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

Τίτλος διπλωματικής

Διπλωματική Εργασία
του
Κωνσταντίνος Χατζηαντωνίου

Επιβλέπων: Νικόλάος Πιτσιάνης
Καθηγητής Α.Π.Θ.

9 Σεπτεμβρίου 2020

Abstract

General Purpose

Abstract

Empty

Ευχαριστίες

Άδειο

Τίτλος διπλωματικής

Όνομα Επίθετο
empty@auth.gr

9 Σεπτεμβρίου 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++ [1]

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

```c
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

```julia
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.

```julia
# 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 $\frac{\text{Julia version's time}}{\text{C time}}$ for a given problem size.

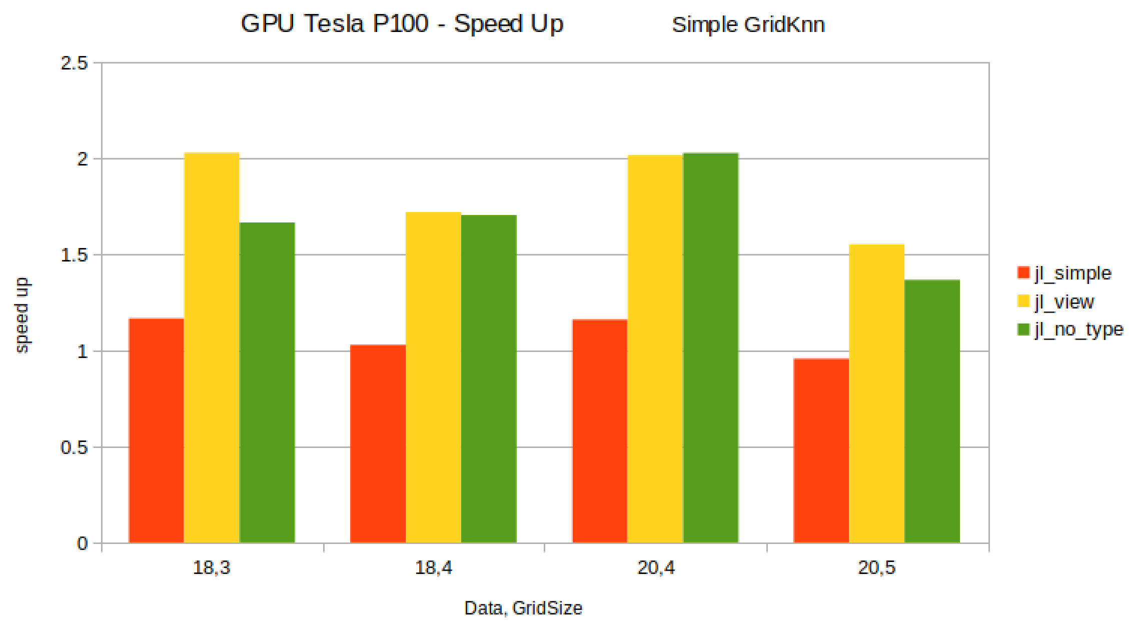

Figure 3.1: Comparison of Julia over C for simple GridKnn

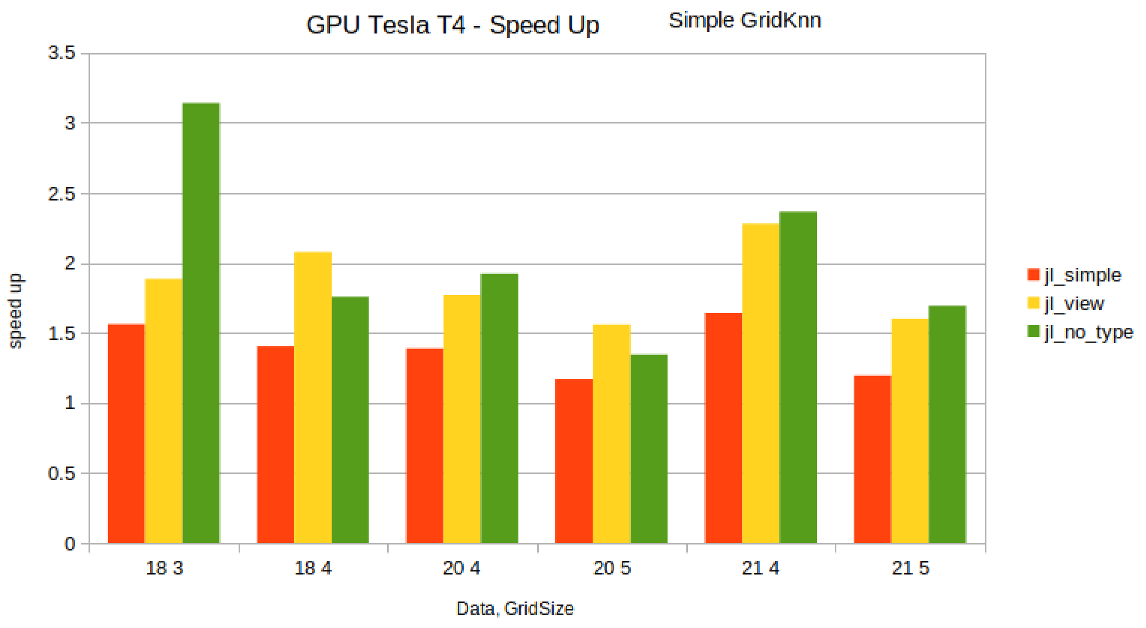

Figure 3.2: Comparison of Julia over C for simple GridKnn

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.

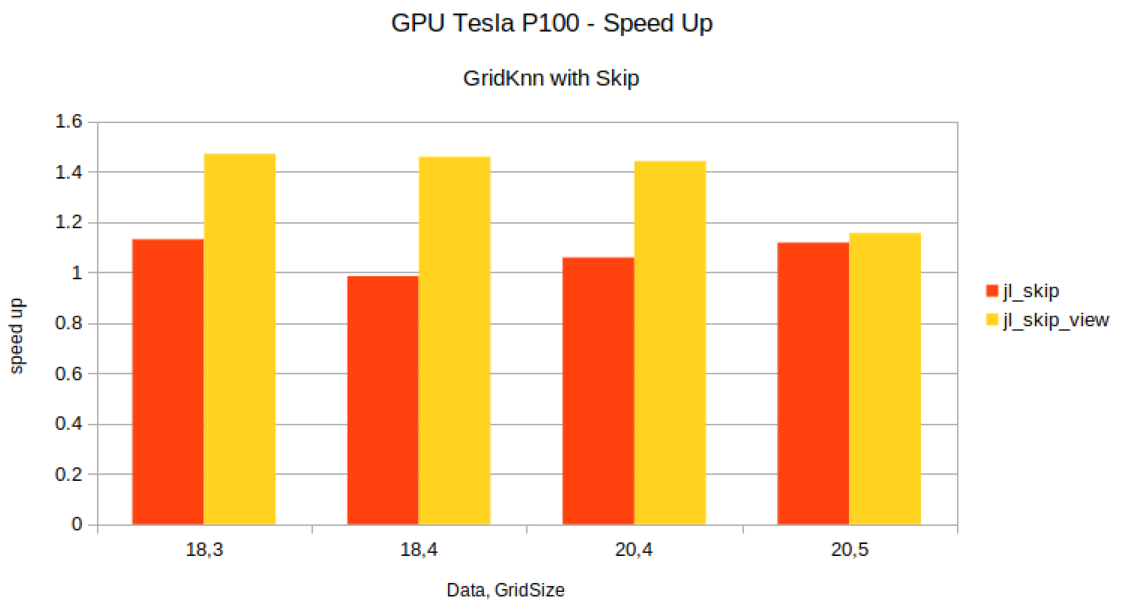

Figure 3.3: Comparison of Julia over C for GridKnn with Skip

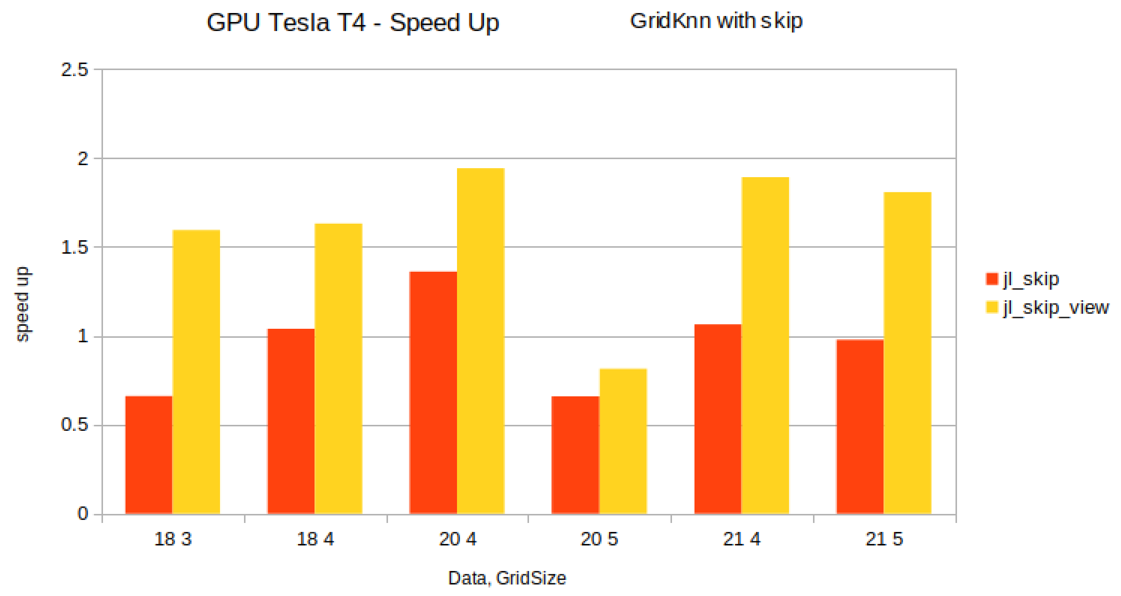

Figure 3.4: Comparison of Julia over C for GridKnn with skip

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 *Share 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.

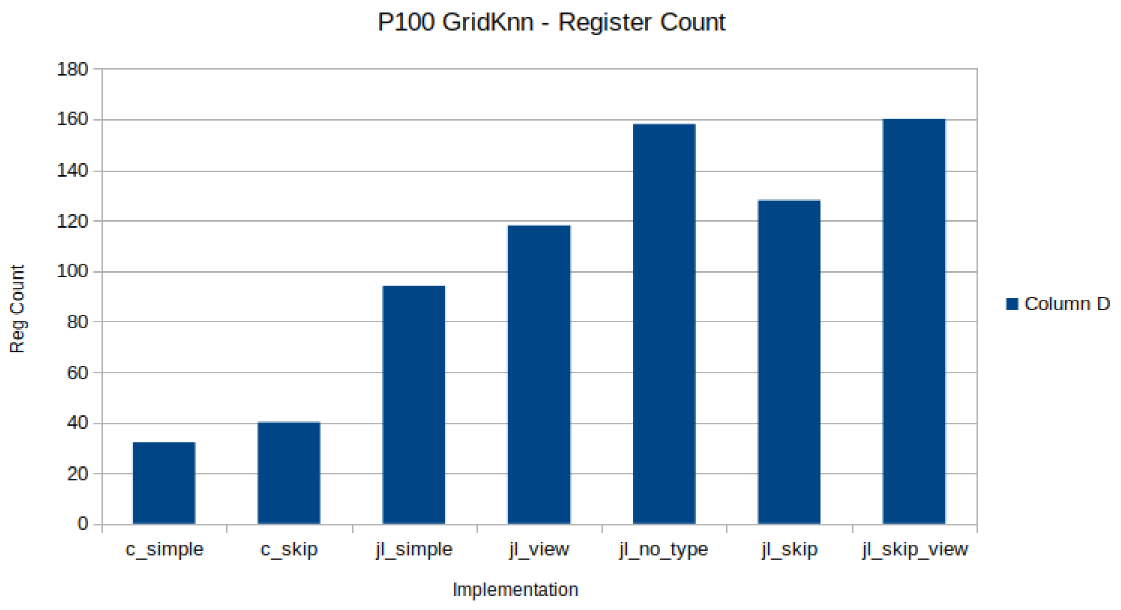

Figure 3.5: Register Usage for GridKnn implementations.

[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.

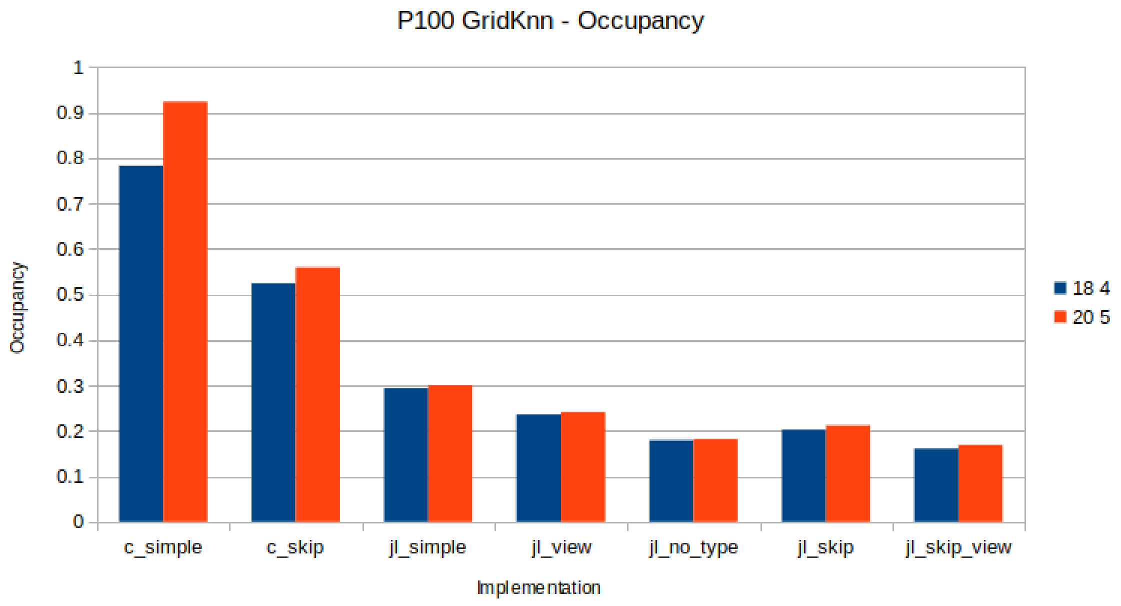

Figure 3.6: Register Usage for GridKnn implementations.

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.

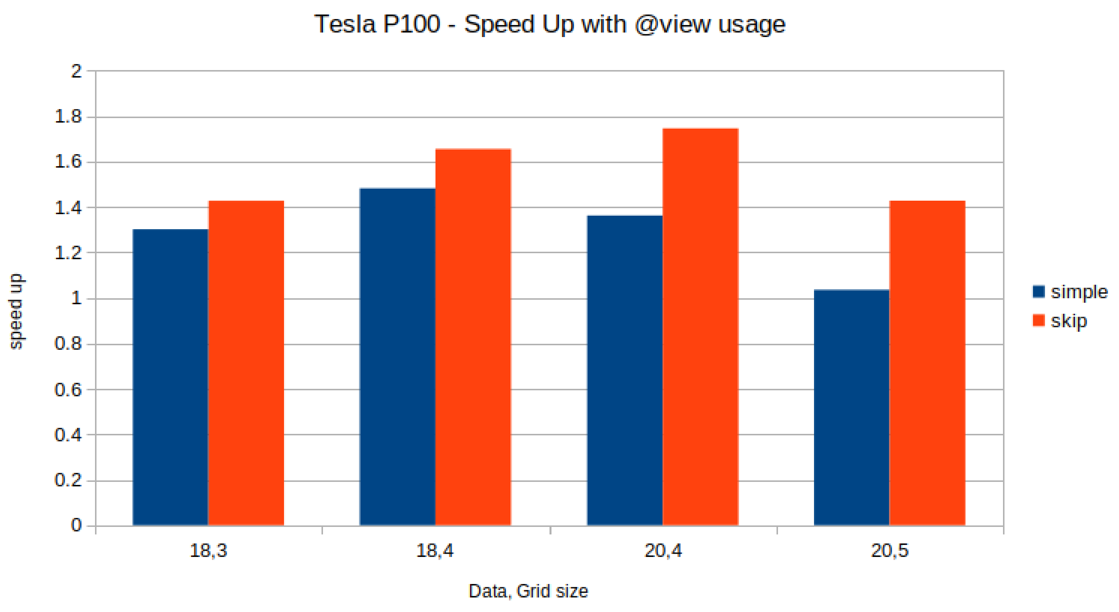

Figure 3.7: Julia - Speed Up with the use of @view.

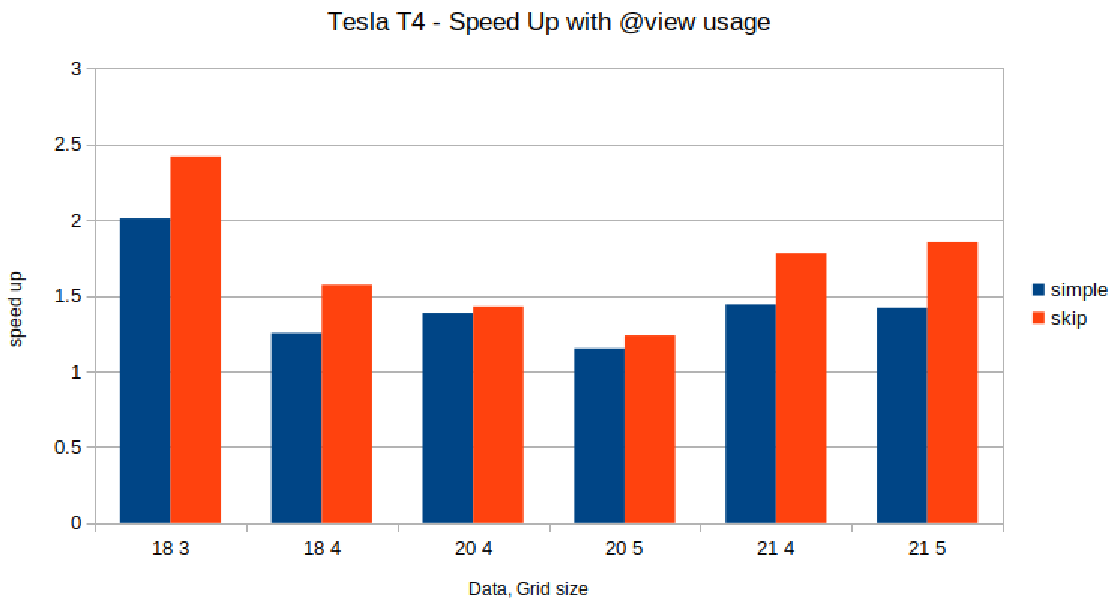

Figure 3.8: Julia - Speed Up with the use of @view.

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.

```c
// ------- 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.

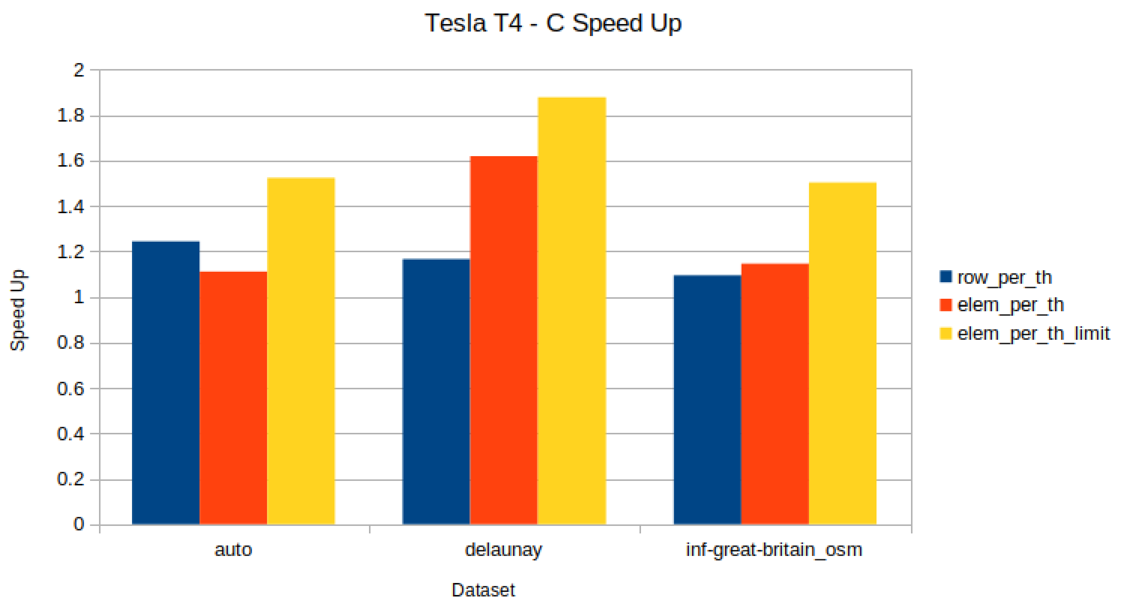

Figure 3.9: Julia - Speed Up with the use of @view.

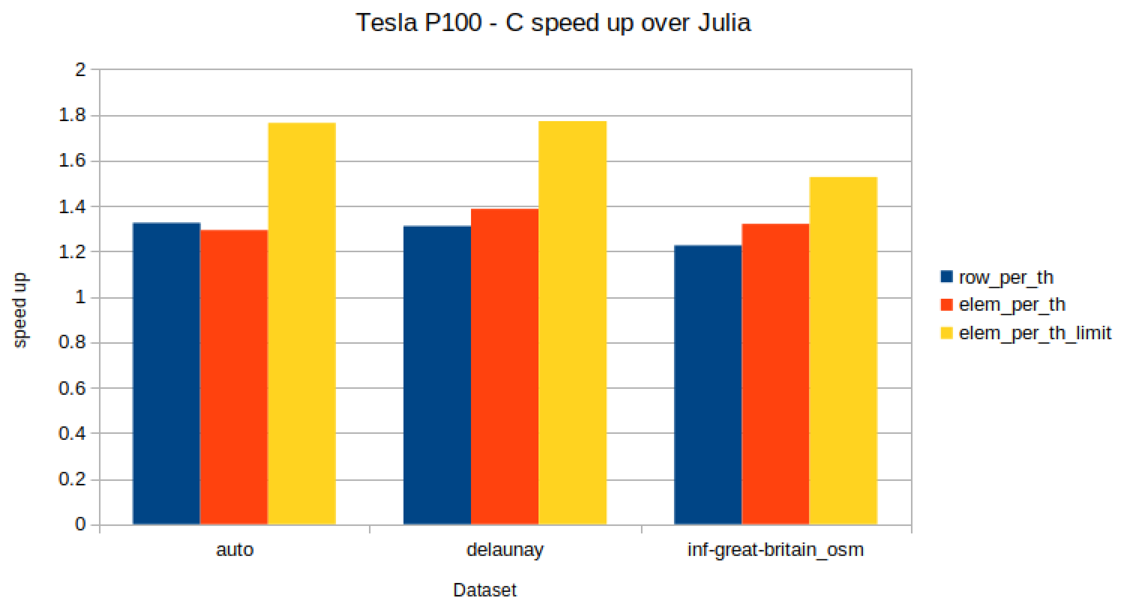

Figure 3.10: Julia - Speed Up with the use of @view.

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.

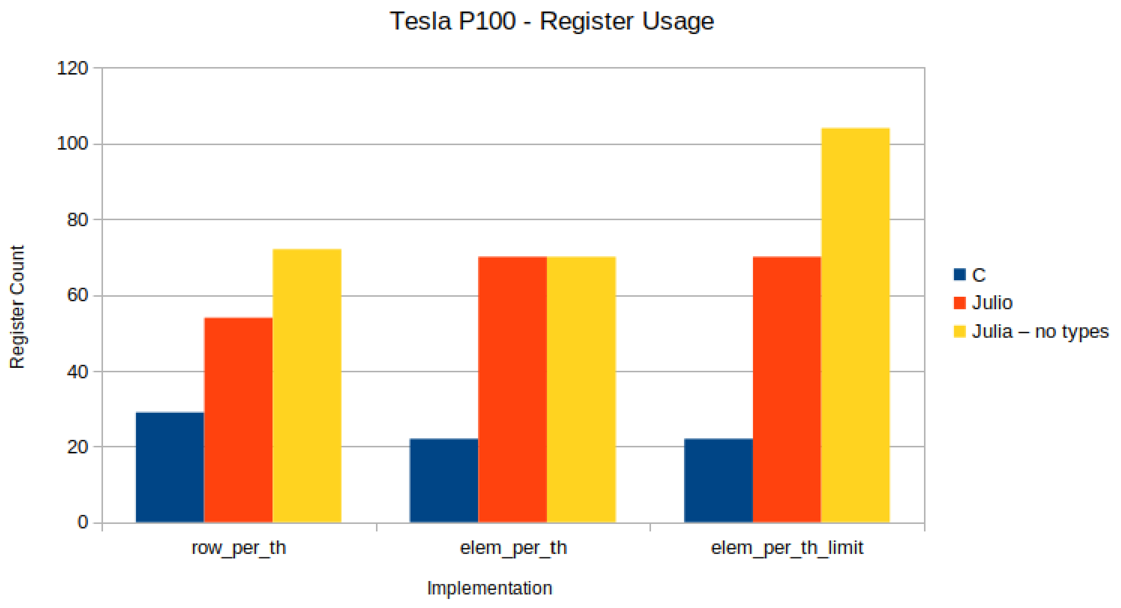

Figure 3.11: Julia - Speed Up with the use of @view.

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

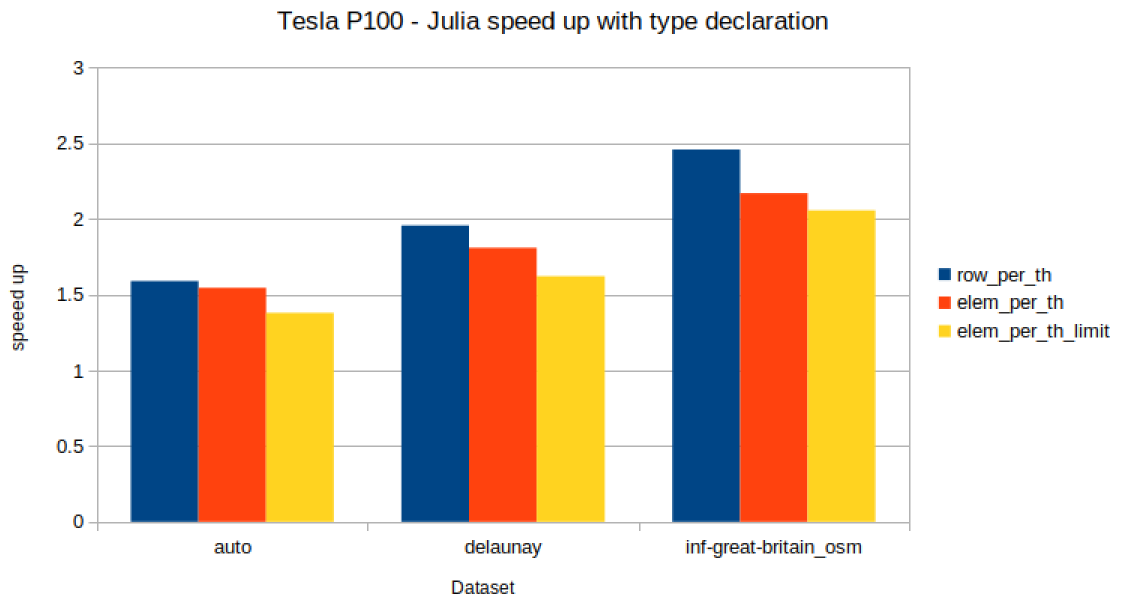

Figure 3.12: Julia - Speed Up with the use of @view.

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...

==

[WIKIPEDIA COPY] Stencil codes are a class of iterative kernels[1] which update array elements according to some fixed pattern, called a stencil.[2] They are most commonly found in the codes of computer simulations, e.g. for computational fluid dynamics in the context of scientific and engineering applications. Other notable examples include solving partial differential equations,[1] the Jacobi kernel, the Gauss–Seidel method,[2] image processing[1] and cellular automata.[3] The regular structure of the arrays sets stencil codes apart from other modeling methods such as the Finite element method. Most finite difference codes which operate on regular grids can be formulated as stencil codes.

4.2 Previous Research

4.3 Stencil on GPUs

Stencil codes are easily

4.4 Optimization

4.5 Time Blocking

Time blocking is an optimization technique that aims enhance data-locality by reusing data ,that are already loaded in local memory, to compute multiple time-steps. [] Describes a cache oblivious, time-blocking, recursive algorithm for stencils computed on CPUs. [] [] Present a time blocking implementations for GPUs.

4.5.1 Hand-written stencil kernels

Micikevicius [2] presents an 3d stencil implementation for 3d finite difference computation. He mainly focuses on reducing redundant memory reads, and compares the one-pass and two-pass approach. He concludes that the one pass approach does half the redundant data accesses than the two-pass approaches (this value varies for different stencil sizes). Micikevicius follows a holistic approach, meaning that all the data needed for a stencil computation are loaded from global memory to either registers or shared memory. This method however doesn't manage to fully utilize the global read memory bandwidth.

Mo et al. [3] solve the coalesced reads problem by introducing a more atomistic approach. Instead of calculating the whole stencil at once, they calculate the partial weighted sum from the points available only at a given XY tile. The final result is obtained by accumulating the partial sums to the registers.

With this approach, the data, both the main 2d rectangular and the ghost zones, are loaded to shared memory in a coalesced manner, whereas Micikevicius loads the main rectangular and the ghost zones in different steps. Their benchmarks show an 1.83x speedup on a Tesla C2050.

This paper [4] suggests another optimization to reduce global memory reads over multiple time steps. They propose overlapped tiling, a kind of temporal blocking. Each thread block, does multiple time steps in every kernel call and each time the result is saved to shared memory instead of the global memory. Although global memory reads are reduced for every time step done internally, this method increases the amount of shared memory used and some threads do redundant calculations on the ghost regions. Their benchmarks for their implementation show a 6 time steps tile, is 3x faster than an 1 time step stencil.

4.5.2 Auto-Tuning

Cache oblivious a cache oblivious algorithm for stencil computations, which arise for example in finite-difference methods. Minimizes cache misses compared to a naive algorithm, and it exploits temporal locality optimally throughout the entire memory hierarchy

BUT

Parallel Data-Locality Aware Stencil Computations For the GPU version of the stencil code, the obvious implementation of the kernel has been chosen. Temporal blocking as described in the previous section is not feasible on the GPU due to its very limited shared memory (16 KB per multiprocessor, which is still distributed among the thread blocks that run on a multiprocessor – however, to optimally hide data transfer latencies, as many blocks as possible should run on one multiprocessor).

Paper Fourier for stencils

4.6 Automatic Stencil Generation

LibGeoDecomp LGD is a library for time discrete simulations in structured grids that utilizes heterogenous paralellization. Although it is not limited to stencil only computations, the library provides an interesting interface for the user to define the problem, and hides all aspects of parallelization from them. The user defines a class that describes how each cells interacts with the neighbours and provides it to the library initializer.

While this is easier than writing your own kernel, or modifying the application, the user must study the boilerplate code for initialization and how to describe cell interactions. Also, C++ doesn't leave much room for expressiveness in the description.

In order to create a stencil kernel from its mathematical description, an implementation is required that is both fast and easily expandable. The trick that [Guy at jacobi] uses to reduce computations is difficult to implement for any type of stencil and requires checks for boundary conditions. The kernel shown at [Guy at NV finite] is fast and simple at the same time.

4.7 Efficient Generic Stencil Kernel

4.7.1 NVIDIA Finite Difference Kernel

Implementation

Pros Why fast..

Cons Can only be used for star stencil. Although it is possible to have more complicated pattern in xy plane, it only allows a simple line in the z axis. Also, the global read efficient is not optimal for many combinations of stencil radius and kernel block size.

4.7.2 NVIDIA kernel modification

Implementation Use regs for results.

Pros Better Global Read Efficiency.

4.7.3 Performance Comparison

The 2 implementation are compared for different block size, stencil radius and data sizes.

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

As stated before, there are many different applications that require stencil codes. Writing the whole kernel from zero takes time. Modifying an existing kernel for the needs of each application is not efficient. First of all, deep understanding of the code is required for correct and efficient implementation. Understanding a code written from someone else requires time too.

Therefore an...

5.2 Framework Infrastructure

The framework, consists of 4 basic components. The *Parser*, that takes as input a Julia expression object, modifies it by evaluating internal macros and functions and restore indexing to 1-based. The second component, transforms the modified expression to symbolic, which optionally can be further processed for temporal blocking. Then, code generator takes the symbolic math, and converts it to arithmetic operations. 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.

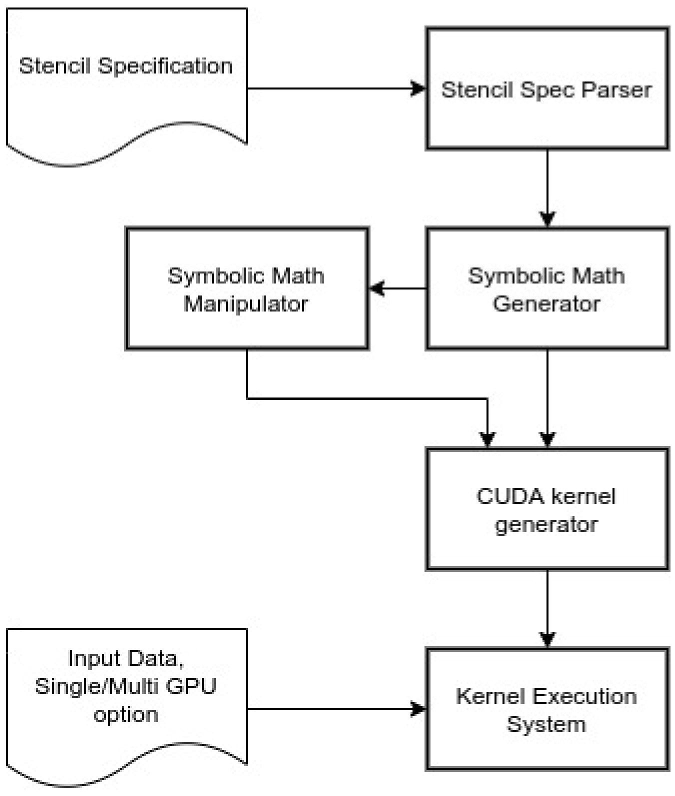

Figure 5.1: High level overview of the architecture.

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.

5.2.1 Symbolic Math Component

The symbolic math component has 2 different jobs. The main job 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 next job, 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++.

5.2.2 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, 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.

5.3 Optimizations

5.3.1 Global Memory

Reading and writing from global memory is slow. The fastest way to access global memory elements is coalescing, meaning each thread of the wrap must access consecutive elements. The optimization applied is explained in the previous chapter for the modified version of NVIDIA's finite differences kernel.

5.3.2 Shared Memory

5.3.3 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. Using unrolling, the kernel reuses data from shared memory, reducing the amount of loading operations. Also, stencils without a pattern, cannot be expressed with an iteration. So unrolling, increases the number of stencils that can be computed.

In listing 5.4, is presented the calculations code of a generated kernel for a star stencil with radius 1. The center cell, used for 3 calculations, is loaded to a temporary variable to reduce the pressure on shared memory.

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
```

5.3.4 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 easily implemented on the GPU. The idea of temporal blocking for this framework, is to create a new one, that can have a 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

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

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.

5.4 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 the simplicity, the fact that the data are contiguous after the split. Also, if the system has different GPUs, the size 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).

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 and larger data transfers.

TODO experiments

Appendix A

Acronyms and Abbreviations

LAN	Local Area Network

Bibliography

[1] 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/

[2] 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

[3] T. Mo and R. Li, “A new memory mapping mechanism for GPGPUs’ stencil computation,” vol. 97, no. 8, pp. 795–812. [Online]. Available: http://link.springer.com/10.1007/s00607-014-0434-5

[4] 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