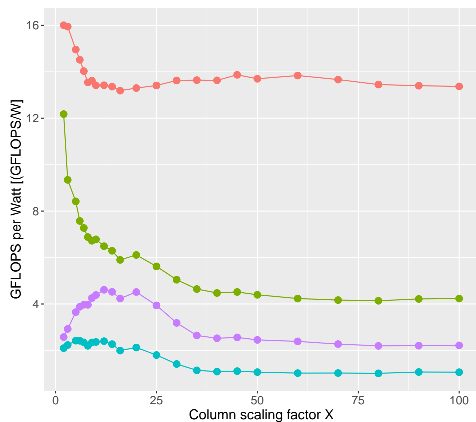
(b) Power (B-spline)



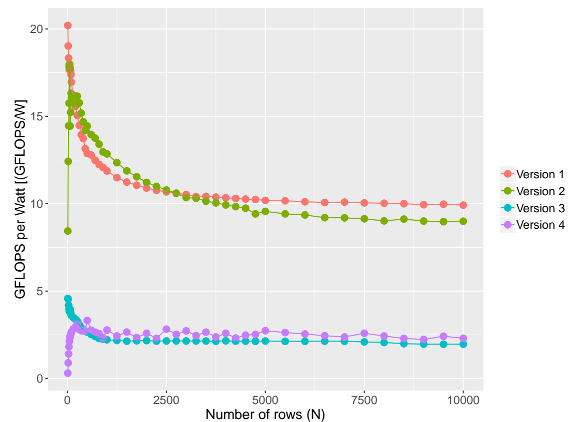
(c) Energy (Hermite)



(d) Energy (B-spline)

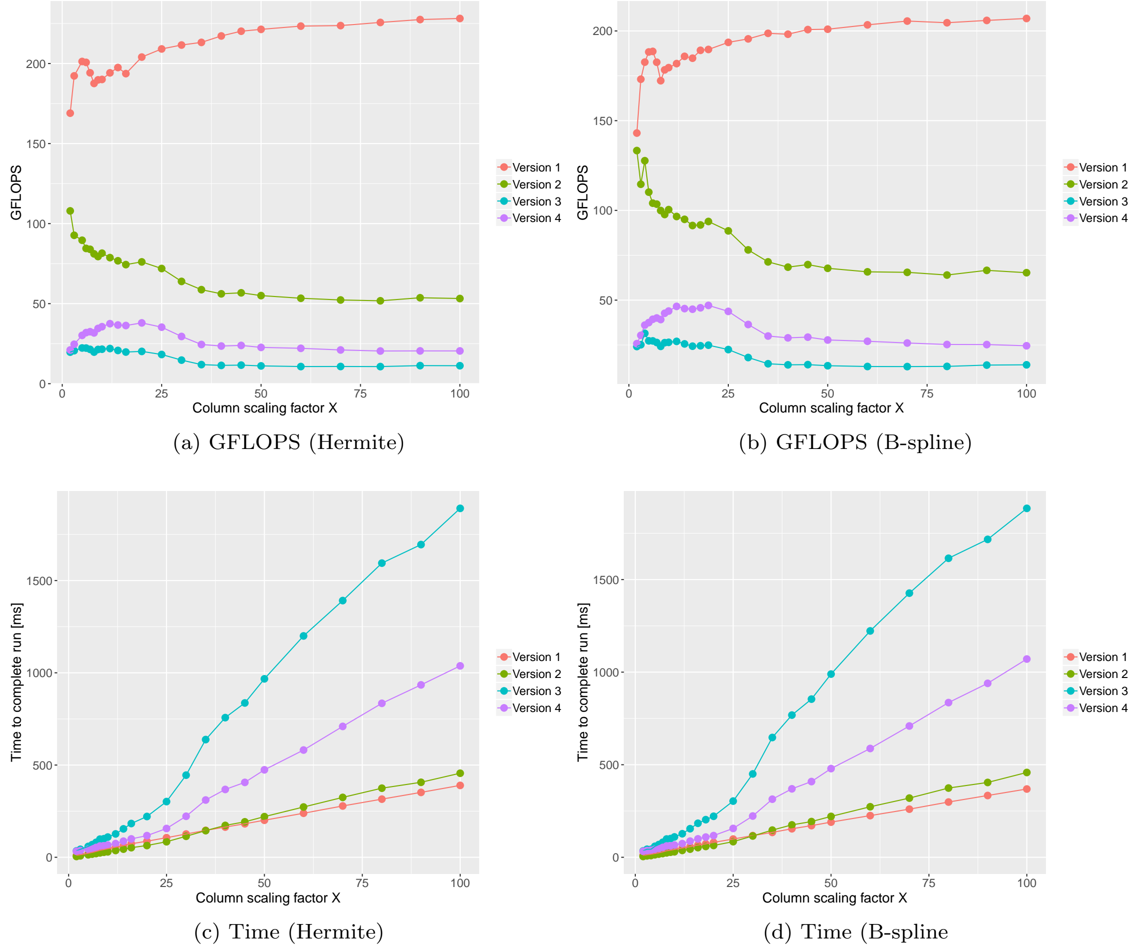


(e) GFLOPS efficiency (Hermite)



(f) GFLOPS efficiency (B-spline)

Figure 5: Bicubic interpolation performance metrics when varying the input size, for the different versions and spline types.



(a) GFLOPS (Hermite)

(b) GFLOPS (B-spline)

(c) Time (Hermite)

(d) Time (B-spline

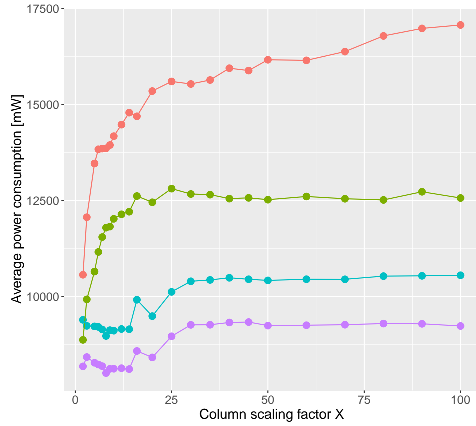
Figure 6: Bicubic interpolation performance metrics when varying the scale factor, for the different versions and spline types.

5.2.2 Varying the scale factor

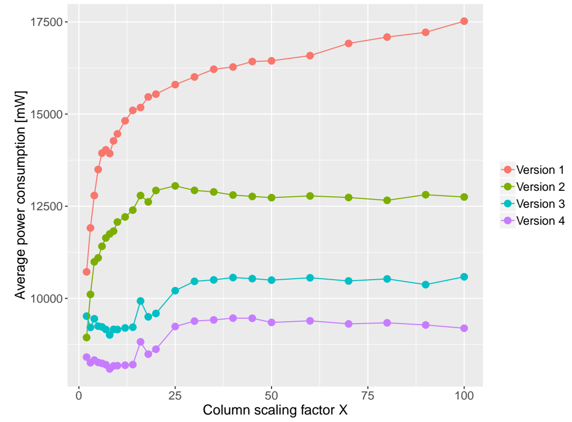
For these benchmarks only the column scaling factor is considered. The other parameters are kept constant at 512 rows, 2048 columns and 8 times row scaling. Figure 6 shows how the theoretical FLOPS and execution time changes as the scaling increases. Power- and energy consumption, and FLOP efficiency is shown in Figure 7.

5.3 Pattern matching

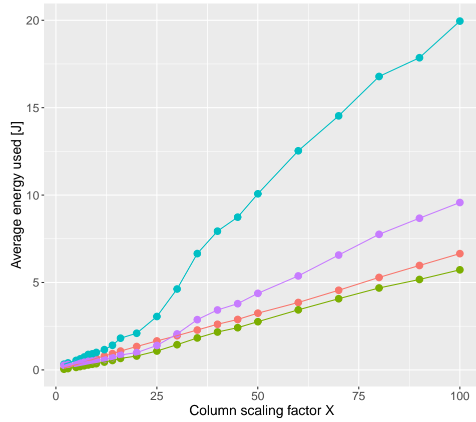
The input used for the results, presented in Figure 8 and Tables 4-5, is a vector size of $N = 32$, $S = 101$ shifts and $G = 50$ gains. Uniquely for pattern matching, the benchmarks are run in different power modes on the TX2, though unless stated otherwise the power mode is set to high performance (power mode 0). All data collected is an average based on 150 runs on the TX1 and 75 runs on the TX2.



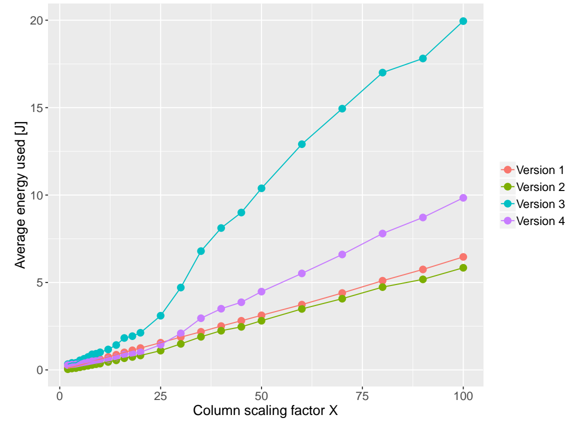
(a) Power (Hermite)



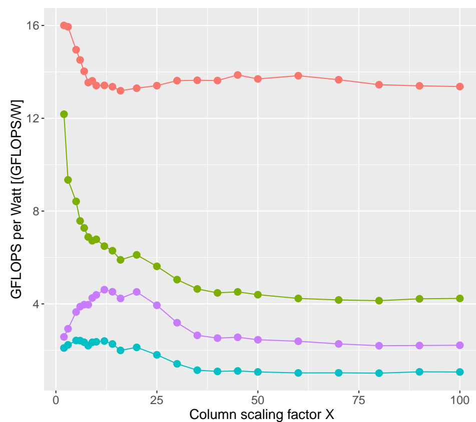
(b) Power (B-spline)



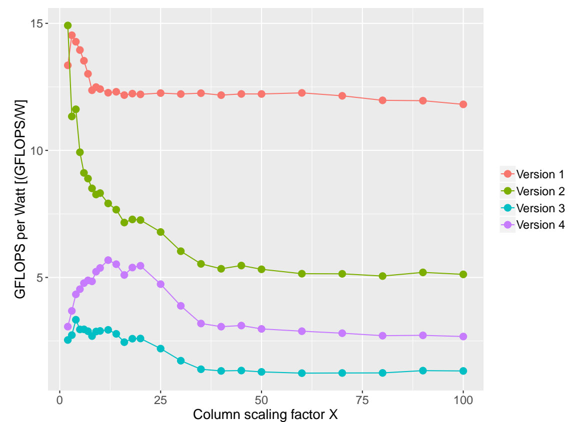
(c) Energy (Hermite)



(d) Energy (B-spline)

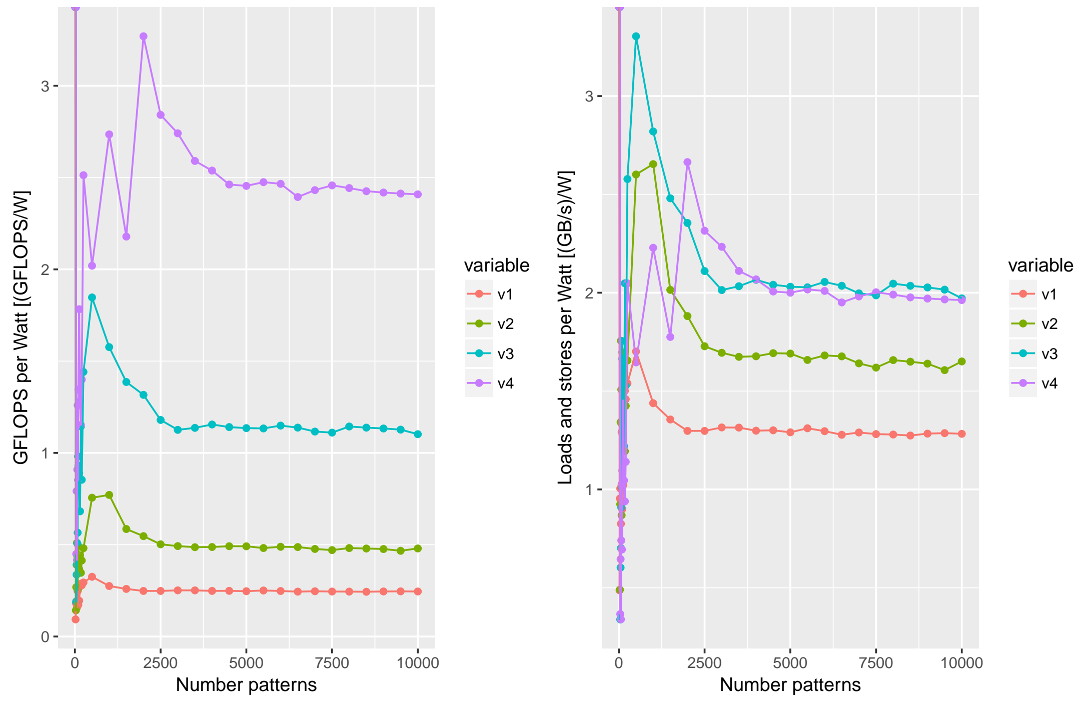


(e) GFLOPS efficiency (Hermite)



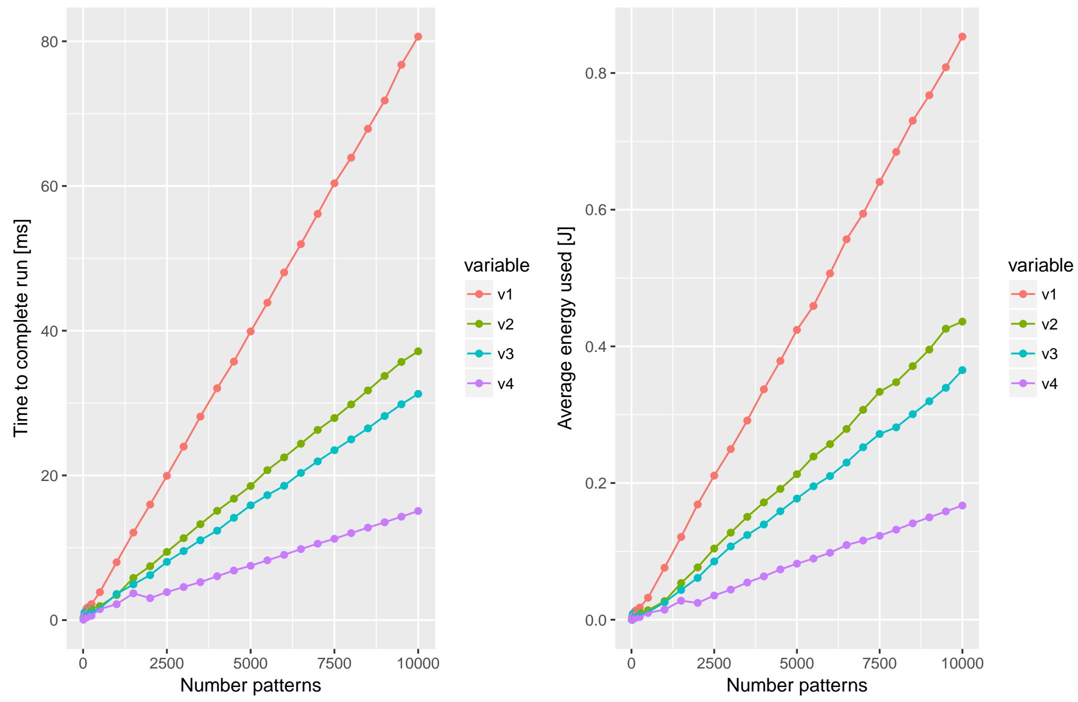
(f) GFLOPS efficiency (B-spline)

Figure 7: Bicubic interpolation performance metrics when varying the scale factor, for the different versions and spline types.



(a) GFLOP efficiency

(b) Throughput efficiency



(c) Time

(d) Energy

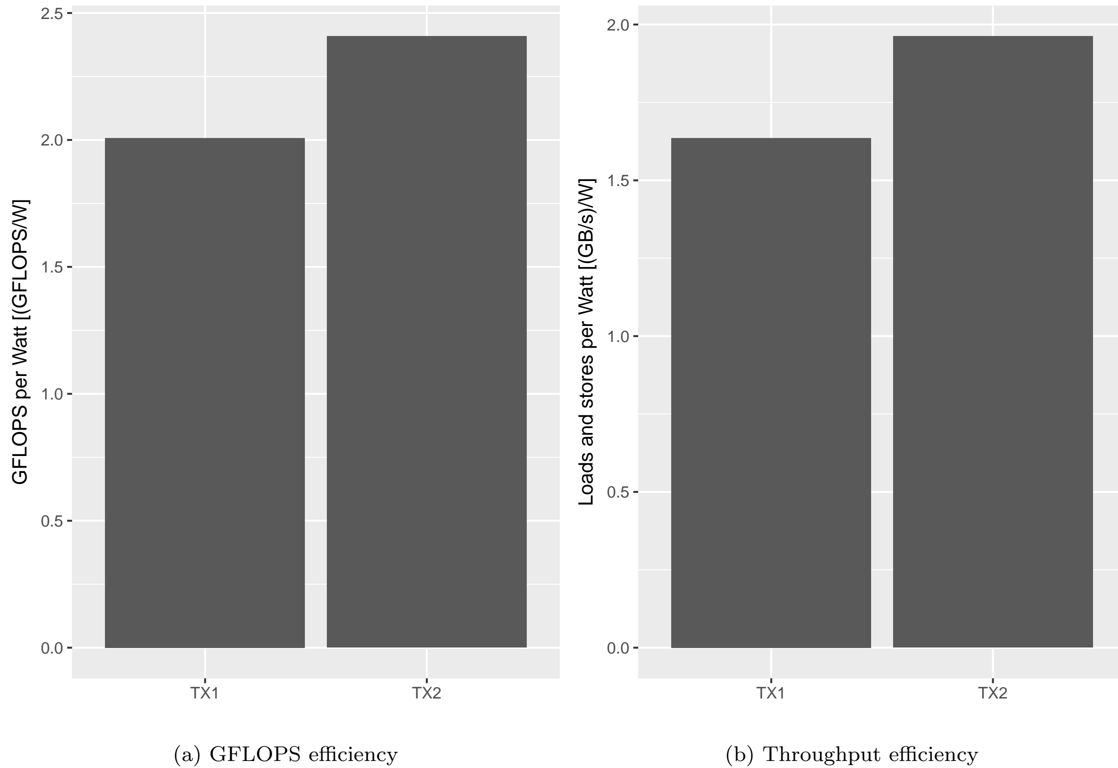
Figure 8: Pattern matching performance metrics on the TX2 for the four different versions while varying number of patterns.

Table 4: Speedup for the GPU versions of pattern matching (v1-v4), compared to the single core CPU version (v0). Here $K = 3000$.

Version	TX1	TX2
v0	1.0	1.0
v1	8.8	4.4
v2	18.2	9.4
v3	22.2	11.1
v4	44.2	23.2

Table 5: Power mode efficiency's on TX2 for v4, $K = 5000$

Power mode	Time (ms)	Power (W)	Energy (mJ)
0	7.55	11.3	85.3
1	9.52	7.0	66.6
2	8.18	8.8	72.0
3	8.14	9.3	75.7
4	8.24	7.6	62.6



(a) GFLOPS efficiency

(b) Throughput efficiency

Figure 9: FLOP efficiency and throughput efficiency for Version 4 on the TX1 and TX2 for 10000 patterns.

6 Discussion

The results of running the different benchmarks on the implementations of the three algorithms are discussed and plausible reasons for the outcome of the benchmarks are presented. An unwanted but unavoidable side effect of utilizing software timing functions is that the timing is not very precise for very quick kernels. This means that for small problem sizes, the benchmark results are not very accurate. This is especially apparent in all the pattern matching benchmarks, where the computation time varies wildly for small sweep parameter values. The effect is less prominent for the bicubic interpolation results as the problem size is always kept large. These poor timing values should not detract from the overall discussion on the benchmarks however, but require any conclusions drawn from small input sizes to be taken with a grain of salt.

6.1 QR-factorization

Utilizing shared memory and the special warp functions proved to be highly important in order to improve performance. Other performance increases are made by properly partitioning the work among the threads, as well as minimizing the number of synchronization calls. Unsurprisingly, knowing the problem size in advance allows for further specialized optimizations, which results in greatly improved performance.

The three versions, designed for general problem sizes (Versions 1-3) are tested for several performance metrics and the results are presented in Figures 1 and 2. Version 1, which uses only global memory performs worse across the board compared to the other two versions. This is mostly due to the increased execution time as a result of the superfluous global reads and writes. The only metric where this version gives an arguably more favorable result is the power consumption as shown in Figure 1. This however is only due to the significantly longer execution time, and as seen in Figure 1 the total energy consumption is much higher for Version 1 compared to the other versions. Versions 2 and 3 perform equally in all measurements. Both versions utilize shared memory, but in Version 3 an attempt is made to reduce shared memory loads and stores and instead use register memory. This is done by declaring variables on the stack instead when possible. The reason that no performance increase can be observed is most likely that the amount of work per thread is relatively low in both versions. This means that once the data is loaded, it will likely stay in register memory for both versions until the kernel is finished, thus effectively making the versions equal in terms of performance.

Additional significant performance increase can be gained when optimizing the implementation for specific problem sizes (64 x 64 matrices). This problem size is of particular interest because it is a typical problem size for radar signal processing algorithms. A comparison between all versions on 64 x 64 matrices for several different metrics are presented in Figure 3, with Versions 4-8 being specifically designed for this problem size. Not surprisingly these versions outperform the more general versions in all metrics. Floating point operations per second is the highest for the fastest algorithms which can be seen by comparing the FLOPS and speedup in Figure 3. This is partly a result of the shorter execution time, but also because the faster algorithms do more floating point operations per run, especially the two versions which use warp functions (Versions 7 and 8). Throughput is much higher for the general versions (Versions 1-3) and the 64 x 64 version which only uses global memory (Version 4) compared to the other versions. Since the FLOPS are generally higher and the throughput is generally lower for the faster algorithms, a possible conclusion is that making the algorithm computationally heavier rather than focusing on high bandwidth is desirable. Another observation is that the faster the algorithm, the less energy it requires to run. This is because power, or energy per second, is equal across all versions, as seen in Figure 3. Thus, the determining factor for total energy consumption is the execution time.

Since reducing the execution time seems to be the most effective way of increasing the performance of an algorithm, it is of interest to identify why some implementations run faster than other.

As already mentioned, the versions that are designed specifically for 64 x 64 matrices (Versions 4-8) have shorter execution times compared to the general versions. This can be seen in the speedup graph in Figure 3 that shows the speedup compared to the fastest general version (Version 3). A large contributing factor to this result is likely the fact that these versions only execute one kernel per run, thus removing the overhead for multiple kernel launches. However, significant additional reductions in execution times are achieved by utilizing shared memory (Versions 5 and 6), as well as the usage of warp functions (Versions 7 and 8). Versions 5 and 6 are both very similar. The difference is that Version 6 performs all global writes at the end of the kernel instead of continuously throughout the run, like Version 5. The idea is to prevent the threads having to wait for each other to write to global memory and instead do it simultaneously at the end. It is not clear why no speedup is achieved with this change, but it is possible that the compiler is able to do this optimization automatically or possibly that the scheduler is able to hide the global store latency at runtime.

Versions 7 and 8 utilize warp functions to do the parallel reduction required in the summation part of the dot product. Since the warp functions allow threads of the same warp to read each other's register memories, many reads and writes as well as synchronization calls can thus be removed. This has a large effect on the execution time (Figure 3). Version 8 uses a different work partitioning scheme compared to Version 7, which further reduces loads and stores from shared memory as well as decreasing the need for thread synchronization. The result is an additional reduction in execution time.

If some application requires QR-factorization of multiple matrices, the versions which use only one block for each run (Versions 4-8) could easily be extended to perform several QR-factorizations in parallel. This would further increase the GPU utilization, as only one block can only run per streaming multiprocessor. Thus doing QR-factorization of only one matrix leaves many SMs idle. Moreover, it could be worthwhile looking into other ways of orthogonalizing the matrix (e.g. Househoulder reflection or Givens rotations), which may be more suited for parallel execution.

6.2 Bicubic interpolation

6.2.1 Hermite and B-spline comparison

As expected the Hermite and B-spline implementations perform very similarly for the different versions. The only major difference can be seen in the FLOPS in Figure 4: Version 2 is much more computationally cheap for Hermite splines than B-splines. This has to do with how the Hermite basis polynomials are slightly cheaper than the B-spline one, but require the computation of the tangents between the central control points (see Section 4.1.2). Each thread in Version 2 only needs to compute these once before interpolating multiple points, while in Version 1 every thread does this separately. As a result, Hermite splines are slightly cheaper than B-splines with Version 2.

6.2.2 Interpreting the benchmarks

Comparing the FLOPS and execution times of Versions 1 and 2 (ex. Figures 4 and 6) it becomes apparent that measuring FLOPS can be misleading, and not always an accurate metric of performance. Version 2 is about twice as fast as Version 1 when varying the input size despite seemingly performing much worse with respect to FLOPS. This is due to Version 1 performing many unnecessary repeat calculations, as every thread always calculates all five 1D interpolations required for one point; in theory many of these could be re-used, which is what Version 2 does.

Instead we may consider energy usage as a measure of efficiency. In most cases Version 2 outperforms Version 1 as it both consumes less power and takes less time to compute, however it is interesting to note that Version 1 performs better than Version 2, both computation time and

energy wise, for very large scaling factors (around 25 and up). This is most likely due to how the algorithm implementations scale: Version 1 simply increases the number of blocks when the scaling factor increases, without changing the computation time per block. In Version 2 on the other hand, the blocks don't increase in number but in computational complexity. The threads likely run out of register space when they need to interpolate too many points, resulting in variables overflowing to global memory, and a much longer execution time.

6.2.3 Transpose versions

Versions 3 and 4 have not been mentioned so far, as they perform worse than 1 and 2 in every metric except for consuming less power (Figures 5 and 7). The reason for their poor performance is likely due to the matrix transpose implementation being subpar. Like the algorithms themselves, the matrix transpose is implemented from scratch for GPU, as to the project heavily relies on special Xtream buffers to move data between the CPU and GPU. It is very likely that this matrix transpose implementation is not optimal, as it performs worse than expected. However, this does not seem to be the only factor. It is interesting to note that for large scale factors Version 4 starts significantly outperforming Version 3 (Figure 6), and that the kernel splitting becomes very apparent in the power consumption when varying the number of rows (Figure 5), moving in an almost zig-zag pattern as the row number increases.

6.3 Pattern matching

Pattern matching is well suited for GPU implementation, with all GPU versions greatly outperforming the CPU implementation. As can be seen in Table 4 the GPU speedup is greater on the TX1 than the TX2, due to the TX2 CPU being relatively more powerful on the TX2 than on the TX1. This makes GPU implementations more worthwhile on the TX1. However the TX2 is both faster and more energy efficient compared to the TX1 for all versions. Figure 9 shows that the TX2 is about 20% more efficient in FLOPS per watt and throughput per watt than the TX1, running Version 4.

There is a crucial aspect regarding the possible parallelization of pattern matching, namely the synchronization needed before the brute force search for best gain can be started. Before this step all threads need access to the best shift, which is found in the first step. So, due to this synchronization there are really two separate problems for every pattern.

The TX2's feature to change clock frequencies on the GPU and CPU with different power modes makes a clear difference on computation time and energy efficiency, as can be seen in Table 5. Power mode 0 yielded the shortest execution time but was the least energy efficient. Power mode 4 on the other hand drains least energy but is the second slowest to compute.

Optimizing the code leads to a shorter total execution time and total energy consumption as can be seen in Figure 8. Many of the arguments presented in the QR-factorization discussion (Section 6.1) also hold true here. Working with shared memory (v2) instead of global memory (v1) allows the GPU to spend more time computing instead of waiting for global memory fetches, so more operations can be done per second per watt. For the same reason, working with warp functionality proves to be even more efficient (v3 and v4) compared to shared memory. The final version also has more workload per thread (fewer total threads) and at the end of the day, there are only 256 GPU cores on the TX1 and TX2. It makes sense to increase computational load per thread for large input sizes, or else the GPU will waste time switching threads in and out performing very simple computations, leading a to large relative overhead.

7 Conclusions

As expected the TX2 proved to be more effective at pattern matching than the TX1, being around 20% more efficient overall. Choosing the low power modes on the TX2 also proved to be both more power and energy efficient than choosing high performance, with a slightly longer computation time as a trade-off.

Drawing on the discussions of the benchmarks of each algorithm, it is safe to say that the energy efficiency of an algorithm is highly dependent on its computation time, which in turn is highly dependent on the quality of its implementation, with poor implementations being less energy efficient. The time and effort spent writing highly parallel code for an algorithm dedicated to a specific platform will therefore directly correlate with its efficiency.

That said, in the implementations presented the efficiency is almost always greatest for smaller problem sizes. However this is only truly valid in the case of bicubic interpolation due to the poor accuracy of the timing functions for smaller problem sizes. In the future, a more accurate measurement should be acquired using the Nvidia visual profiler instead of the build-in CUDA software timers to avoid this problem, as it likely gives better timing estimates in these cases.

The most energy efficient algorithm rarely uses the least power. A slow algorithm implementation which poorly utilizes the GPU consumes less power than a fast one, resulting in lower power consumption, but uses more energy in total due to the longer compute time. If low-power usage is important to the application and this behaviour is desirable then the practical way of achieving this would not be by writing subpar code but to utilize different power modes or manually under-clocking the CPU/GPU.

In general, the performance of an implementation should not be evaluated solely based on one metric alone. Instead, a variety of performance metrics should be considered when evaluating the efficiency of a particular implementation. Knowledge about the CUDA API and general GPU architecture is also imperative in order to write highly optimized code.

References

[1] About CUDA, https://developer.nvidia.com/about-cuda

[2] ARM CPU architecture, https://developer.arm.com/products/architecture/cpu-architecture

[3] Björck, Å. BIT (1967) 7: 1. https://doi.org/10.1007/BF01934122

[4] CUDA-Enabled GPUs, https://developer.nvidia.com/cuda-gpus

[5] History of GPGPUs, https://www.nvidia.com/content/gtc-2010/pdfs/2275_gtc2010.pdf

[6] High Performance Consulting, http://www.hpcsweden.se/

[7] The HPEC Challenge Benchmark Suite, http://www.omgwiki.org/hpec/files/hpec-challenge/index.html

[8] JavaScript Object Notation (JSON), https://www.json.org/

[9] Jetson TX1, https://developer.nvidia.com/embedded/buy/jetson-tx1

[10] Jetson TX2 OEM Product Design Guide, version 2018-10-09

[11] Nvidia CUDA C Programming Guide, version 9.0, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[12] Open Computing Language (OpenCL), https://www.khronos.org/opencl/

[13] Pettersson, J., Wainwright, I. (2010). *Radar Signal Processing with Graphics Processors (GPUs)* (Master's thesis). Uppsala Universitet, Sweden. Retrieved from http://www.diva-portal.org.