Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Ηλεκτρονικής

Τίτλος διπλωματικής

Διπλωματική Εργασία
του
Κωνσταντίνος Χατζηαντωνίου

Επιβλέπων: Νικόλαος Πιτσιάνης
Καθηγητής Α.Π.Θ.

23 Οκτωβρίου 2020

Abstract

General Purpose

Abstract

Empty

Ευχαριστίες

Άδειο

Τίτλος διπλωματικής

Όνομα Επίθετο
empty@auth.gr

23 Οκτωβρίου 2020

Contents

List of Figures

Chapter 1

Introduction

Thoughts The main focus of this chapter, is to describe C CUDA as the default language for GPU application, so we can then introduce Julia as the new competitive alternative. Then the goals of the thesis:

- Describe the 2 case studies to compare performance and code quality.
- Show how Julia code can be a whole lot simpler without the boilerplate code and how it can solve the multi language problem, but also show how more spefic cases (like the use of "streams" or memcpy part of the array can be equally complex as C, or even more because of the lack of documentation and examples.
- Develop a framework for stencil computations, that points out Julia's strengths (metaprogramming, data visualization with the same language, domain specific interface for multi gpu stencil computations, similar perfomance with C).

==========

General purpose computing on Graphics Processing Units is increasingly used for data-parallel applications and for applications with inherently parallel computations. "Such applications fall into a variety of problem domains, including quantum chemistry, molecular dynamics, computational fluid dynamics, quantum physics, finite element analysis, astrophysics, bioinformatics, computer vision, video transcoding, speech recognition, computed tomography, and computational modeling." One of the most popular ways create parallel programs is to use the CUDA parallel computing API. The most widespread languages for developing CUDA applications are C/C++ [3]

1.1 C/C++ for CUDA

Although the low level nature of C/C++ allows the user to be closer to the hardware they are targeting and exploit its capabilities, the low level nature is also an obstacle. Longer development times and multi-language usage for larger projects are some of the disadvantages of low-level languages.

1.2 Julia for GPU

Julia is a high-level programming language for mathematical computing... In 2018 support for NVIDIA GPUs was added to the language... Results from the Rodinia benchmark show that Julia can be used to write GPU code with similar performance to CUDA C.

1.3 Goal of the thesis

The goal of the thesis can be divided in two smaller goals

- Verify the results of Rodinia benchmarks by comparing Julia and C code for GPU for two different problems and study the code quality, performance and the metrics (provided by CUDA toolkit)
- Implement a fully-featured application that takes advantage of powerful tools provided by Julia and targets multi-GPU systems.

Chapter 2

Background

Thoughts This chapter should contain everything about CUDA, its architecture and the mindset when writing an application (independent of C or Julia), how C works internally (static, compiled) and how Julia works internally (dynamic, type inference, JIT, introspection). Some of the apparent pros and cons could be mentioned here (like the 2 language problem and code verbosity. Maybe other tools, like nvvp or nvprof should be described.

==========

This chapter will provide background knowledge of the tools, languages and frameworks that will be used throughout the thesis.

2.1 Julia

Julia is a...

2.1.1 JIT

How JIT provides similar performance to C and statically typed languages in general.

2.1.2 Introspection and Metaprogramming

Easy to generate code and parse code to generate functions.

2.2 GPU

2.2.1 NVIDIA GPU architecture

2.2.2 CUDA C

2.2.3 CUDA Julia

2.3 Code Quality

C Verbosity, both for CUDA kernels and GPU communication.

Mixed languages (e.g. C for CUDA kernel and python for Data Visualization)

Chapter 3

Case Studies

Thoughts I believe this chapter is kinda complete (except from the comments part that could use more opinions. Long story short, the case studies show different results: Julia is significantly faster in the first and significantly slower in the second. **In the second chapter (Background) i mentioned something about the code quality. I believe that presenting some minimal examples about CUDA streams, and 2 language problem is also relevant with this chapter**.

==========

Comparing the performance and code quality of Julia and C, is the first step to solidify Julia as a valid or even better option for CUDA kernel programming. Although there is already a benchmark set that indicates Julia's good performance [rodinia], going through new implementations from zero can help gain contextual and in-depth knowledge about the process of writing Julia code and allows easier manipulation of the source code to explore more technicalities.

The following case-studies, not only compare Julia with C, but also compare the effect of using Julia's higher level syntax inside the kernel function. For the two case-studies, one dense and one sparse problem were chosen and implemented in multiple ways to cover as many characteristics as possible.

3.1 Case Study 1 - Grid KNN

3.1.1 Problem Description

Grid KNN is a specialised version of the KNN algorithm for 3d points. The 3d points (data and queries) are binned into blocks of equal size that form the 3d grid. To find the nearest neighbour of a query, first the block that it belongs to is searched, then the adjacent blocks.

3.1.2 Implementation Details

Only the case for k = 1 is implemented. There are two approaches for searching adjacent blocks· either search every adjacent block (named "simple") or check if the current nearest neighbour's distance is farther than the boundaries of the block

(named "with skip"). The "skip" happens when all the points that a warp is processing currently, have a neighbour closer than the boundaries of the box.

For each language the implementations are identical from the algorithmic perspective. For Julia, the "simple" implementation has 3 versions:

- "jl_simple" Identical to C counterpart. Has type declaration for each variable. The only difference is 3d arrays instead of 1d.
- "jl_view" Similar to "jl_simple" but uses @view macro to avoid offset usage throughout the kernel.
- "jl_no_types" Similar to "jl_view" but without type declaration for variables.

The Julia versions "with skip" follow the same naming scheme.

3.1.3 Code quality

Variable Declaration

Julia doesn't require type declaration for its variables to work. This results in less *clattered* code.

Listing 3.1: C example with type declaration

```
int tid = threadIdx.x + threadIdx.y*blockDim.x;
int stride = blockDim.x*blockDim.y;
int start_points = intgr_points_per_block[p_bid];
int start_queries = intgr_queries_per_block[q_bid];
```

Listing 3.2: Julia example without type declaration

```
tid = threadIdx().x + (threadIdx().y-1)*blockDim().x
stride = (blockDim().x)*(blockDim().y)
startPoints = IntPointsperblock[p_bid]
startQueries = IntQueriesperblock[q_bid]
```

One would except type declaration to be an important part of the kernel programming. 64-bit arithmetic instructions can be 8 times slower than 32-bit instructions[CUDA C Programming]. Although Julia's inference system defaults all numbers to either Int64 or Float64 for CPU code, this is not the case for CUDA kernels, as the results below show.

Array Access

2d Indexing Julia can use multi-dimensional indexing for device arrays, static and dynamic shared memory arrays. In contrast, C uses linear indexing in all cases, except for static shared memory arrays· there is the option to use an array of pointers.

Using multi-dimensional indexing makes the code *nicer*, more readable and less prone to logical errors.

Listing 3.3: C example accessing array

```
int q_index = q + tid + start_queries;
if(tid + q < total_queries){
    for(int d = 0; d < dimensions; d++){
        sh_queries[tid + d*stride]
                = queries[q_index + d*num_of_queries];
    }
    distance = distsances[q_index];
    neighbour = neighbours[q_index];
}
```

Listing 3.4: Julia example accessing array

```
qIndex = startQueries + q + tid
if tid + q <= totalQueries
    for d = 1:dimensions
       @inbounds SharedQueries[d,tid] = Queries[qIndex, d]
    end
    @inbounds dist = Distances[qIndex]
    @inbounds nb = Neighbours[qIndex]
end
```

Pointer to row/column Julia's multi-dimensional indexing can be enhanced by the use of the @views macro. This allows the definition of a subarray on any dimensions.

C can achieve a similar effect by using pointers to denote the start of a row/column, but is limited by the row-wise or column-wise topology of data in memory.

Listing 3.5: Julia example with @view macro. Only the thread id (tid) and loop variables are used for array accesses.

```
# Defining the views
@inbounds Queries = @view devQueries[
        (startQueries+1):(startQueries+totalQueries), :]
@inbounds query = @view SharedQueries[:, tid]
@inbounds Distances = @view devDistances[
        (startQueries+1):(startQueries+totalQueries)]
@inbounds Neighbours = @view devNeighbours[
        (startQueries+1):(startQueries+totalQueries)]

...
# Reading Queries from Global Memory
 if tid + q <= totalQueries
    for d = 1:dimensions
        @inbounds  query[d] = Queries[tid+q, d]
    end
    @inbounds dist = Distances[tid+q]
    @inbounds nb = Neighbours[tid+q]
```

```
end
```

It's common in CUDA kernels, each block to operate on part of the data. So, instead of using an offset in every array access, @view can specify the subarray that the block will work with. 2D indexing along with @view can reduce the cognitive load for the person who writes or reads the code, as they will not have to keep track of the offset and any complicated linear indexing.

3.1.4 Performance and Metrics

The execution time for each implementation was measured on NVIDIA's GPUs Tesla T4 and P100 for 2^{18} 3d points with 2^3 and 2^4 grid size, and for 2^20 3d points with 2^4 and 2^5 grid size.

The speed up here, is defined as $\frac{\text{C time}}{\text{Julia version' time}}$ for a given problem size.

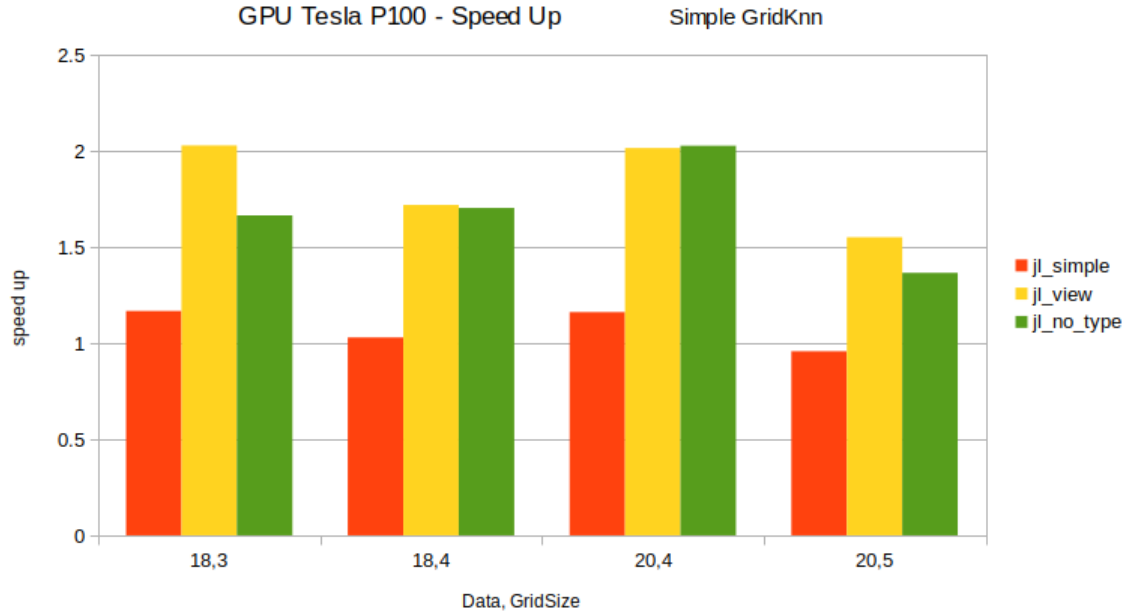

Figure 3.1: Comparison of Julia over C for simple GridKnn

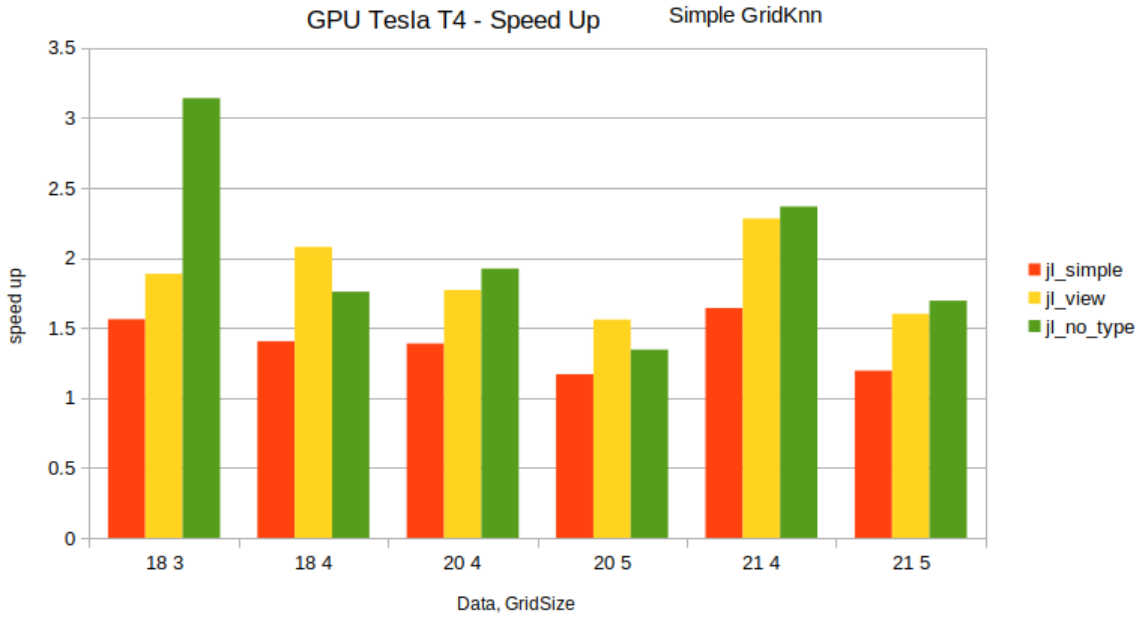

Figure 3.2: Comparison of Julia over C for simple GridKnn

At first glance, the C program is always slower than at least one of the Julia's versions. Most of the Julia's versions are close to 2 times faster and they manage to achieve 3x speed up.

For the version with skip Julia still achieves better times, but the gap is not as big as for the simple version.

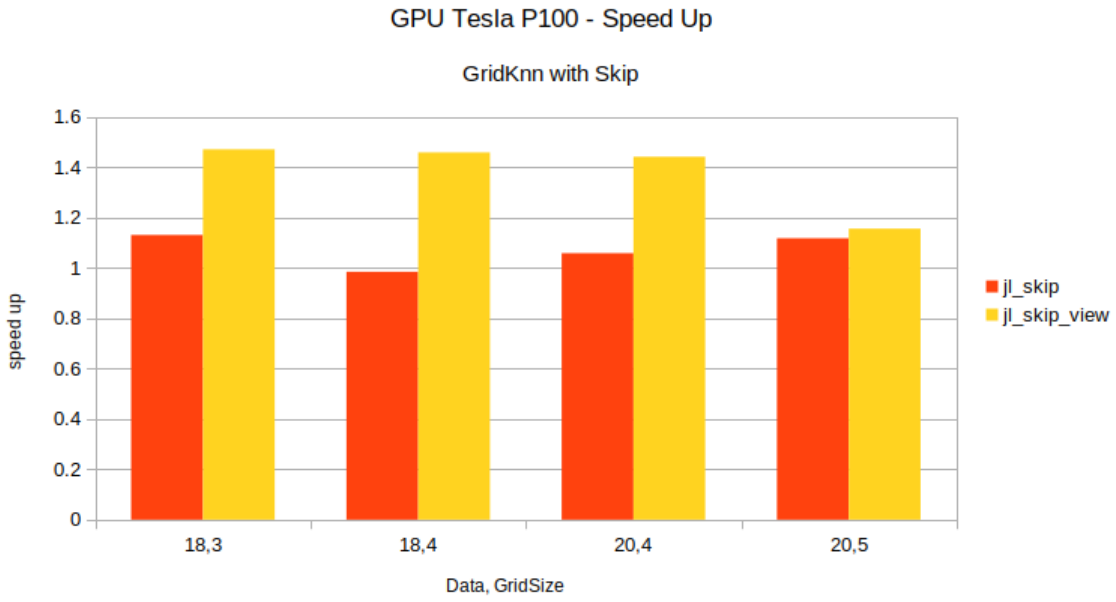

Figure 3.3: Comparison of Julia over C for GridKnn with Skip

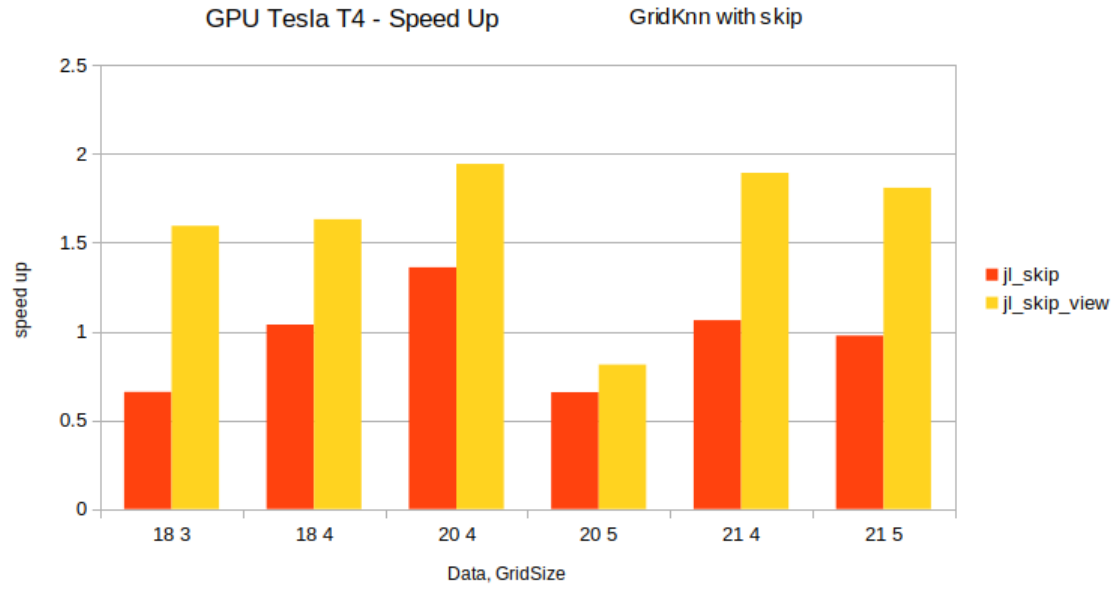

Figure 3.4: Comparison of Julia over C for GridKnn with skip

It is interesting how a dynamic, higher level language can be faster than a static, low-level language on a task that is closely connected on the hardware. Even more interesting though, is how seemingly identical source codes can have different PTX assembly, as shown by the metrics collected.

The 3 basic metrics, that are used to further optimize kernels, are identical along every version and language. Those metrics are *Global Memory Read Efficiency*, *Global Memory Write Efficiency* and *Shared Memory Efficiency*. Those 3 metrics are associated with the data access pattern. For the Global Memory, reads and writes must be coallesced and for the shared memory bank conflicts must be avoided. It was

expected those metrics to have the exactly same value.

Register usage is the first sign that the assembly codes are different.

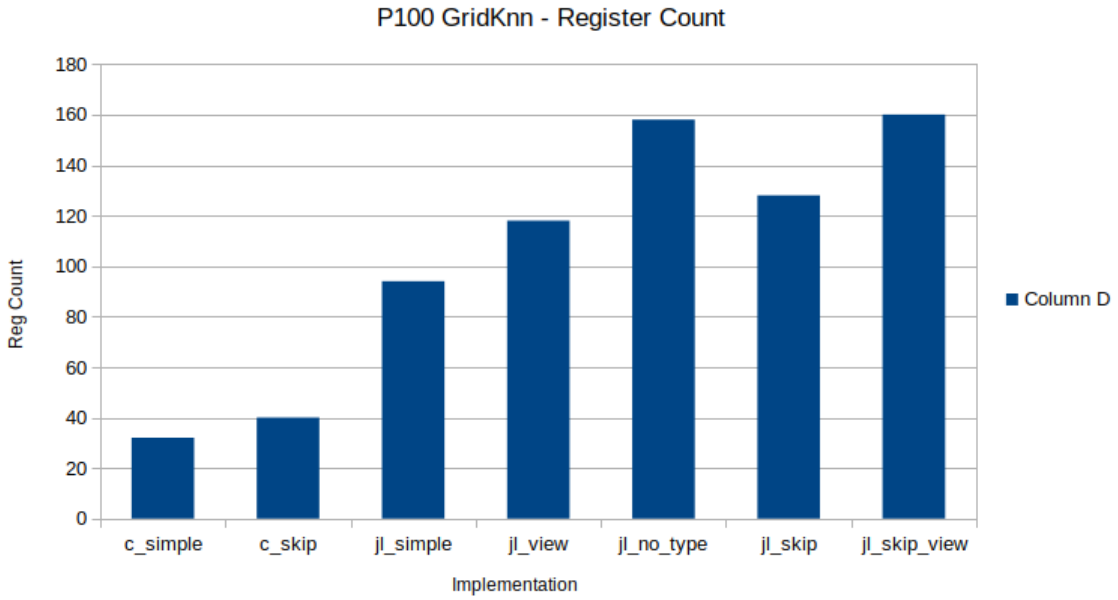

Figure 3.5: Register Usage for GridKnn implementations.

[CUDA C programming guide] states that the fewer registers a kernel uses, the more threads and thread blocks are likely to reside on a multiprocessor, which can improve performance. Although the first part of the claim is true in our case (C programs have higher occupancy), they are not faster.

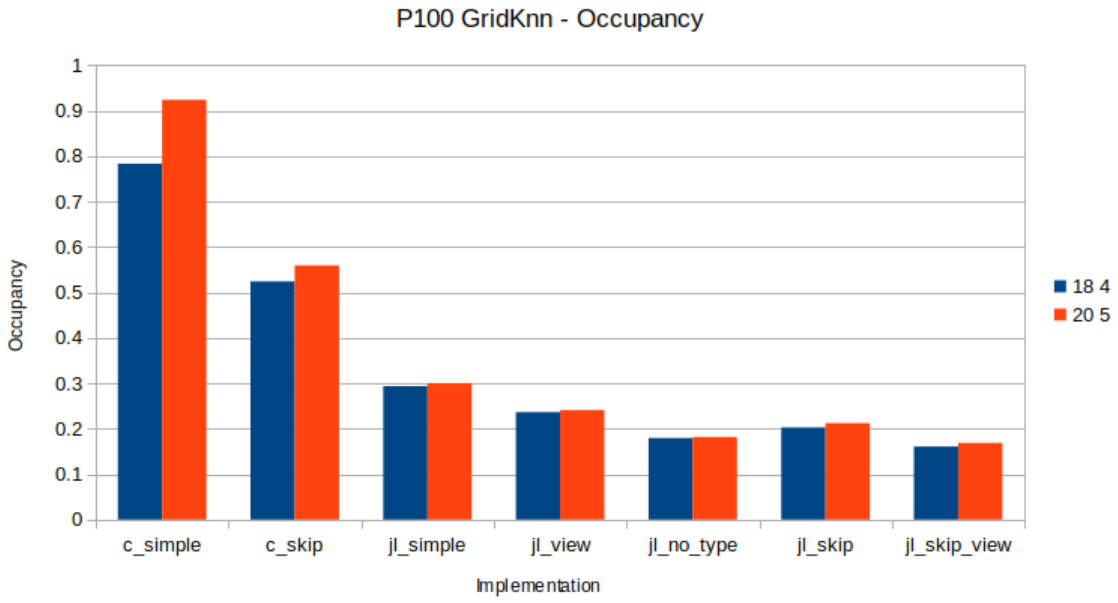

Figure 3.6: Register Usage for GridKnn implementations.

Another significant difference in metrics, was for L2 cache write throughput. L2 cache is mainly used to cache accesses to global and local memory. While C implementation had a couple hundreds MBps L2 cache write throughput, Julia implementations had a couple GBps throughput.

Although the variance in metrics is significant, it is not informative about the way the produced assemblies differ.

Last but not least, it should be noted how the usage of the @view macro affects performance.

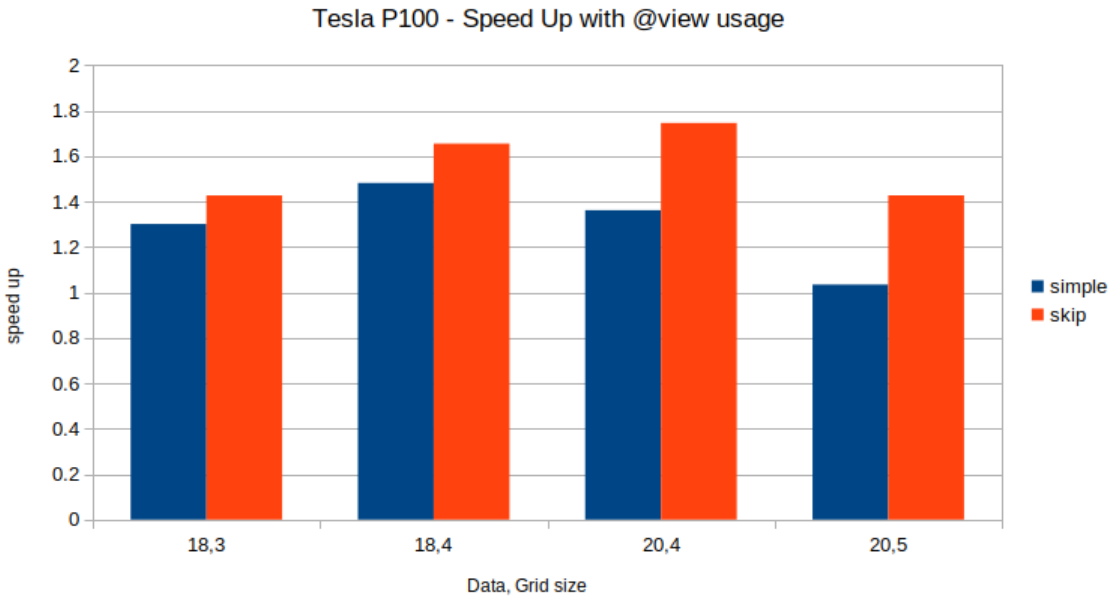

Figure 3.7: Julia - Speed Up with the use of @view.

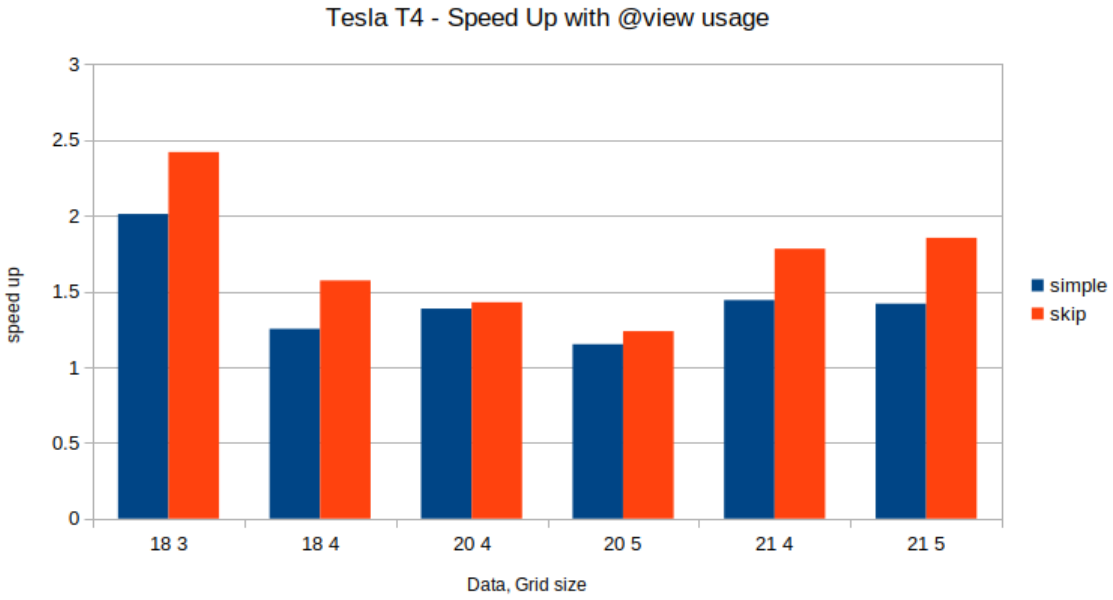

Figure 3.8: Julia - Speed Up with the use of @view.

In both versions, "simple" and "with skip", the use of macro improves performance.

3.2 Case Study 2 - Counting Graph Triangles

3.2.1 Problem Description

The number of triangles of a symmetric undirected graph can be counted using the adjacency matrix A and the formula $n = \frac{1}{6}\sum_{ij}(A \cdot A) \odot A$. Using the properties of symmetry of A the formula can be reduced to $n = \sum_{ij}(U^T \cdot U) \odot U$ where U is the upper triangle of the adjacency matrix. The last formula, reduces the problem to a simple comparison of the rows and columns of U, pointed by the non-zero cells of U itself.

3.2.2 Implementation Details

There are 3 different versions, each with its own disadvantages:

- "row_per_th" Each thread is assigned a row and compares its element with every column.(Disadvantages: lot of shared memory, warp divergence, low occupancy)
- "elem_per_th" All of the rows are stored in shared memory. Every thread is responsible for one of the elements in shared memory and compares the element with every column.(Disadvantages: lot of shared memory, more computations)
- ""elem_per_th_limit" The same as "elem_per_th", but only part of the rows is stored each time, to allow computation of triangles for sparse arrays with many elements per row.(Disadvantages: more computations)

Each of these versions was written 2 times, with and without type declaration.

3.2.3 Code quality

Variable Declaration

In this problem, not declaring the type of variables caused low performance. As shown by the following code snippets, the Julia and C code have the same level of simplicity and verbosity.

Listing 3.6: C example type declaration.

```
// ------- Read all the columns -------
for(int i = 0; i < len; i++){
    int col = sh_row[i];
    start_row = csr_rows[col];
    end_row = col==(rows-1)? nnz : csr_rows[col+1];
    int templen = end_row - start_row;
    if(tid == 0)
        sh_len[i] = templen;
    if(tid < templen){
        sh_cols[i*stride + tid] = col_indx[start_row + tid];
```

```
    }
}
```

Listing 3.7: Julia example type declaration

```
 # #Read all the other rows
for i::Int32 = 1:len
    if threadIdx().x == 1
        @inbounds col::Int32 = sh_row[i]
        @inbounds row_start = csr_rows[col]
        @inbounds row_end =
            (col == rows) ? nnz+Int32(1) : csr_rows[col + 1]
    end
    row_start = shfl_sync(0xffffffff, row_start, 1)
    row_end = shfl_sync(0xffffffff, row_end, 1)
    templen::Int32 =  row_end-row_start
    (tid==1) && (sh_len[i] = templen)
    if tid  <= templen
        @inbounds sh_cols[i,tid] = col_indx[row_start+tid-1]
    end
end
```

Array Access

The sparse adjacency matrix is stored in CSR (Compressed Sparse Rows) format. So every array used is 1D. Julia can't benefit from mutli-dimensional indexing.

Also, in this case, C can use pointers to denote the part of the row the block needs. So Julia doesn't benefit from the use of @view macro, in terms of code quality.

3.2.4 Performance and Metrics

The C and Julia with and without type declaration were run on a Tesla K80 GPU, to measure execution time and collect metrics. Also, the C and Julia with type declaration were run on a Tesla T4 GPU to collect additional execution times. The 3 datasets that were used, had varying size, average and maximum non zero elements per row.

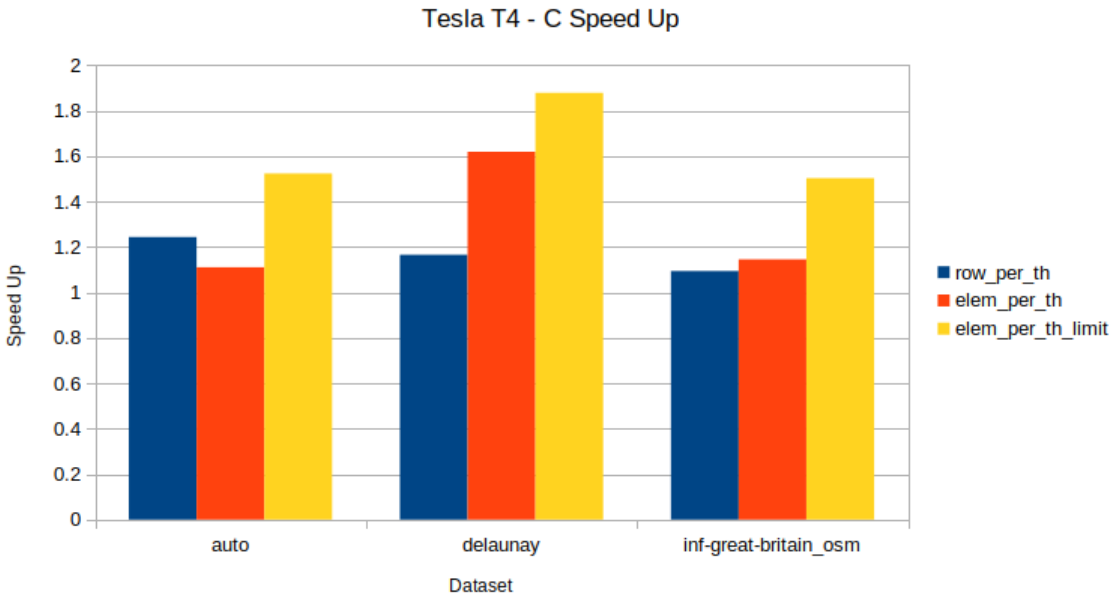

Figure 3.9: Julia - Speed Up with the use of @view.

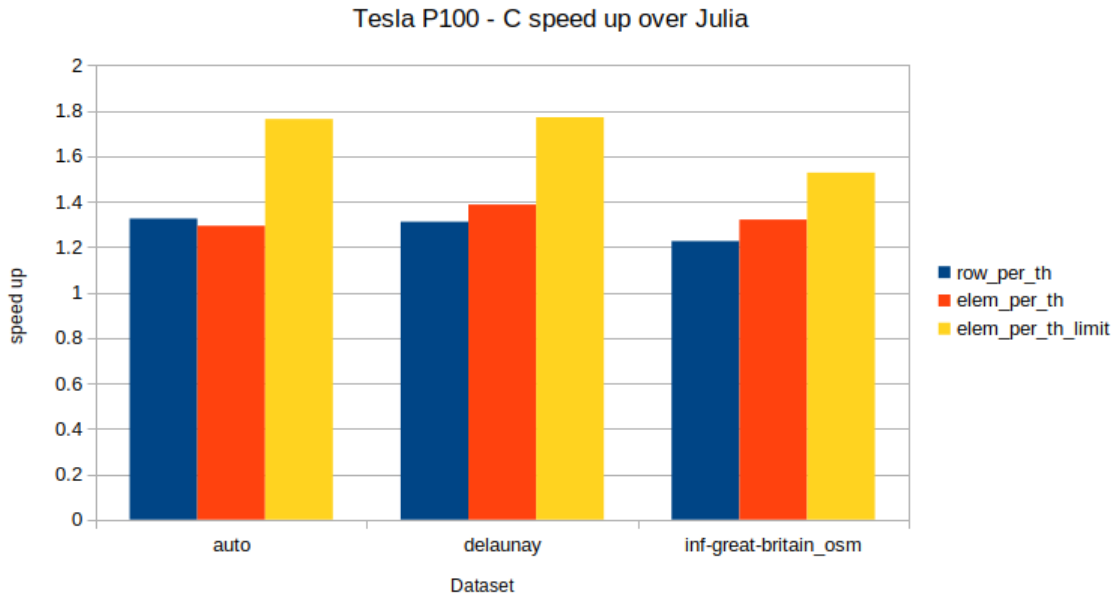

Figure 3.10: Julia - Speed Up with the use of @view.

Is is faster than Julia in every version. In most cases the speed up is less than 1.3x. Julia performs badly for the 3rd version "elem_per_th_limit", where C is close to 1.8x faster in both GPUs and all datasets.

Once again the 3 basic metrics, *Global Memory Read Efficiency*, *Global Memory Write Efficiency* and *Share Memory Efficiency* are identical for both languages per version.

Similarly, Julia has higher register count. Not declaring the types further increases register usage.

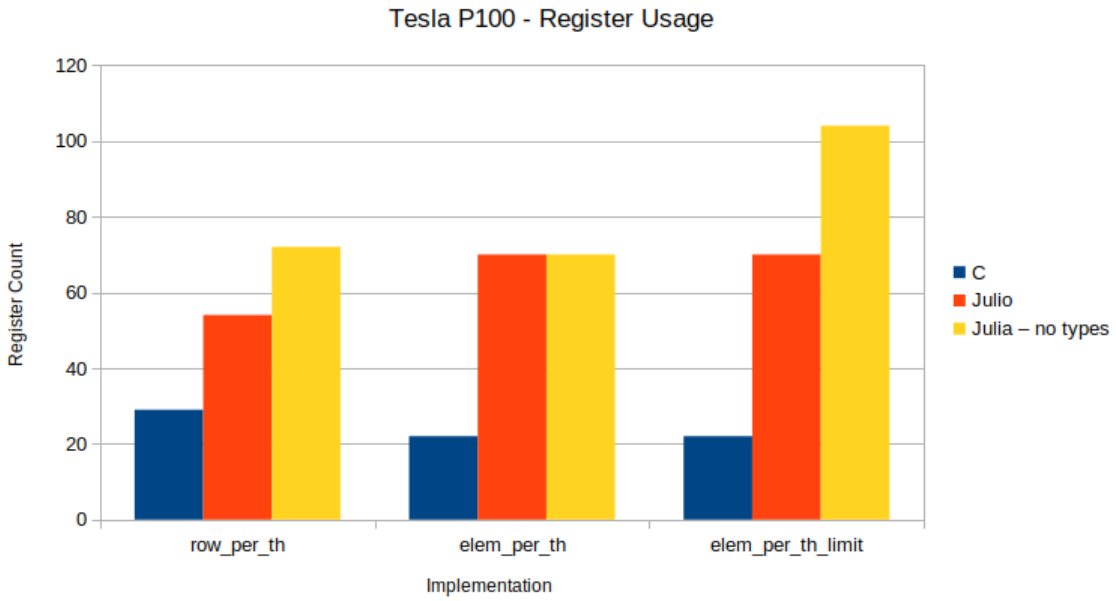

Figure 3.11: Julia - Speed Up with the use of @view.

As stated in the Code Quality section, not declaring types reduces performance in all versions.

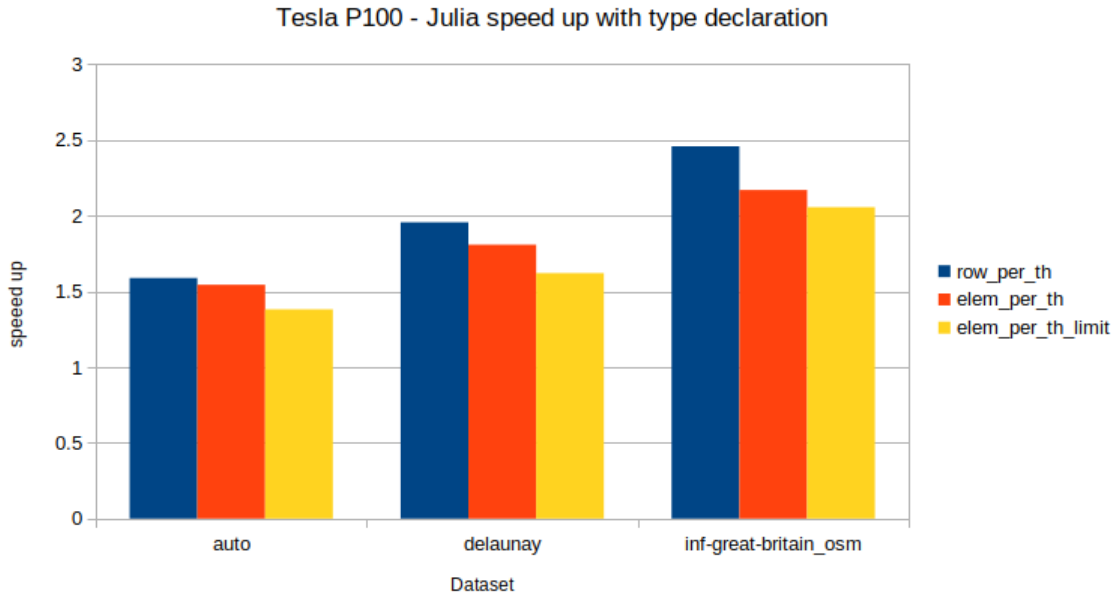

Figure 3.12: Julia - Speed Up with the use of @view.

Declaring the type, achieves $> 1.5x$ speed up in almost every case. This contradicts the result from GridKNN where the kernel without types was notably faster.

Lastly, it is worth mentioning the L2 cache write throughput. All C versions and the Julia version with type declaration have 100-300 Mb/s throughput, whereas Julia without types goes as far as 4.8 Gb/s, a 12x increase at least.

3.3 Comments

The case studies didn't converge to a single "recipe" for writing fast Julia code. For the first case, not declaring types increased performance, while for the second, the opposite happened. While Julia was a lot faster for the GridKNN, on the other hand was significantly slower for triangle counting.

It is noteworthy however, that the use of @view macro, a feature of a high level language, that contributes in code quality, could also help increase performance.

A topic for further investigation ,could be the reason Julia has higher register usage, how they are being utilized and how it affects performance.

As for the rest of the code, besides the CUDA kernel, Julia seems to be better. C requires more boilerplate code to communicate and transfer data to the GPU, and requires more work to preprocess the data. Julia higher level syntax and math functions make the process a lot easier and faster, without trading performance.

Chapter 4

Stencil Computations

4.1 Stencil Definition/Description

Thoughts This chapter should contain a complete description of stencil computations and any previous research/implementations. I'm sceptical on what to include in this chapter as it overlaps with the next one (Stencil kernel generation on multi gpu cluster framework). I have read and downloaded enough papers about stencils to fill this chapter but I'm not sure which ones are relevant enough. I already know which ones are relevant to my implementations, but maybe more papers should be mentioned here. Examples of papers that i think are not relevant enough: Cache oblivious stencil, any paper that has stencil for multicore CPUs and GPUs...

==========

Stencil codes are a class of iterative kernels used in a wide variety of scientific applications.

Stencil computations involve a sequence of sweeps (time-steps) over an array of data, updating every element of the array at each step according to a function of some neighboring elements. The geometric shape of that neighborhood comprises the stencil.

Stencil computations can be used to either simulate multiple time-steps of a system or to determine whether the system converges. Stencil codes can be used to solve any problem that involves Finite Difference Methods to numerical solve PDEs (Partial Differential Equation), like heat diffusion, as well as in image processing, cellular automata and more scientific domains.

Stencils applied on a single regular grid are the simplest, with adaptations to unstructured and/or multi-grid problems being more complex.

Since computing time and memory consumption grow linearly with the number of array elements, parallel implementations of stencil computations are of paramount importance to research [4].

An indicative example of stencil application, is the 3D Discrete Laplacian

$$D^{t+1}[x,y,z] = aD^t[x,y,z] + \beta(D^t[x-1,y,z] + D^t[x+1,y,z] + D^t[x,y-1,z] + D^t[x,y+1,z] + D^t[x,y,z-1] + D^t[x,y,z+1])$$

an analog to the continuous Laplace Operator: $\nabla = \Delta^2 = (\frac{\partial^2}{\partial^2 x} + \frac{\partial^2}{\partial^2 y} + \frac{\partial^2}{\partial^2 z})$.

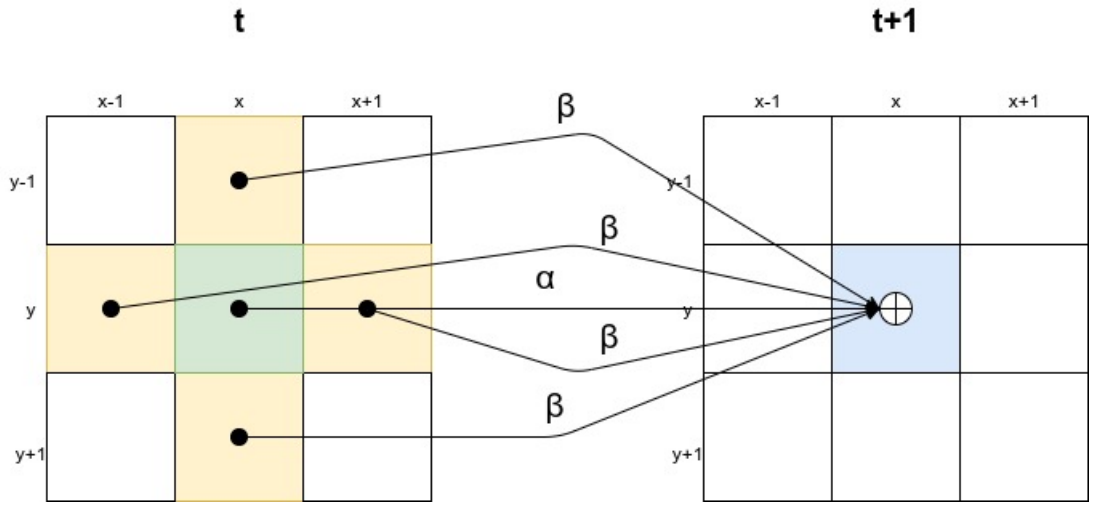

Figure 4.1: Visualization of stencil for one element/time step for 2D discrete Laplacian .

4.2 Stencil on GPUs

Stencil computations are an inherently parallel problem. The value of each element at the next time step can be calculated individually from the others. This makes GPUs a suitable candidate for acceleration. However, exploiting the inherent parallelism doesn't lead to the most efficient solution. [4] Stencil calculations perform on much larger data-sets than the available data-caches. Additionally, data re-usability is restricted by the small number of stencils points, making them memory bandwidth-bound in multi-core systems. [2] Reorganizing those stencil to take full advantage of memory hierarchies, along with other optimization techniques, has been the subject of research over the years.

4.3 Optimizations

4.3.1 Data Access Pattern

[5] Memory optimizations are the most important area of performance, and since stencil computations are memory-bandwidth bound such optimizations will have significant impact. Reading and writing to GPU memory, both global and shared, is slow and particular access patterns are required to fully utilize the available bandwidth.

For 2D stencils, the case is trivial. A thread block reads a 2D tile and halos in shared memory, taking advantage of coalesced reads and data re-usability by neighboring

threads. Afterwards, each threads their center element of the stencil, or neighbouring one with the offset being constant to all threads, avoiding bank conflicts without any effort.

For 3D stencils, data access is more difficult. Following the same rationale, a 3D block should be read in shared memory. Theoretically, coalesced reads can be utilized and bank conflicts are avoided, the amount of shared memory requested is too large, causing a decrease in occupancy or even exceed available resources. Micikevicous [6] gave an implementation of 3D stencil in which the shared memory is used to load XY-tiles and registers are made available to save shared memory space. However, his code is focused on 3D Finite Differences and consequently can not be applied on any stencil type rather than the star stencil. Mo et al. [7] present a similar implementation uses registers to store partial results and not data. Apart from being more efficient memory-reading-wise, it can also be applied on stencils of arbitrary shape.

4.3.2 Array Padding

[5] Flow control instructions can significantly affect the instruction throughput by causing threads of the same warp to diverge; that is, to follow different execution paths that are executed separately. In stencil computations, warp divergence can appear when checking for in-bound data accesses. In order to avoid this phenomenon, input data arrays are padded according to the radius of the stencil, eliminating the possibility a thread trying to access out of bound memory without specifically checking it. Data can be further padded to be an integer multiple of the thread block dimensions, so that thread-blocks at the edge don't have to check about idle threads.

4.3.3 Temporal Blocking

Temporal blocking is an optimization technique that aims to enhance data-locality by reusing data that are already loaded in local memory, to compute multiple time-steps.

Temporal blocking has been successfully applied for 1D and 2D stencils on a variety of architecture, single core [8], multi threaded [9] and GPUs [2] [1] achieving an indicative 5x speed up. For GPUs, this technique is implemented by iterating over the shared memory data multiple times before storing them to global memory, effectively reducing the global memory accesses.

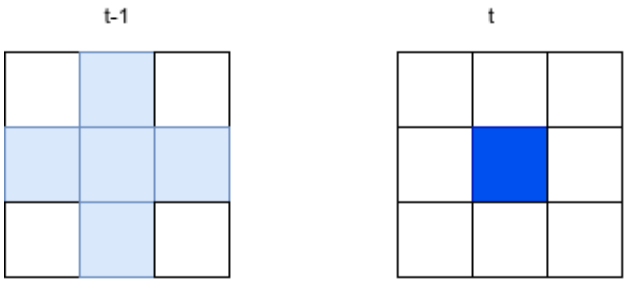

Figure 4.2: One time step for the center cell. No additional elements are required.

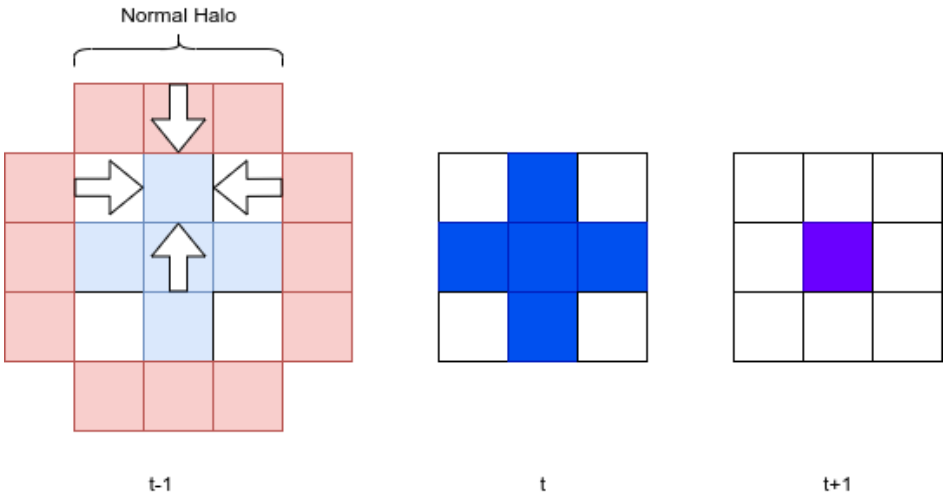

Figure 4.3: Two time steps for the center cell.The additional elements are already loaded as the normal halo.

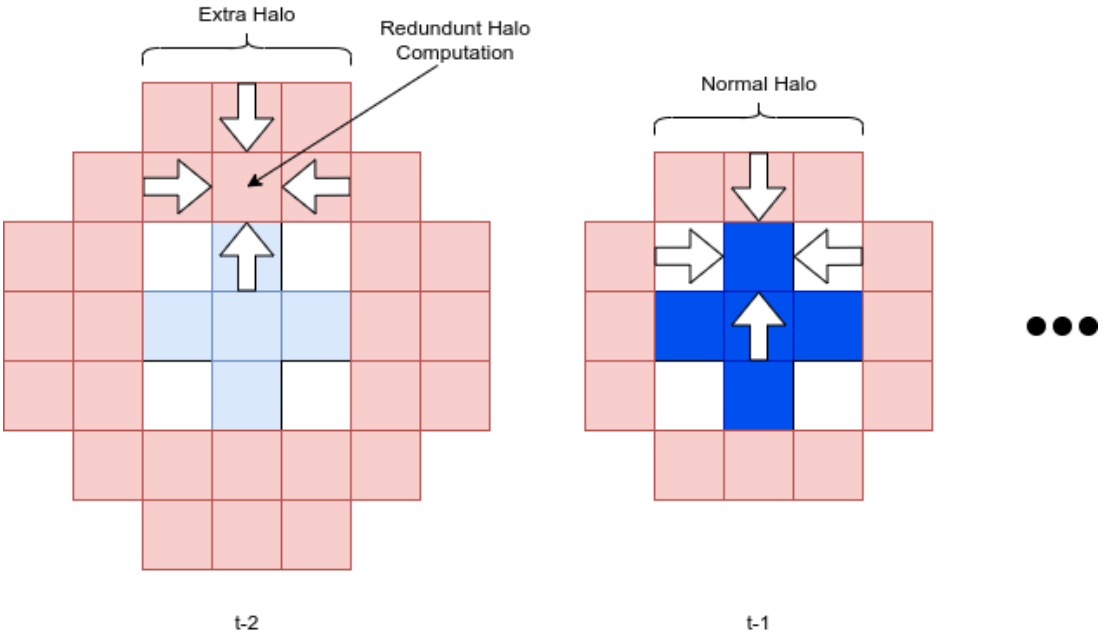

Figure 4.4: Three time steps for the center cell. Additional halo read. Redundant computations to update the halo value.

Figure 4.2 shows the elements needed to update the center cell for one time step. Two time steps are applied in Figure 4.3. The cross needs to be updated before the center cell can be updated for two time steps. The cross uses elements from the halo for the first step. For three time steps, in Figure 4.4, the halo needs to be updated before proceeding. It is obvious that as time steps increase, more halos needs to be loaded. Also, the halo needs to be updated. This is redundant, because neighboring thread-blocks have those halos as their basic elements and they are responsible for updating and storing their final value. Finally, the more time steps are merged, more threads do only redundant computations, less threads participate in the basic block and more thread-blocks are needed to cover the whole data.

It's necessary to found a balance between the mentioned drawbacks and the reduction in global memory usage. This was the topic of research of Meng et al. [1]

For 3D stencils the shared memory usage and redundant calculations scale faster for more time-steps. Holewinski et al. [2] and Meng et al. [1] didn't achieve any kind of speed up with temporal blocking. Nguyen et al. [9] achieve x1.8 speed up with temporal blocking for a 7-point stencil, but their implementation [2] doesn't consider fully automatic high-performance code generation for stencil computation on GPUs.

4.4 Auto Tuning

Auto-tuning has been applied successfully for various kernels, ranging from sparse and dense linear algebra to signal processing. Programmers writing CUDA programs have to specify grid and block size along with in-kernel optimizations and may not have the intuition to select the best combination since there are complex and unpredictable interactions among optimizations [10] Auto-tuning is necessary to find the optimal configuration in an extensive and convoluted configuration space. For stencil computation multiple auto-tuning frameworks have been proposed [4] [10] [11] [12], that not only configure CUDA launch parameters, but also in-kernel optimizations, such as temporal blocking, loop unrolling etc. Those frameworks use diverse techniques to achieve their goal, parameterized configuration space, machine learning and heuristic approaches.

The most noteworthy approach is that of Lim et al. [10] where they statically analyze the PTX assembly by NVCC to estimate metrics and configure the parameters without the time cost of running the application.

4.5 Automatic Stencil Generation

Automatic parallel code generation in an important topic of research that aims to abstract away the acceleration procedure from domain experts. Frameworks like LibGeoDecomp [13], PATUS [14] and Holewinksi's [2] create efficient parallel code, for multiple architectures including GPU, and incorporating in-kernel optimizations with great success. The user gives the problem/stencil definition in a small domain specific languages, which is then parsed, converted to parallel code and executed with the given data. It is noteworthy, that all these frameworks use external frameworks/libraries written in different languages to parse, generate and execute the code, leading to a complex infrastructure.

Chapter 5

Stencil kernel generation and multi-gpu framework.

Thoughts I will describe everything here so you don't have to read the sketchy and incomplete chapters below. The "framework" I wrote, takes as input one ore more mathematical descriptions of an isomorphic stencil, or a 3D coefficients matrices that describe any type of stencil, parses them with Julia's introspection, and creates a CUDA kernel. Then a function can be called to apply the kernel to the input data. I also have created a version where the data is split amongst different streams (I didn't have multiple gpus, so I just tested that the result is correct). For the stencil creation I used a modification of the NVIDIA's kernel for Finite Forward Differences**[1]**. I found the same approach in an other paper**[2]**, also mentioned in the previous chapter. My implementation can handle the following types of stencil.

1. $D^t[x, y, z] = \sum_{ijk} c[|i|]D[x + i, y + j, z + k]$ This is the simplest stencil, isomorphic, that depends on a coefficients vector, and the current values.
2. $D^t[x, y, z] = aD^{t-1} + \sum_{ijk} c[|i|]D[x + i, y + j, z + k]$. This is the same stencil as above, but it also uses the past value of the current cell.
3. $D^t[x, y, z] = vsq[x, y, z] \sum_{ijk} c[|i|]D[x + i, y + j, z + k]$. This is also the same stencil as the first, but it uses a viscosity matrix, that can denote for example the viscosity of a liquid, or the thermal conductivity of the different materials in space.

The main focus, is the first type, the simplest stencil, for which I wrote a function to create a new stencil that combines multiple timesteps (a kind of temporal blocking). I included the other two types, because they were easy it implements and they are used in many applications. I didn't write a time combination function for the last two types, because it was too complicated and it may require more resources (more shared memory or global reads in case of the viscosity type). **I don't know whether to include the process and benchmarks of modifying the NVIDIA kernel here or in the previous chapter**. Also, there are more optimizations to include here or in the previous chapter (coalesced reads, shared memory fetching). Finally, although I found a number of papers on auto_tuning stencils, I will use empiric and expirimental results to set some parameters (CUDA block size, ratio of split data between GPUSs, time steps internal to kernels and time steps before multiple GPUs

communicate the halos. Enumerated References Explained.

1. I have already sent you a report on the results, and the implementation is briefly mentioned in the previous chapter.
2. Although they have the same approach, they benchmark against NVIDIA's implementation on a small stencil, for a particular GPU and data size, for which the achieve 1.83x speed up. They don't present any actual code, only pseudo code that abstracts most of the functions.

I was thinking i should put any implementation details in this chapter, and all the results, benchmarks and parameter tuning in the next, 6th chapter.

==========

5.1 Motivation

Stencils computations are used in a wide range domain of scientific domains. Creating a new kernel or modifying an existing one for the needs of each application is inefficient and error-prone.

Abstracting the parallelization from domain experts increases the efficiency of research, by reducing computational and development time and also reducing human error probability.

As mentioned in the previous chapter, previous implementations of automatic code generation have complex infrastructure, use external frameworks written in different programming languages. Additionally, user interface can be written in lower level languages such C/C++ that don't allow much expressiveness in the definition of the problem and require more boilerplate code.

5.2 Framework Overview and Purpose

The framework acts as an automatic stencil kernel generator, performing additional requested optimizations and applying the stencil to the given data in a single or multiple GPU. The purpose of this framework is not to invent a new technique or improve an existing one, but to point out that Julia is capable of performing the task without the need of external libraries. Specifically the goals are to show that Julia can:

- create an intuitive interface with minimal boilerplate
- easily parse user-input code
- generate efficient kernels and apply optimizations
- performs well on a single GPU context
- naturally utilize multiple GPUs efficiently

Apart from the good performance and reduced development cost Julia can provide, it is equally important to allow rapid and unhindered research of new techniques.

Although, it was mentioned this work is not focused on raising performance standards, two previously unstudied optimization techniques, regarding stencil, are presented:

- new temporal blocking implementation that favors non holistic approaches
- applying multiple time-steps of a stencil with FFT

Hopefully Julia can be proved a powerfully ally for prototyping and research purposes.

5.3 Framework Infrastructure

The framework, consists of 4 basic components. The *Parser*, that takes as input a Julia expression object and modifies it. The second component, transforms the modified expression to symbolic, which optionally can be further processed for temporal blocking. Then, the code generator takes the symbolic math, and converts it to arithmetic instructions. Finally, the kernel execution system, handles the data padding, data transfer between CPU and GPU(s) and the actual kernel execution.

It is important to note, that every component was written in Julia, without relying on external frameworks for parsing or code generation and compilation.

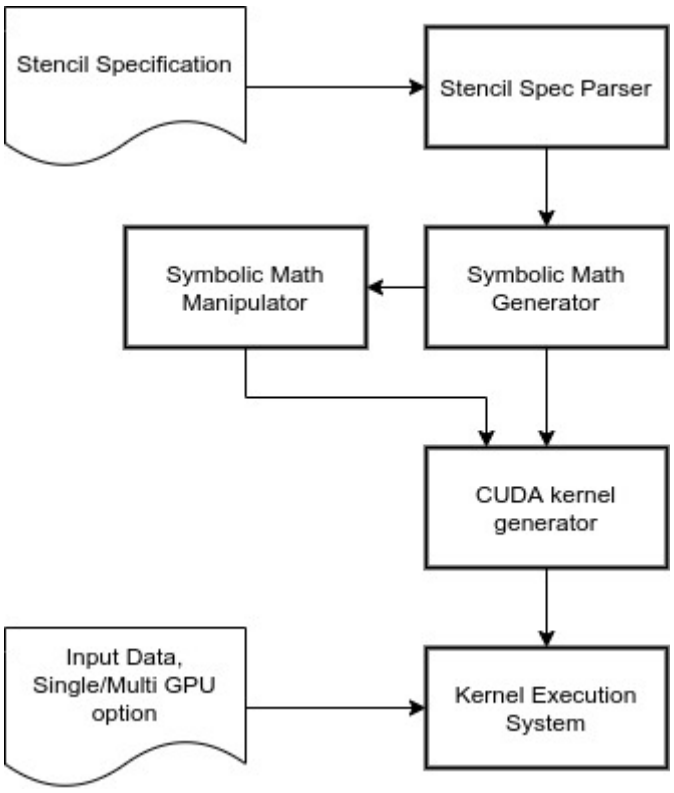

Figure 5.1: High level overview of the architecture.

Stencil Specification Parser The user can specify the stencil with two ways, either a 3d coefficient array or a "mathematical" expression. The case of the 3d coefficient array is very simple and doesn't require any parsing. The second case uses the

Julia's meta-programming and introspection capabilities to easily tackle the parsing problem.

The user can specify the stencil the following intuitive way:

Listing 5.1: Julia example defining star stencil

```
star_stencil = @def_stencil_expression c[0]D[x,y,z]
            + @sum(i, 1, 4, c[i]*(
            D[x+i,y,z] + D[x,y+i,z] + D[x,y,z+i] +
            D[x-i,y,z] + D[x,y-i,z] + D[x,y,z-i]))
```

The first thing the macro @define_stencil_expression does is to evaluate and substitute any internal functions and macros. In the above case, the macro @sum has to be evaluated. The @sum (included in the framework) replaces the given index, with the range of the 2 numbers to the expression. Julia can view its own expressions, represented in a tree-like structure. So it is simple to recursively search the expression for array indices, find the given symbol and replace the node with a number. The next step, is to transform the zero centered, both positive and negative indexing, to 1-base indexing. The same recursive procedure described before is followed for this transformation. The expression is ready for the next step.

5.3.1 Symbolic Math Component

The symbolic math component has 2 different tasks. The main task is to generate a symbolic mathematical expression. This can be done by evaluating the modified expression to the global scope. This means, that Julia will look for an Array "D" and "c" and substitute their values to the expression. So, an Array "D" is declared to the global scope that contains "symbols" and the global Array "c" is assigned the coefficients of the stencil, given by the user.

The secondary task, which is optional and can only be used for simple stencils,is to apply temporal blocking. Temporal blocking as an optimization technique will be described at a following section. The main goal, is to create a new stencil, that when applied has the same effect as if the original stencil was applied multiple time-steps. In order to create that new stencil, the symbolic stencil expression is substituted to each one of the stencil components iteratively.

The SymEngine library was used for this task. SymEngine provides wrapper for the symengine symbolic manipulation library, written in C++.

5.3.2 Generating Kernel

This component, takes as input the symbolic stencil expression and determines the arithmetic operation necessary for the stencil computation. The modified version of NVIDA's kernel ,described in the previous chapter, was used as a template. The template can be split in independent parts:

- Initialization. Includes the shared memory allocation, the selection of the sub-array for the thread-block and the register initialization.

- Calculations. Includes the loop along the z axis, the arithmetic operations, reading the input array to shared memory and storing the result to global memory.

The symbolic math library was most useful for the arithmetic operations part. For each symbol that may be included in the expression (restricted by the maximum radius of the stencil) the offset and coefficient is queried. Whether a symbol is used by the stencil is inferred by the value of the coefficient, checking if it is different than zero. The register for the result and which cell of the shared memory is required are inferred by the offset.

After the kernel code is constructed as an expression object, it is passed to the SyntaxTree library, that creates a Julia function.

Kernel Execution System This system provides the execution of the generated kernel to a single or multiple GPUs. For the single GPU, the system is responsible for data padding, to transfer the data to the GPU memory and to call the kernel with the correct execution parameters.

For the multi-gpu approach, the system splits the data to the GPUs. The split data may overlap, depending on the amount of time steps each GPU does before communicating. After the determined number of time steps is executed, the GPUs have to exchange their overlapping regions and update them to GPU memory.

5.4 Optimizations

5.4.1 Global/Shared Memory

Since the template for kernel generation is based on the modified NVIDIA's 3D finite differences kernel by Mo et al., the same memory optimization are applied. Global memory efficiency is as high as possible and shared memory efficiency is 100%. Specifics were mentioned in the previous chapter.

5.4.2 Loop Unrolling

The calculation of the stencils, can be done in an iterative fashion over the radius, just like the example of the mathematical expression of the star stencil, given above. Also, there are stencils without a "periodic" pattern that cannot be expressed with an iteration. The framework fully unrolls any loops, resulting in the code in Listing 5.2.

Listing 5.2: Generated kernel. Unrolled calculations part.

```
@inbounds temp = tile[txr + -1, tyr + 0]
current += cfarr[1] * temp
@inbounds temp = tile[txr + 0, tyr + -1]
current += cfarr[1] * temp
@inbounds temp = tile[txr + 0, tyr + 0]
current += temp
infront1 += cfarr[1] * temp
```

```
behind1 += cfarr[1] * temp
@inbounds temp = tile[txr + 0, tyr + 1]
current += cfarr[1] * temp
@inbounds temp = tile[txr + 1, tyr + 0]
current += cfarr[1] * temp
```

This aggressive loop unrolling has both benefits and flaws. [15] Benefits include reduced instructions executed (removed branching and conditional checking), increased ILP (Instruction Level Parallelism) and efficient shared memory reading. In the example, the 3 registers require the same value. If loops were used, that one value from shared memory would be loaded 3 times. The flaws can be increased register pressure and decreased the occupancy which help hide memory access latency.

Unfortunately, not using any lp is the only way for partial contributions stored in registers.

5.4.3 Temporal Blocking

The main type for temporal blocking proposed by other papers, is to do multiple iterations over "cached" data, to calculate multiple time-steps. This type cannot be efficiently implemented for 3D stencils on GPU [2] [1]. The idea of temporal blocking for this framework, is to create a new stencil, that has the same multi-time step effect, but with only one iteration. A 2d star stencil is described by the following equation.

$$D^{t+1}[x,y] = c[0]D^t[x,y] + \sum_{i=1}^{n} c[i]D^t[x+i,y] + D^t[x-i,y] + D^t[x,y+i] + D^t[x,y-i]) \tag{5.1}$$

For the next time, t is replaced with t+1

$$\begin{aligned} &D^{t+2}[x,y] = c[0]D^{t+1}[x,y] \\ &+ \sum_{i=1}^{n} c[i]D^{t+1}[x+i,y] + D^{t+1}[x-i,y] + D^{t+1}[x,y+i] + D^{t+1}[x,y-i]) \end{aligned} \tag{5.2}$$

If 5.1 is replaced in 5.2, we get the value at D^{t+2}, using only the values at D^t. This procedure can be repeated m times to get the value at D^{t+m}. The new stencil will have $m * n$ maximum radius and the number of additional operations depends on the shape of the stencil.

The same applies for 3D stencil. With the non-holistic approach, the effects of the temporal blocking on shared memory usage are the same as a 2D stencil. However, register usage for partial results is increased. Additionally, redundant calculation are also performed. In this case, the redundancy is not in updating halo values, but in requiring more computations per element. A 2D star stencils requires $2 * 5 = 10$ calculations for 2 time steps whereas the time augmented 2D star stencils requires 11 computations. A 2d dense stencil goes from $2 * 9 = 18$ to 25 calculations.

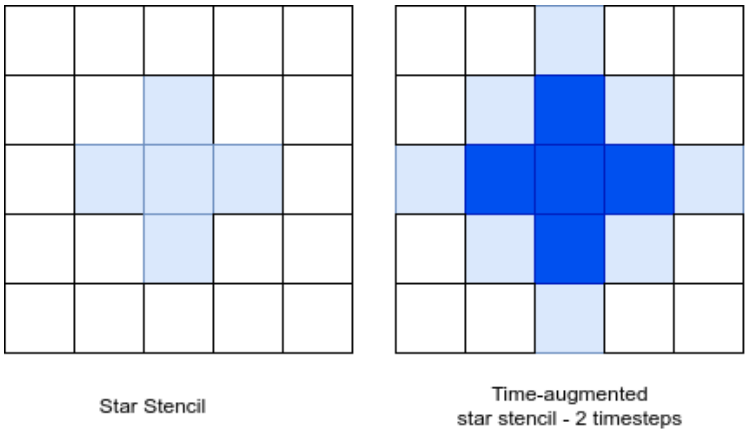

Figure 5.2: 2D time-augmented star stencil.

5.5 Multi GPU

The multi gpu implementation can be reduced to 3 problems:

- How to split the data
- How to communicate the updated halo
- How many time steps each GPU does per communication

The approach to the first problem, was to split the data along the z-axis. The advantages of this approach is its simplicity and the fact that the data are contiguous after the split. Also, if the system has different GPUs, the ratio of the split data is analogous to their speed.

This approach also helps with the second problem of transferring halos. The halos are contiguous in space if the split is done along the z axis, which helps with the communicates irregardless of the parallelization method (threads, MPI, processes, peer to peer). In particular, the halo communication was implemented as peer to peer memory copy.

The number of time steps per communication is highly dependent on the system memory and method of communication. Large number of time steps results in less communications, and thus low communication overhead, but more GPU memory usage, larger data transfers and more redundant calculations.

5.6 Frequency Domain stencil

Increasing the size of the stencil, has a large impact on performance. Large stencils require more calculations, more registers and more shared memory. Shared memory and registers are the most important reasons for low occupancy. After a certain limit, the requested registers and shared memory exceed the available resources of the GPUs and computation of such stencils is unfeasible. So, frequency domain stencil, using FFT for convolution is both a solution for too large stencils and a possible optimization for aggressive temporal blocking.

FLCC shows for large templates, it is preferable to do convolution in the frequency domain.

Chapter 6

Results and Conclusion

In this chapter, Julia is evaluated in terms of performance and code quality.

6.1 Performance Results

First and foremost, the performance of the framework has to be compared against an already known high performance implementation. For this purpose, NVIDIA's kernel, that was provided in the paper, was selected in addition to the modified version that was reimplemented based on the paper by Mo et al. Both implementations are written in C.

Since, NVIDIA's 3D finite differences kernel can only calculate the star stencils, the three implementations were benchmarked only for that type. The benchmarks were run on a Tesla K80 and a Tesla T4 gpus for different block size and stencil radius.

IMAGE

For block size of 32x32 threads, the implementation in Julia is always faster than the NVIDIA's kernel and sometimes faster than the equivalent kernel in C. The benchmarks show that the template kernel that was chosen is well qualified, Julia as a language performs as well as C and the kernel generator works correctly.

With the good performance of the framework's foundation in mind, the research can continue on the optimizations and multi-gpu performance.

6.1.1 Temporal Blocking

Important parameters that affect performance are the density of the stencil, the initial radius and the number of merged time-steps. Three types of stencil were chosen with varied density, flux summation (CITE PDF), star and dense from least to most dense.

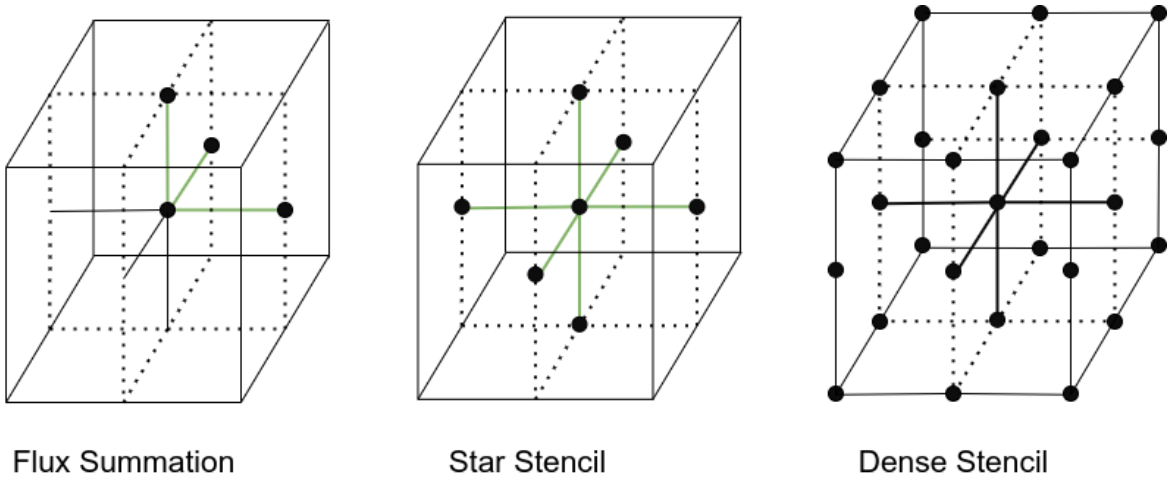

Figure 6.1: 3 types of stencils. 1) Flux Summation 2) Star 3) Dense

Benchmarks were run on Tesla T4 and Tesla K80, for 256x256x256 data size. The plots in Figure 6.2, show the speed up of time augmented stencil over one time step relative to the original stencil responsible for one time-step. The original stencil has radius 1.

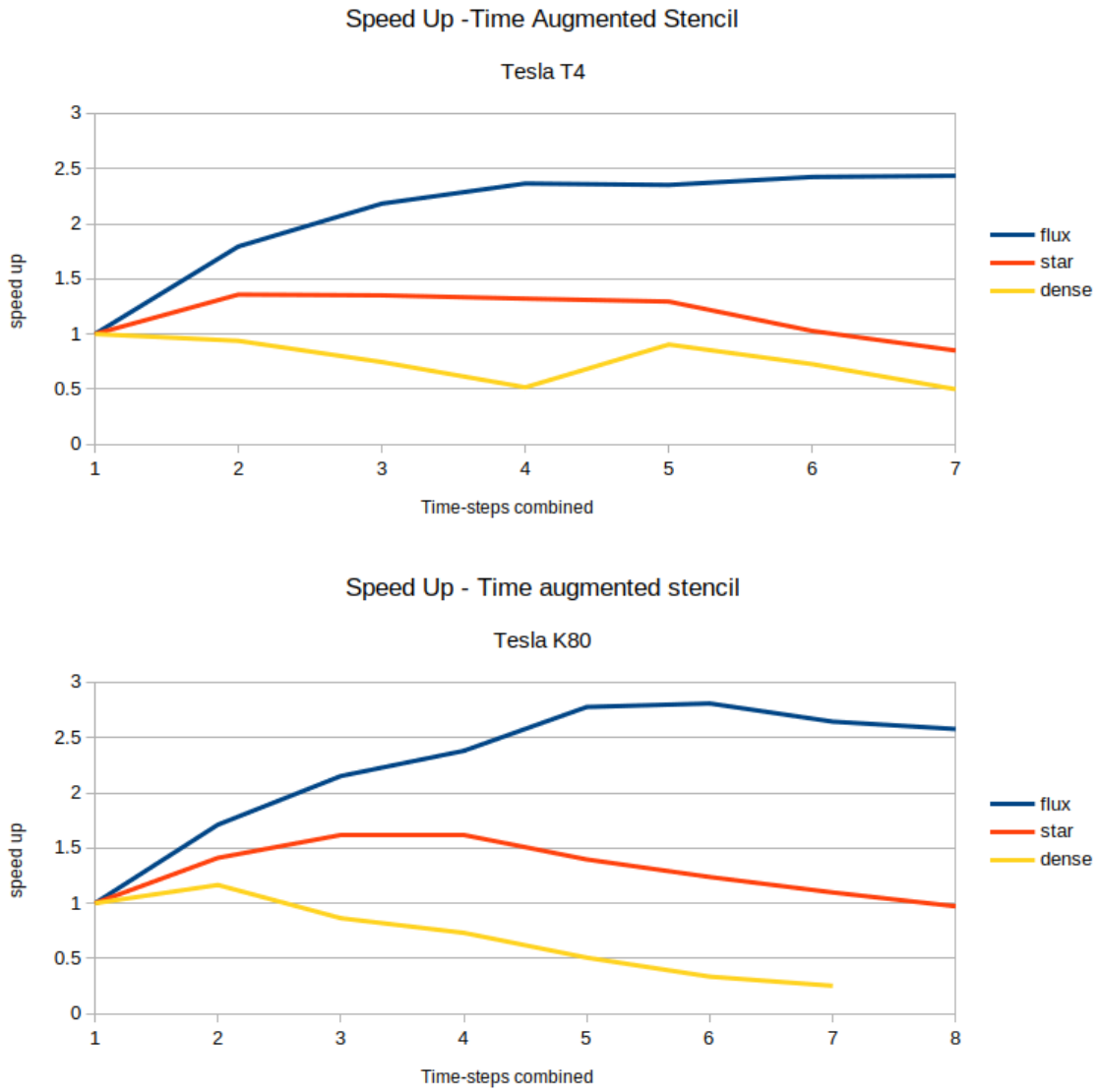

Figure 6.2: Time augmented stencil. Speed up over one time-step

Dense stencil doesn't seem to benefit from temporal blocking. Time augmented stencil is always slower than the original, except for one case, for two combined time-steps in K80, being 1.16x faster than the original.

For the star stencil, the largest speed up is

Applying temporal blocking to star stencil with radius 2,has 1.15x speed up ,only for 2 combined time-steps, and none for more time-steps. If the radius is larger than 2 there is no benefit at all.

In order to compare the time augmented stencil optimization performance against an existing implementations, results from Meng et al. [1] and Holewinski et al. [2] will be used. Cell stencil is used in Conway's Game of Life, a cellular automation and reduces the values of a neighborhood to a single transition. The shape of the neighborhood is the same as the star stencil. Jacobi 3D is identical to the star stencil.

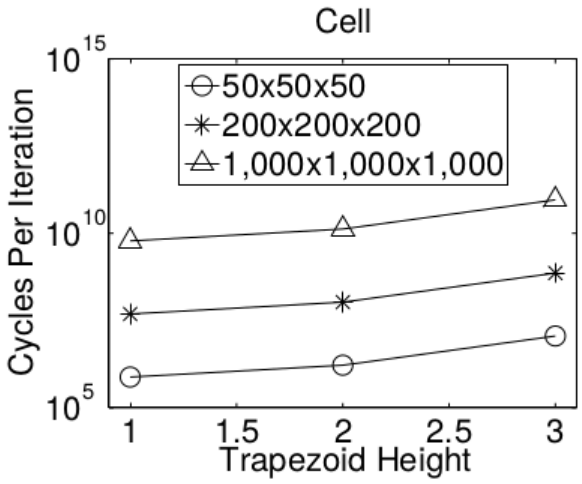

Figure 6.3: Meng et al. [1] Temporal blocking performance for various input sizes. Trapezoid height implies time-steps combined.

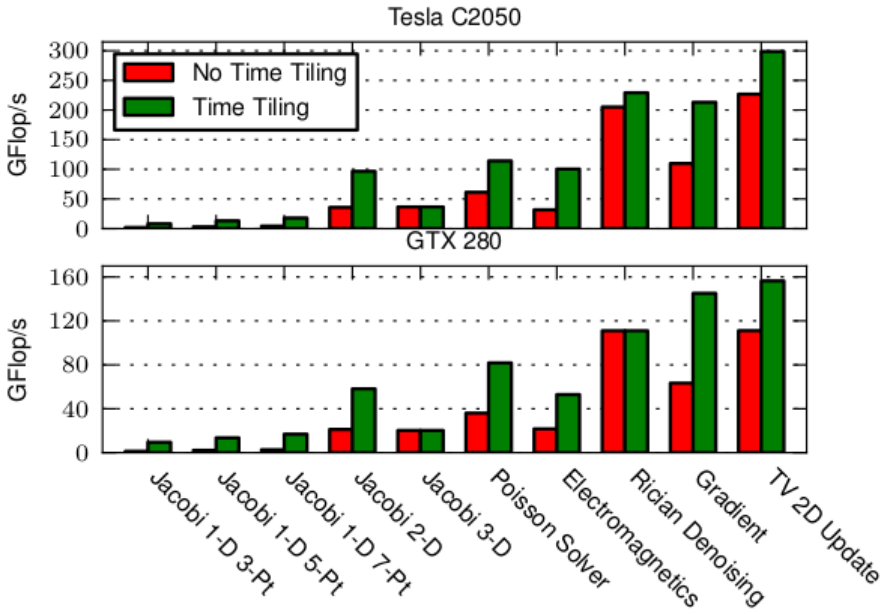

Figure 6.4: Holewinski et al. [2] Performance on multiple GPUs, with and without temporal blocking. Only Jacobi 3D is three dimensional stencil. The rest or 2D or 1D

It's apparent that none of them achieves any speed up for 3D stencils with temporal blocking. Moreover, the CELL stencil is slowing down. his comparison and the significant x1.5 speed up (for the star stencil) are encouraging for further investigation and additional optimizations for large stencils.

Although the framework was built with 3D stencils in mind, the same temporal blocking technique can be applied to 2D stencils. Only benchmarks for 2D star stencil with radius 1 and 2 will be executed, so they can be compared directly to the identical 5-point and 9-point Jacobi results from Holewinski et al.

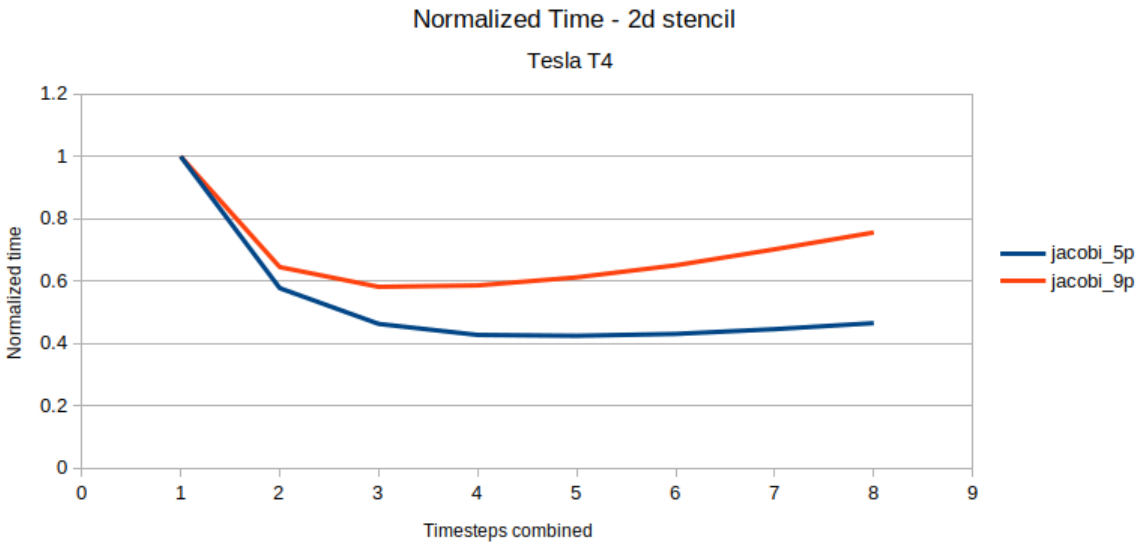

Figure 6.5: Time-augmented stencil normalized time for 2D Jacobi(star) kernel.

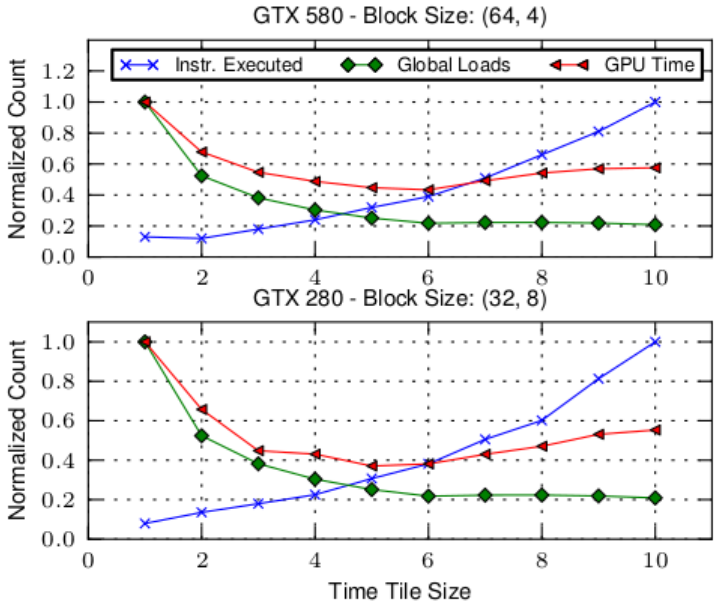

Figure 6.6: [2]Temporal blocking normalized time for 2D Jacobi kernel.

Holewinski achieves achieves the least time for 6 time-steps, just below 0.4 fo the one-time-step time. Time augmented stencil, achieves just over 0.4 of the one-time-step time for the 5th time steps. The difference is minimal and shows this technique works well even for 2D stencils.

6.1.2 Multi GPU

The benchmarks were run on 4 Tesla K80 GPUs, for different radius and data sizes. Additionally, it was measured whether it is better to do multiple time-steps in each

GPU before communicating the halo. This was suggested for research by Micikevicius [6] for small stencils.

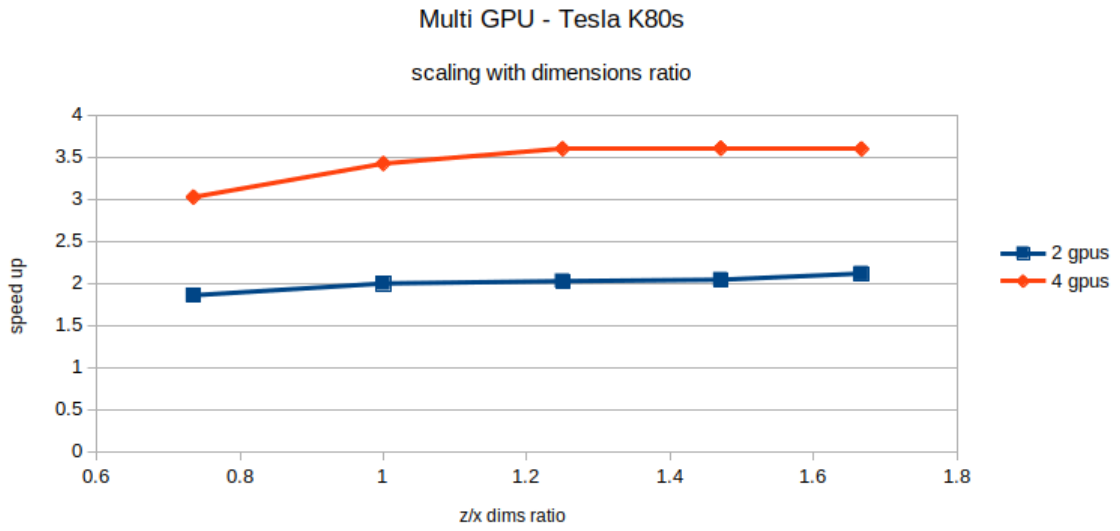

Figure 6.7: Speed up scaling for different z/x dimensions ratio. Star stencil, radius 8

It is obvious that the ratio between the size of x (or y) dimension with z impacts performance. Since the data between GPUs is split on the z axis, the size of transferring data region is proportional to the x and y dimensions size.

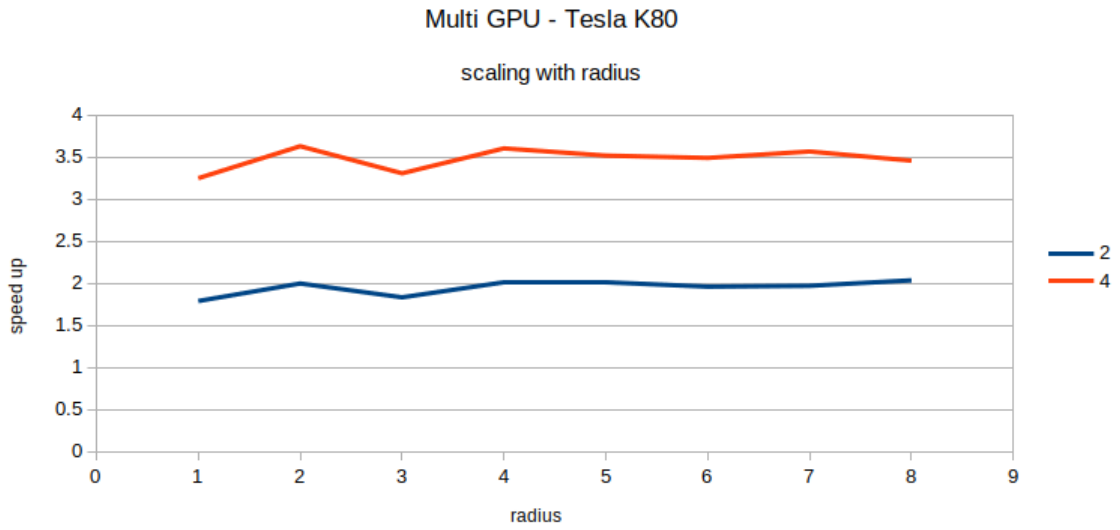

Figure 6.8: Speed up scaling for different z/x dimensions ratio. Star stencil, radius 8

Figure 6.8, shows that the size of the radius, doesn't affect how well performance scales with multiple GPUs. Using 2 GPUs, results in a consistent 2x speed up and for 4 GPUs 3.5x speed up.

Doing multiple time-steps in one GPU before communicating didn't yield any speed up. The best that was measured was x1.06 speed up. For more combined time-steps the performance was worse. For many combined time-steps and/or big radius, doing multiple time steps causes 2x slow down.

While doing multiple time-steps in one GPU reduces communication overhead, it does not decrease the communication cost, since the same amount of data is transferred. Moreover, the computational cost increased because of the additional ghost region the kernels had to perform on. Probably, the computational cost is larger than the communications overhead.

Since the server didn't allow the use of a profiler, the results can be noisy. Additionally it was not feasible to measure computation and memory transfer time separately. Although absolute comparisons may not be precise, the trend of measurements for different configurations is correct and apparent.

6.1.3 FFT for stencil

6.2 Code Quality

Chapter 3 includes the review in code quality when writing a CUDA kernel in Julia. But writing kernels is only a small part of an application. Usually, applications include GPU configurations, data pre-processing, data visualization, result verification, etc. In this chapter, the scope of code quality is expanded, covering an entire framework.

6.2.1 Framework

Introspection -> already parsed Easy to manipulate, Recursive

no trouble with command line to compile. Code generation. Expression first class citizen.

Creating a code block is as easy as appending expression to array

One function to transform an expression to function.

View macro and 3d indexing remove complexity making it easier to create code

Embedding constants

6.2.2 Single GPU

CUDA.jl is really optimized for single GPU context Uploading CuArray(code) download Array() (code)

6.2.3 FFT

showcases Julia prototyping efficiency

6.2.4 Multi GPU

Does not include higher level abstraction for multiple gpus. CuArray is not compatible with Unified memory, powerful feature for simplifying multi gpu control code

CUDA functions, are wrapped but not documented (no comments or even type information)

not mature, limited to non existant resources online.

trial and error

problem with Garbage Collection

doesnot support thread per gpu, only serial inefficient or CUDAaware MPI

6.3 Discussion

Documentation not mature for multi stream multi gpu. Slow runtime kernel compilation at first call.

6.4 Conclusion

Appendix A

Acronyms and Abbreviations

LAN Local Area Network

Bibliography

[1] J. Meng and K. Skadron, "A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 115–142, Feb. 2011. [Online]. Available: http://link.springer.com/10.1007/s10766-010-0142-5

[2] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. Association for Computing Machinery, pp. 311–320. [Online]. Available: https://doi.org/10.1145/2304576.2304619

[3] J. Nickolls and W. J. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar. 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5446251/

[4] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin, TX: IEEE, Nov. 2008, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/document/5222004/

[5] "CUDA C Best Practices Guide," p. 85.

[6] P. Micikevicius, "3d finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*. ACM Press, pp. 79–84. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1513895.1513905

[7] T. Mo and L. Renfa, "A new memory mapping mechanism for gpgpus' stencil computation," *Computing*, vol. 97, pp. 795–812, 11 2014.

[8] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," *Proceedings of the 19th annual international conference on Supercomputing - ICS '05*. [Online]. Available: https://www.academia.edu/19784992/Cache_oblivious_stencil_computations

[9] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-d blocking optimization for stencil computations on modern cpus and gpus," 11 2010, pp. 1–13.

[10] R. V. Lim, B. Norris, and A. D. Malony, "Autotuning GPU Kernels via Static and Predictive Analysis," *arXiv:1701.08547 [cs]*, Jun. 2017, arXiv: 1701.08547. [Online]. Available: http://arxiv.org/abs/1701.08547

[11] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," 04 2010, pp. 1–12.

[12] J. D. Garvey and T. S. Abdelrahman, "A Strategy for Automatic Performance Tuning of Stencil Computations on GPUs," *Scientific Programming*, vol. 2018, pp. 1–24, May 2018. [Online]. Available: https://www.hindawi.com/journals/sp/2018/6093054/

[13] A. Schäfer and D. Fey, "LibGeoDecomp: A Grid-Enabled Library for Geometric Decomposition Codes," Jan. 1970, pp. 285–294.

[14] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," 05 2011, pp. 676–687.

[15] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for GPGPU programs," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. Atlanta, GA, USA: IEEE, 2010, pp. 1–11. [Online]. Available: http://ieeexplore.ieee.org/document/5470423/