Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Βελτιστοποίηση πολλαπλασιασμού αραιού πίνακα με διάνυσμα σε επεξεργαστές γραφικών με τη χρήση Συνελικτικών Νευρωνικών Δικτύων.

Διπλωματική Εργασία

Πέτρος Αναστασιάδης

Αθηνα,

Ιουλιος 2018

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Βελτιστοποίηση πολλαπλασιασμού αραιού πίνακα με διάνυσμα σε επεξεργαστές γραφικών με τη χρήση Συνελικτικών Νευρωνικών Δικτύων.

Διπλωματική Εργασία

Πέτρος Αναστασιάδης

Επιβλέπων καθηγητής: Γεώργιος Γκούμας

Εγκρίθηκε από την τριμελή επιτροπή στις 5 Ιουλίου 2018.

Γεώργιος Γκούμας	Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου
Επίκ. Καθηγητής, Ε.Μ.Π.	Καθηγητής, Ε.Μ.Π.	Αναπ. Καθηγητής, Ε.Μ.Π.

Αθηνα,

Ιουλιος 2018

Πέτρος Αναστασιάδης
Διπλωματούχος Εθνικού Μετσοβίου Πολυτεχνείου

Περίληψη

Σκοπός της παρούσης διπλωματικής εργασίας είναι η υλοποίηση μιας μεθόδου πρόβλεψης για τον πολλαπλασιασμό αραιού πίνακα με διάνυσμα σε επεξεργαστές γραφικών. Η προϋπάρχουσα έρευνα έχει δείξει πως η πρόβλεψη βέλτιστης μεθόδου για τον πολλαπλασιασμό αραιού πίνακα με διάνυσμα δεν είναι εύκολη, και η ακρίβεια πρόβλεψης που μπορούν να προσφέρουν τα υπάρχοντα μέσα δεν είναι αρκετά ικανοποιητική. Μια και η πλειοψηφία των επαναληπτικών μεθόδων και άλλων τεχνικών της γραμμικής άλγεβρας εμπεριέχουν πάρα πολλούς συνεχόμενους πολλαπλασιασμούς αραιού πίνακα με διάνυσμα, ακόμα και κάποια χρονοβόρα μέθοδος πρόβλεψης θα μπορούσε να οδηγήσει σε σημαντική αύξηση απόδοσης. Με αυτο ως βάση, σε αυτή τη διπλωματική εργασία θα εξερευνήσουμε και αξιολογήσουμε τη χρήση Συνελικτικών Νευρωνικών Δικτύων για την πιο ακριβή πρόβλεψη της βέλτιστης μεθόδου πολλαπλασιασμού αραιού πίνακα με διάνυσμα σε επεξεργαστές γραφικών.

Λέξεις κλειδιά: Πολλαπλασιασμός αραιού πίνακα με διάνυσμα; Επεξεργαστές γραφικών; Συνελικτικά Νευρωνικά Δίκτυα ; Πρόβλεψη βέλτιστης μεθόδου; Κατηγοριοποίηση εικόνων

Abstract

This thesis focuses on implementing a method of format prediction for Sparse Matrix-Vector multiplication in GPUs. Previous research has shown that SpMV format prediction is not an easy task, and the prediction rates it can achieve with normal means are not satisfactory enough. We believe that with the majority of iterative solvers and other linear algebra operations requiring a lot of consecutive SpMV kernel launches, even time consuming methods of prediction could lead to a noticeable increase in performance. To this end, in this thesis we will explore, train and evaluate Convolutional Neural Networks in order to predict the best SpMV format for GPUs with satisfactory accuracy.

Keywords: Sparse matrix-vector multiplication; SpMV; Graphic Processing unit; GPU; Convolutional neural network; CNN; Format prediction; Image Classification

Acknowledgments

This thesis was assigned by the computer system laboratory (cslab) of the school of electrical and computer engineering of the National Technical University of Athens. The supervisor for this thesis was Georgios Goumas. I would like to thank the whole cslab for providing the resources and basis for this thesis (especially the cslab admin for his patience regarding installation of obscure python dependencies).

More specifically, i would like to thank the PHD-candidate Athena Elafrou for her assistance, advice and guidance for the whole duration of this thesis, the Professor Georgios Goumas for his support and my family and friends for helping me all this time. Without them, i wouldn't be able to arrive at this point.

Contents

Chapter 1

Background

1.1 Graphics Processing Units (GPUs)

The history of GPUs

Graphics Processing Units have been around for quite a long time. The first GPU-like systems were introduced around the mid-70s in the form of "video shifters" and "video chips". After that, more systems specialised in 2D and 3D rendering made their appearance, resulting in the "first GPU" introduced by Nvidia in 1999, GeForce 256. Nvidia defined the term graphics processing unit as "a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second" [Singer, 2013].

The era of the general purpose GPUs began in 2007. Both Nvidia and ATI (since acquired by AMD) started packing their graphics cards with ever-more capabilities. Until then, a GPU was nothing like the modern programmable GPUs. It could only process certain graphics operations with greater speed than CPUs, and was designed and programmed for very specific tasks. That was about to change with the introduction of general purpose computing GPUs (GPGPUs), and the release of programming frameworks for GPUs. However, the two companies took different tracks to general purpose computing GPU (GPGPU).

In 2007, Nvidia released its CUDA development environment, the earliest widely adopted programming model for GPU computing. CUDA was focused in harnessing the full power of Nvidia GPUs for a number of different applications, and was designed to give the programmer some control and knowledge of the work flow in the GPU, without uncovering the whole architecture of the GPU in use.

Two years later, OpenCL was introduced and became widely supported, allowing the development of code for both GPUs and CPUs with an emphasis on portability. In the next years OpenCL extended support to many other accelerators. Thus, GPUs became a more generalized computing device.

Today, graphics processing units are not only for graphics. They have found their way into fields as diverse as machine learning, big data research and AI, oil exploration, scientific image processing, statistics, linear algebra, 3D reconstruction, medical research and even stock options pricing determination. The GPU technology tends to add even more programmability and parallelism to a core architecture that is ever-evolving towards a general purpose CPU-like core.

GPU vs CPU performance

A central processing unit (CPU) is designed to handle complex tasks, such as time slicing, virtual machine emulation, complex control flows and branching, security, etc. In contrast, graphics processing unites (GPUs) only do one thing well. They handle billions of repetitive low level tasks. Originally designed for the rendering of triangles in 3D graphics, they have thousands of arithmetic logic units (ALUs) compared with traditional CPUs that commonly have only 4 or 8. That is why they are so widely used today, since most scientific algorithms spend most of their time doing just what GPUs are good for: performing billions of repetitive arithmetic operations.

Today, a GPU is one of the most crucial hardware components of computer architecture. Initially, the purpose of a video card was to take a stream of binary data from the central processor and render images to display. Over several decades, the GPU evolved from a single core, fixed function hardware used for graphics solely, to a set of programmable parallel cores.

In the last years, the memory bandwidth of GPUs has also increased by a tremendous amount in order to support architectures with smaller operational intensity (Flops to memory access ratio) , resulting in systems able to support even more paradigms. GPUs are increasingly designed as "super processors", capable of providing speed and raw computational power, compared with increasing programmability and customization.

We can see the huge increase in theoretical peak FLOP rate (fig. 1.1) and Memory bandwidth (fig. 1.2) the last 10 years. The explosive growth after 2013 suggests a GPU-centered future in the computational heavy research reality.

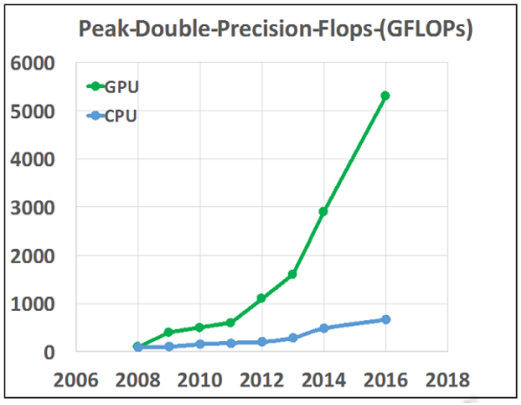


Figure 1.1: GPU-CPU GFLOPs increase.

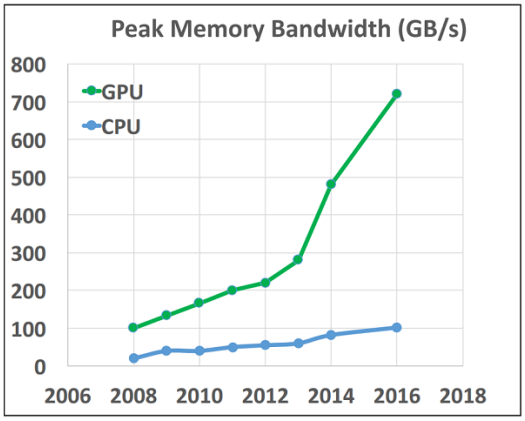


Figure 1.2: GPU-CPU memory bandwidth.

source: HPC wire

GPU architecture

To understand why SpMV format prediction is even more important in GPUs, a basic understanding of GPU architecture is required. Unlike today's multipurpose CPUs, GPUs still have certain limits in their programming ability in order to be efficient. All the SpMV implementations for GPUs try to better take advantage of these special characteristics they possess.

Modern CPUs are defined by their small number of powerful complex cores (2-32) with high frequency clocks (2-4 GHz), small register files and a small number of ALUs. They feature complicated control logic mechanisms like branch prediction and out-of-order execution and support

small scale simultaneous multi-threading, with thread scheduling handled by the OS. They support 3-4 levels of hidden memory, with relatively big capacities (MB), and an average central memory bandwidth. Their memory transaction speed optimization is based on memory prefetchers and cache coherent protocols for multi-threaded programs.

On the other hand, modern GPUs focus on parallelism rather than raw core power. They feature a large number of simpler cores with much lower frequencies. For example, Nvidia Tesla K40 consists of 2880 cores of 745-875 MHz each. They have a much bigger register file, and a huge number of simpler ALU units, but follow a much simpler control logic mechanism (no branch prediction or out-of-order execution). They are designed to support large-scale interleaved multi-threading, with thread switching handled entirely by the hardware resulting in nearly nonexistent context-switch cost. They support less levels of hidden memory, with much smaller capacities (KB), but feature a very high central memory bandwidth (288 GB/s for Tesla K40).

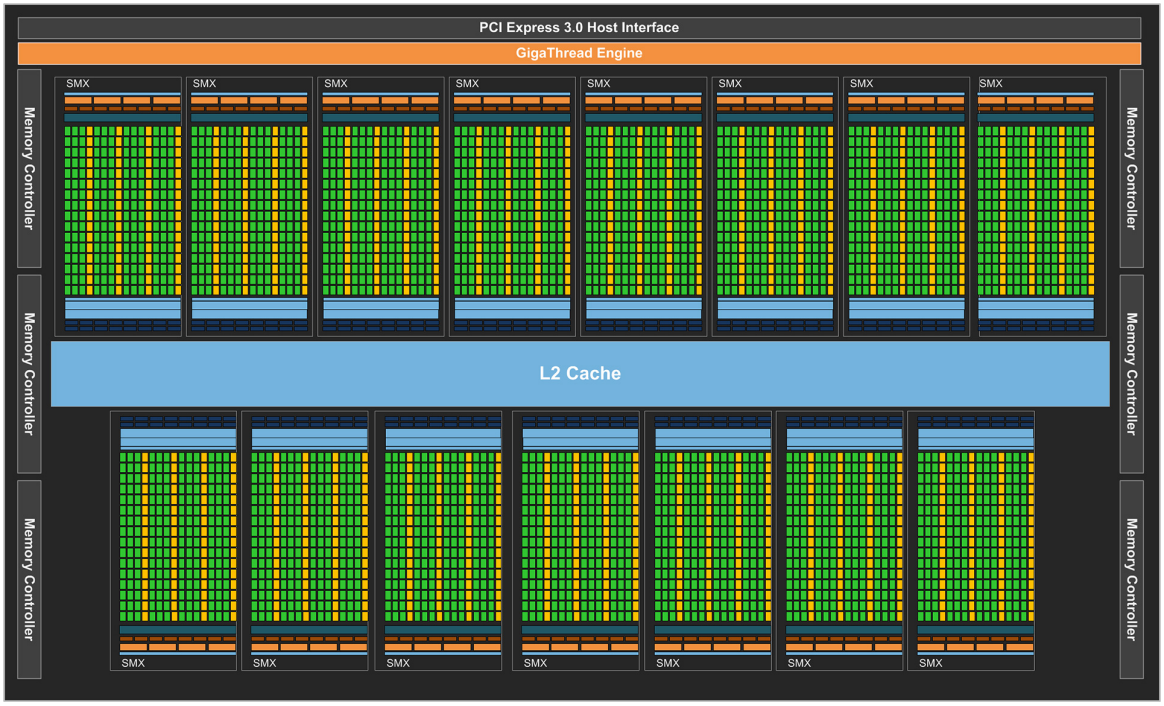


Figure 1.3: The Kepler architecture. *source: Nvidia CUDA programming guide*

We can see in fig. 1.3 the layout of the Streaming Multiprocessors in a Kepler GPU (K20X here). Each SM, addressed to as SMX in Kepler, features 192 single-precision CUDA cores, 64 double-precision units, 32 SFUs and 32 load/store units. Kepler supports the full IEEE 754-2008 compliant single- and double-precision operations like Fermi. A Kepler GPU features 13-15 SMXs (depending on the model) and six 64-bit memory controllers.

General-Purpose computation on Graphics Processing Units (GPGPU)

With the introduction of programmable GPUs at 2007, a new programming paradigm started to evolve. Until then the GPU was bound to certain uses, which where scheduled and executed by the system. With the introduction of GPUs that could be programmed, programmers were now able to write their own code and have it executed in the GPU. The unique architecture of GPUs lead to the founding of a new programming style, in order to compensate with the difficulties faced but also take advantage of the new supported perks. This was later known as General-Purpose computation on Graphics Processing Units (GPGPU).

In GPGPU things are quite different than a normal CPU-centered application. The application process starts executing on the CPU, named the "host system". Input, data management and Error checking (the whole data pre-processing) is handled in the host system. Then, the computation data is copied to the GPU memory and the parallel execution in the GPU starts. The host controls the code executed in the GPU by writing a main "device" function, which is executed parallel in the whole GPU. Additional "device" functions can be provided to the GPU, and be utilized by the main device program. Then, the output data is copied to the CPU and processed further by the host program.

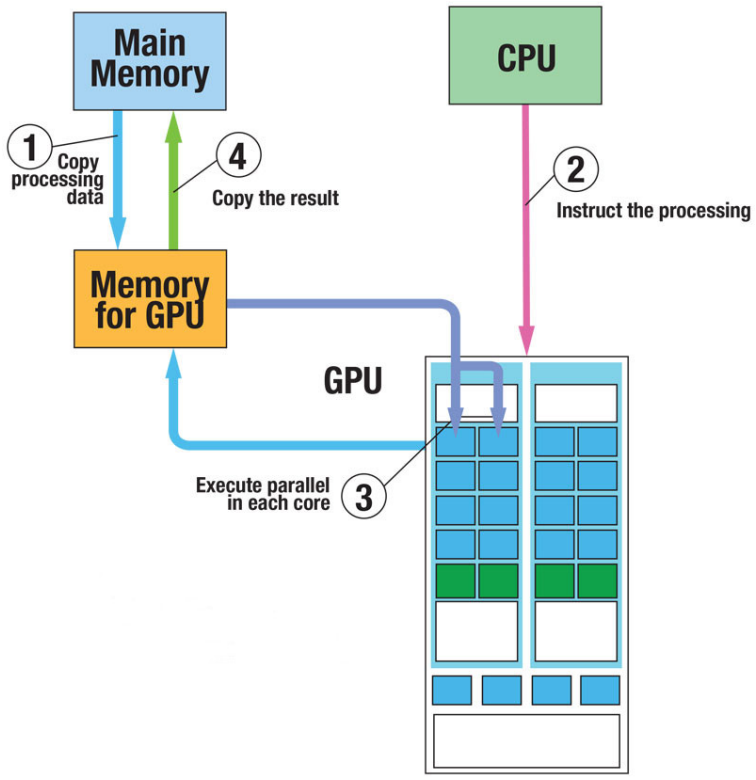


Figure 1.4: A graphic representation of the GPGPU paradigm. *source: cslab CUDA model presentation*

In fig 1.4 we can see the simple GPGPU paradigm described above. While many things have improved over the last years, the basis of this style remains the same. With the introduction of GPUs with greater compute capabilities though, some of these steps became easier or irrelevant. The best example is the introduction of Unified memory with CUDA 6.0 for GPUs of compute capability of 3.0 or greater. With Unified memory, copying the data to and from the GPU is handled

and optimized automatically by the system, resulting in much less work for the CUDA programmer.

1.1.1 The CUDA programming model

When Nvidia in 2007 released CUDA, its purpose was to give programmers advanced control over their GPU kernels, without uncovering the actual architecture of their GPUs. Since all our benchmarks, algorithms and Neural Network training were executed in a Nvidia Tesla K40, and for the Network training we used exclusively SpMV CUDA implementations we need to make a quick explanation of the CUDA programming model and its characteristics.

In CUDA, instead of actual processors and hardware threads, we have Streaming Multiprocessors and warps. Streaming Multiprocessors contain streaming processors (SPs-CUDA cores) and their own register file, shared memory, constant and texture cache. The number of CUDA cores in each SM is very big (For example 192 cores in 1.3), and each CUDA core can see the device memory, which can be accessed by all cores in the GPU, but has a shared memory bandwidth. Nevertheless, data in the device memory must contain all data that must be accessed by all the SMs.

On the other hand, each SM sees only his own shared memory and caches and has a exclusive register file. These much faster memories are reserved for exclusive data or read only operations. When a CUDA programmer creates device code, he must try to split the work in equal parts, computation and memory wise, in order to better utilize each SM's register file and shared memory.

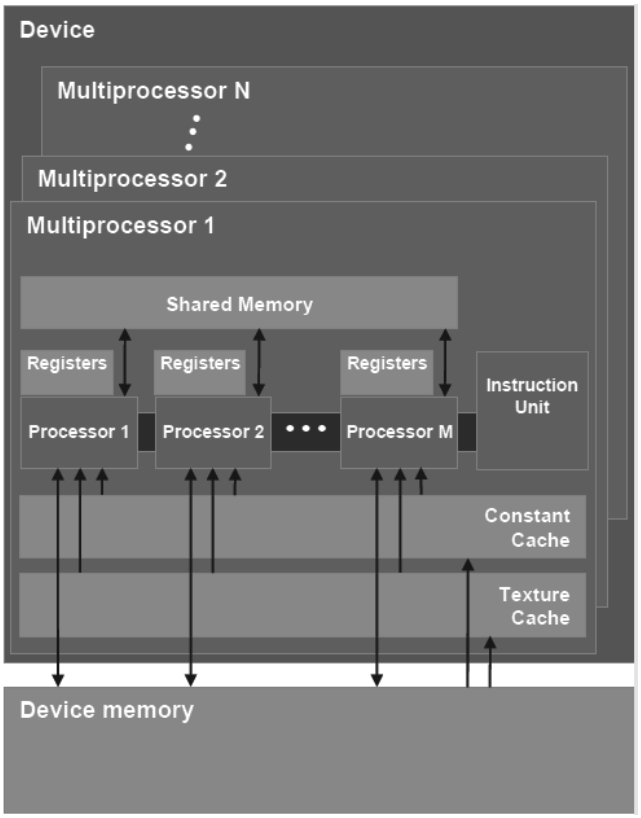


Figure 1.5: The Nvidia GPU CUDA architecture model. *source: Nvidia CUDA programming guide*

In the above fig 1.5 we can see the architecture model as derived by the CUDA programming model. To be precise, this is not the actual GPU architecture, but a "interaction" architecture for

the programmer. The actual architecture still remains hidden for Nvidia GPUs for competitive purposes.

Compute capability

With the appearance of GPGPU, GPUs started to greatly update their arsenal of commands, capabilities and programming options with each new release. To "count" the level of abilities each model possessed and allow programmers to customize their programs for different devices, Nvidia introduced with the release of the CUDA toolkit in 2007 the "compute capability" metric for its GPUs. In the compute capability model, each device has a number in the form of <major>.<minor>, which defines its model and operation support. The <major> number is the core architecture of the said device, while the <minor> are possible improvements in the core architecture this device implements. The Nvidia Tesla K40 has a compute capability of 3.5.

CUDA warps and half-warps

Since the CUDA model is a SIMD based programming model the independent use of single CUDA cores is meaningless, in each SM the threads are grouped in teams of 32, named "**warps**". Each warp in a SM is controlled by a warp scheduler, who assigns the CUDA cores to a warp with nearly 0 overhead. Each warp sees the same resources with all the other warps in the SM, but can't see the ones from warps of other SMs. To keep the GPU parallel SIMD logic, each core in every warp must always execute the exact same command with all the other warp threads at each clock circle. It is thus very important for the CUDA programmer to create warp-friendly SIMD code in order to avoid inactive warps (which is also one of the major problems with SpMV in GPUs).

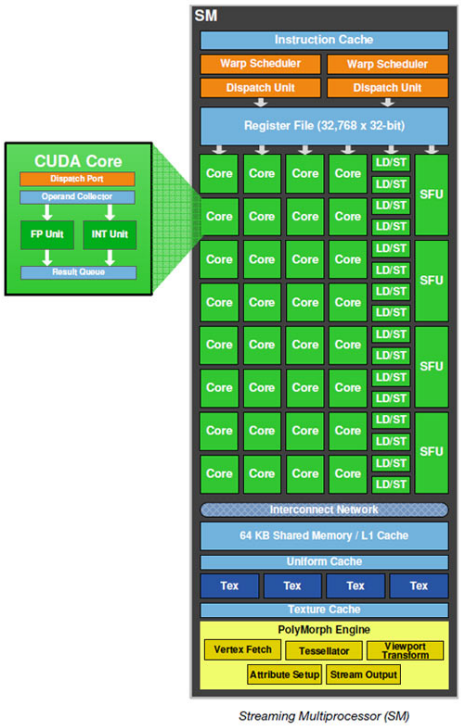


Figure 1.6: The Nvidia SM CUDA architecture. *source: Nvidia CUDA programming guide*

1.2 Matrix Vector Multiplication

In mathematics, matrix multiplication or matrix product is a binary operation that produces a matrix from two matrices. The matrix vector product is a subcategory of matrix multiplication, where the second matrix is a single-column matrix. It is one of the most basic tools of linear algebra, required in almost all eigenvalue operations, and thus a very common one in many scientific fields like chemistry, biology and engineering.

The Matrix Vector Product Formula

To define multiplication between a matrix A and a vector x , we need to view the vector x as a single-column matrix. The operation $A \cdot x$ can be defined only if the number of columns of A equals the number of rows of x . If that is true, we can define for a $m \times n$ matrix A and a $n \times 1$ vector x the operation of $A \cdot x = b$ which results to a new single-column matrix b of size $m \times 1$. Therefore the size of vector b depends only in in the number of rows m of matrix A. The general formula for computing the above matrix-vector product is

$$Ax = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}$$

Therefore the computation of the matrix vector product consists of m identical domestic product operations between every row of A and the whole x vector. The algorithm of Matrix-Vector multiplication (which we will from now on refer to us DMV) is based on this property.

The Matrix Vector Product Algorithm

The general algorithm of the matrix vector multiplication is a very simple, direct implementation of the above formula.

Algorithm 1: Matrix Vector Product

Input: Matrix $A[m][n]$, Vector $x[n]$
Output: Vector $y[m]$

```
1 for i = 0; i < m; i++ do
2     y[i] = 0;
3     for j = 0; j < n; j++ do
4         y[i] = y[i] + A[i][j] * x[j];
5     end
6 end
```

The algorithm consists of an outer for loop, which iterates through the rows of A, and then from each row a domestic product with the input vector x is computed. The inner loop requires n multiplications and $n-1$ additions, thus resulting in inner $\Theta(n)$ complexity. The outer loop is then executed for each of the m rows, resulting in a final complexity of $\Theta(nm)$.

1.2.1 Parallel Implementations of the Kernel

The most important characteristic of the matrix vector multiplication algorithm is that there is no dependency between the computation of the m domestic products. That, along with the fact that sometimes the matrix A is large, makes it ideal for parallelization. There are many different approaches regarding the subject, but we only need a basic understanding of the ideas behind each one, since most also partially apply in the SpMV kernel.

Input Matrix Decomposition

The basic idea is to split A between threads of execution, and let each one compute a different part of y. The simplest way to do that is by dividing the rows of A in $p = parallelthreadsofexecution$ groups, and then each thread of execution computing the corresponding $y[m/p]$ part, as seen in the first array of figure 1.7. This is called the "row based distribution", and is the most common way to compute the matrix vector product in GPUs.

Another equal work distribution is the so called "column based decomposition". As the name suggests, each of the p threads of execution is now responsible for a number of columns of A equal to n/p, as seen in the middle array of figure 1.7. With this approach, either all threads of execution write to the same vector y, so a lock is required for the critical part (line 6 of Algorithm 2), or each thread of execution ends up with a sub-copy of the output y vector, and the final vector is computed by gathering and adding all the sub-vectors. Column based decomposition can work in multi-node environments with tools like MPI (Message Passing Interface), where gathering the output vector can be done efficiently, but is inferior for SIMD architectures.

Algorithm 2: Column Based Matrix Vector Product

Input: Matrix $A[m][n]$, Vector $x[n]$
Output: Vector $y[m]$

```
1 for i = 0; i < m; i++ do
2     y[i] = 0;
3 end
4 for j = 0; j < n; j++ do
5     for i = 0; i < m; i++ do
6         y[i] = y[i] + A[i][j] * x[j];
7     end
8 end
```

Finally, there is also the combination of the two, where each thread of execution is now responsible for a equal block of A (right array of figure 1.7), and is technically like executing $n \cdot m \div p$ different matrix vector multiplication kernels. This is also better for multi-node environments with the use of MPI for scattering the input matrix A and gathering the y vector.

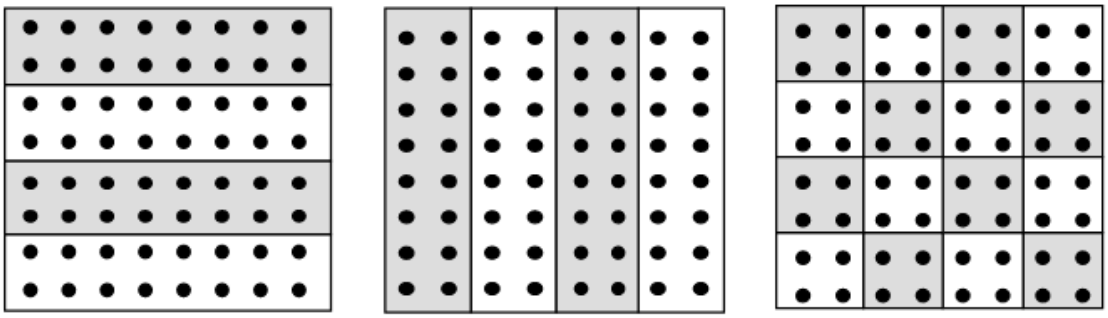


Figure 1.7: The three most widely used decomposition schemes for input matrix A

Input Vector Distribution

Regarding the input vector x, there are two possible store methods. The first is for each thread of execution to have only the required part of x for the computation it is responsible, which is ideal if we use a column based distribution for the input matrix A, . This method is impossible for row based distributions, since they require the whole x vector. As a result, in such cases each thread of execution must have read access to the whole x vector. In shared memory architectures this is quite simple and efficient, but in multi-node non-shared memory ones the remote read cost is most of the times prohibitive. In such cases, the x vector is copied to each thread's local memory, since its size n is relatively small compared to the $m \cdot n$ of A.

In GPUs, since in SpMV the x vector is not sparse by default, it is most of the times stored in the device shared or texture memory (basically in the fastest memory available for read only instructions) in order to increase access speed. More details about the SpMV kernel implementations considered in this thesis will be explained in the next sections.

1.3 Sparse Matrix Vector Multiplication (SpMV)

Solving sparse linear systems has been an important part of a wide variety of scientific applications since the early days of computing systems. The utilization and optimization of their unusual sparse nature has been a research target for years. This has lead to the development of a wide variety of methods, both direct and iterative, over the last decades, whose performance is vital nowadays to a wide variety of HPC applications. Although these applications originate from different scientific fields, they all share an intersection in terms of sparse linear system solvers, spending, in many cases, most of the execution time on solving such systems.

These last years, the GPUs capabilities have also increased greatly, resulting in a renewed interest for SpMV research. But, as we explained, GPUs require certain characteristics in order to be fully utilized by a program, and SpMV lacks some of them. In this chapter, we briefly present the different methods for solving large sparse linear systems and designate the importance of selecting the ideal Sparse Matrix-Vector Multiplication kernel for each input matrix, which is the focus of this diploma thesis.

1.3.1 Sparse Matrices

Many different definitions have been given about the exact nature and density of a so called "Sparse Matrix". In general, a sparse matrix is defined as a matrix populated primarily by zeros. The sparsity of a matrix is defined as the number of zero elements divided by the total number of elements in the full matrix. For example, a sparsity value of 95% means that nearly 95% of the elements in the matrix are zeros. In many cases the sparsity value is a little over 99%.There is no particular sparsity value after which a matrix is recognized as sparse, but in most cases a sparsity value of $Sp < n$ for an $n \times n$ matrix is required.

Sparse matrices can be found in scientific applications that use finite difference, finite element or finite volume discretizations of partial differential equations (PDEs) in problems with underlying 2D or 3D geometry coming from computational fluid dynamics, electromagnetics, thermodynamics, materials, semiconductor devices, model reduction and structural mechanics, but also in other application domains where problems do not seem to have such a geometry, as in electrical circuit simulation, chemical process simulation, economic and financial modeling, power networks and graphs. For example, in circuit simulation, the elements of a sparse matrix may represent circuit voltages, currents, impedances and power sources, in model reduction the elements of a sparse matrix may represent the porous micro-architecture of a human bone, after applying a finite element method, and in most graph problems the adjacency matrix, which contains the edges of the graph, is sparse.

Sparse matrices can be categorized in two basic classes: structured and unstructured. A structured matrix is any matrix that has nonzero entries which form a pattern. This pattern can take many forms, with the most usual being the diagonal pattern (nonzero elements only in/around the diagonal). Matrices that follow this pattern are usually called banded matrices, or bands. On the other hand, unstructured matrices comprise of elements which are irregularly located. Of course, the two classes usually overlap; for example unstructured matrices sometimes contain one or more bands in their diagonal. The distinction between the two types of matrices is very important to the

performance of the SpMV kernel, since their performance may significantly differ, depending on whether the matrix is structured or not.

Of course, the matrix class is not the only information needed in order to select the best format for a matrix. As we mentioned before, some matrices contain parts from both classes. Also, even for matrices of the same class (especially of the unstructured class), both the performance and the ideal format may differ. That is because other metrics like sparsity level, average nonzeros per row, block count and many more also affect performance. A lot of scientific research has been focused to that end, but no "perfect" method has been implemented for the classification of these matrix characteristics. This is the motivation, the idea and the target of this thesis; to implement a method of such nature with the use of deep learning networks.

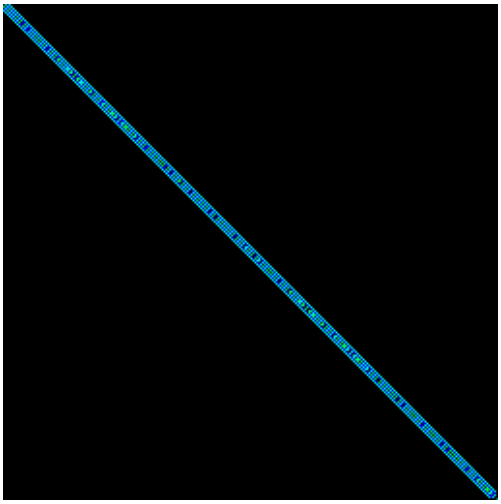


Figure 1.8: Apache1

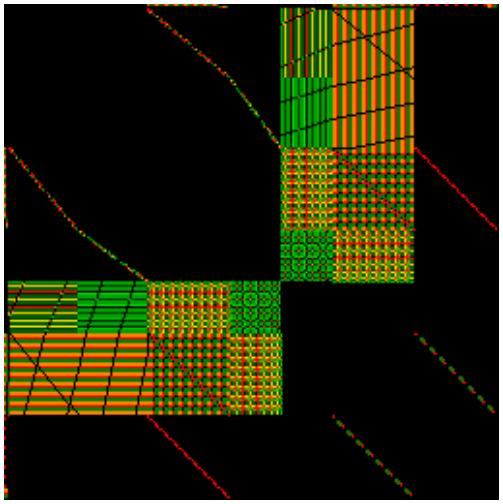


Figure 1.9: net50

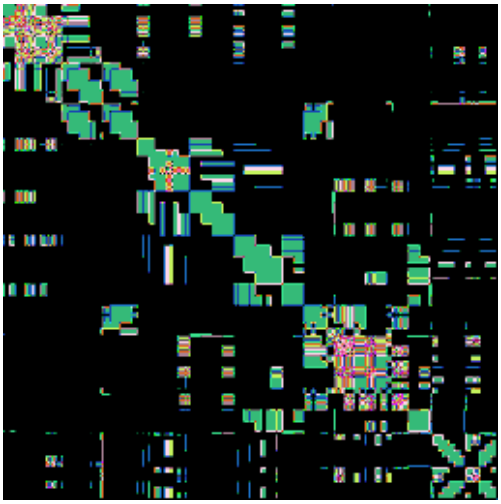


Figure 1.10: heart3

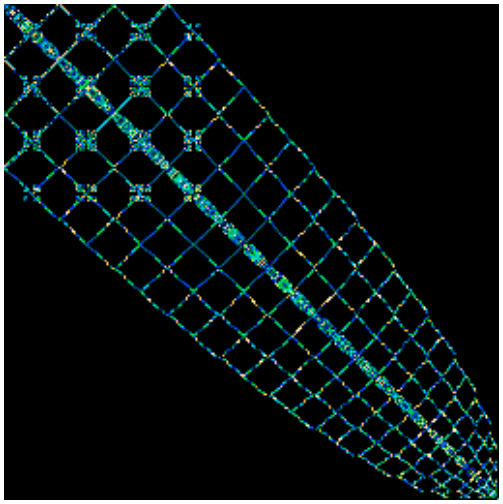


Figure 1.11: Si5H12

In fig. 1.8 we can see an example of a structured matrix. Specifically here we see an SPD matrix (finite difference 3D) from APACHE small. In contrast, we can see in 1.9 an unstructured matrix of a graph optimization problem; following we have in 1.10 an unstructured Quasi-static FE model of a heart; and finally, in 1.11 we see a sparse matrix of the Real-space pseudo-potential method. All these matrices originated from the Florida SuiteSparse matrix market collection.

1.3.2 Sparse linear systems

After introducing the sparse matrices classes and characteristics, we have to explain **where** these matrices are used and why optimizing SpMV is among the most important steps to optimizing the whole scientific applications which contain it. Sparse matrices are usually involved in the solution of large linear systems of the form $A \cdot x = b$ where A denotes the coefficient matrix, b is the right-hand side vector, and x is the output vector. There are two basic classes of methods for solving linear systems of the above form.

Direct Methods

The first class is represented by direct methods, which theoretically yield an exact solution in a (predictable) finite number of steps. In practice, of course, the solution obtained will be contaminated by the round-off error that is involved with the arithmetic being used. Such methods include the Gaussian Elimination Method, the LU Decomposition Method and the Cholesky Method [Duff et al., 1989, Barrett et al., 1994, Davis, 2006]. These methods rely on the factorization of the coefficient matrix as a product of three matrices $A = L \cdot D \cdot U$, where L and U are lower and upper triangular respectively and D is diagonal. Then, the solution of the system comes down to solving two easily invertible triangular systems $L \cdot y = b$ and $U \cdot x = D^{-1} \cdot y$. Forward elimination followed by backward substitutions complete the solution process.

It is vital to understand the importance of Direct methods. Because of their generality and robustness, they are ideal for "tough" linear systems arising in some applications (e.g., circuit simulation) where the importance of precision makes them the only feasible solution. Furthermore, they provide an effective means of solving multiple systems with the same coefficient matrix A but different right-hand side vectors, because the factorization only needs to be performed once. On the downside, the asymptotic time complexity of all dense direct methods is $O(n^3)$ for the factorization and $O(n^2)$ for solving the system based on the precomputed factorization. When the dimension of the coefficient matrix is in the order on 10^5 , 10^6 or more, the total cubic complexity is prohibitive. Furthermore, the elimination process of these methods may introduce fill-in, i.e., matrices L and U may have nonzero elements in locations where the original matrix A has zeros, complicating the solution of sparse systems as well. This is why most methods introduce a fill-in minimization step during the and factorization process in order to increase the sparsity of matrices L and U.

Iterative Methods

The complexity and computational difficulties of the direct method lead to the introduction of approximation methods for solving linear systems. These include iterative techniques that try to find an approximate solution, but are much faster than direct methods. There are two broad categories of iterative methods: stationary methods and projection methods. Stationary methods begin with a given approximate solution of the system and iteratively modify step by step the components of the approximation, one or a few at a time until convergence is reached. These modifications, called relaxation steps, usually aim to annihilate some component(s) of the residual vector $b–A{\cdot}x$. The most common stationary methods are the Jacobi, Gauss-Seidel and the Successive Over-relaxation

(SOR) methods. The convergence of these methods cannot be guaranteed for all matrices and therefore they are rarely used separately [Saad, 2003].

Projection methods try to extract an approximate solution x of a linear system from a subspace of $\Re^n$. If K_m is this subspace of candidate approximates where m denotes its dimension, then, in general, m constraints must be imposed on the residual vector $b–A{\cdot}x$ to be able to extract such an approximation. More specifically, the residual vector must be orthogonal to m linear independent vectors, which form the subspace of constraints L_m . At each step of a projection method a new pair of K_m and L_m sub-spaces is used with an initial guess x_o equal to the approximation obtained from the previous step. Some of the most important iterative techniques for solving large linear systems are the projection methods known as the **Krylov subspace methods**. A Krylov subspace method is a method for which the subspace K_m is the Krylov subspace:

$Km(A, r_0)$ = span{ $r_0, A{\cdot}r_0, A^2{\cdot}r_0, ..., A^{m-1}{\cdot}r_0$ }

where $r_0 = b–A{\cdot}x_0$, is the residual of the initial guess x_o . Different versions of Krylov subspace methods arise from different choices of the subspace L_m and from different preconditioning techniques applied to the original system. Preconditioning involves applying a transformation, called the preconditioner (which is usually a direct solution method), on a system in order to make it more suitable for numerical solution by improving its convergence characteristics. It is the key ingredient to the success of Krylov subspace methods when applied on systems derived from large "real-world" problems, whose convergence ratio would otherwise be very low.

Preconditioned Krylov subspace methods, such as the Generalized Minimum Residual (GMRES) method [Saad and Schultz, 1986] and the Conjugate Gradient (CG) method [Hestenes and Stiefel, 1952], are widely used nowadays for solving large sparse linear systems. From a computational point of view, and setting aside the preconditioning process, these methods rely mainly on the following computational kernels:

Vector Updates

Operations of the form $y = y + a{\cdot}x$ where a is a scalar and y and x are vectors, are known as vector updates or SAXPY operations (Scalar Alpha X Plus Y according to the naming conventions of the Basic Linear Algebra Subprograms package, or simply BLAS).

Dot Products

The dot product is the inner product of two vectors x and y : $t = x^T{\cdot}y$

Matrix-by-Vector Products

This is the product of a sparse matrix A (the coefficient matrix of the system) and a vector x : $y = A{\cdot}x$

The performance of the iterative method strongly depends on implementing efficiently the above computational kernels on the target architecture. Vector updates and dot products have been optimized to perform greatly both on multi-core CPUs and GPUs. The matrix-vector product on the other side, since the multiplied Matrix is always sparse, has a very low operational intensity (FLOPs/Memory ratio per the roofline model [Williams et al., 2009]); it ends up one of then most expensive operations involved in iterative solvers. As seen in Fig. 1.12, the SPMV kernel is the

slower part of the CG solver, taking around 80-90% of its total execution time. It is therefore the most important target for optimization in such solvers.

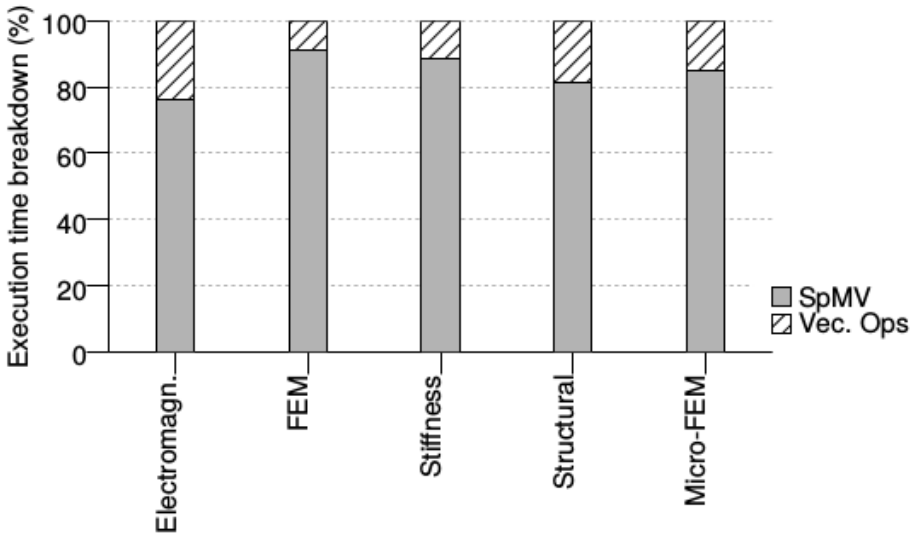


Figure 1.12: Execution time breakdown for the non-preconditioned CG iterative method for different problem categories [Karakasis, 2012]

1.3.3 The SpMV algorithm

In terms of simplicity, the SpMV kernel is quite similar to the DMV kernel. The main difference is that matrix data is stored using a compact format, in order to decrease the dataset size, resulting in an $O(nz)$ complexity compared with the $O(n^2)$ of DMV, where nz is the number of non-zero elements in the input matrix. Compared to a dense implementation the SpMV kernel is genuinely faster for high sparsity levels. Furthermore, for problems that contain huge sparse matrices DMV is out of the question.

Thus, while SpMV greatly decreases the complexity of the DMV, a new problem arises. The SpMV kernel's input is a list of the non-zero elements of matrix A and their locations; resulting in imbalanced accesses for the vector x. In addition, like DMV, the operational intensity of the kernel is low. These two problems make the SpMV kernel memory bound in problem sizes where the input size exceeds the local cache hierarchy size. That means that especially in GPUs, which are designed to produce a huge FLOPs rate, all this computational power is "blocked" because the memory can't keep up. This exact problem is the motivation for all SpMV research, and has lead to the implementation of a huge number of storage formats for the input matrix, in order to reduce memory operations as much as possible. Below we will explain the most basic of these formats, in order to better understand why classification between them is so important for a given problem.

1.3.4 Basic SpMV formats

In this chapter we will explain the basic SpMV storage formats and analyze each ones strengths and weaknesses. The selection of the right format for each matrix can greatly improve performance; it

is very important to understand **when** and **why** this happens, since it is the purpose and motivation of this thesis.

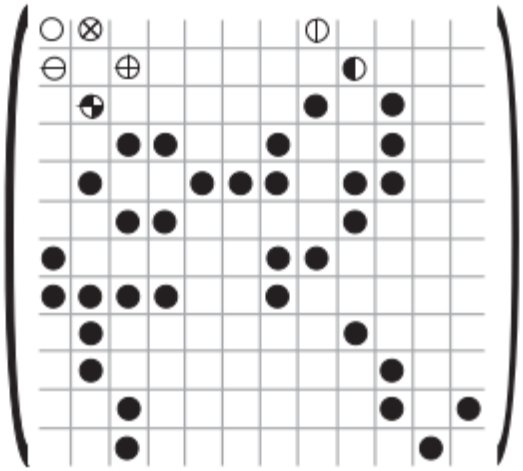


Figure 1.13: An example sparse matrix A [Filippone et al., 2017]

Coordinate format (COO)

The first and simplest sparse matrix storage format is the COO format. It is the most obvious storage scheme, defined by the three scalars M, N, and NZ and the three arrays IA, JA, and AS. All three arrays have size nz, and the AS array contains all the non-zero elements while the IA and JA arrays contain each one's exact coordinates in the A matrix (hence the format name). The basic COO SpMV kernel for the product $y = A \cdot x$ is shown in Algorithm 3; it costs five memory reads, one memory write, and two floating-point operations per iteration, that is, per nonzero coefficient. While the x vector normally resides in the GPU shared memory or constant cache, the AS value list must be fetched at each iteration and thus results in increased memory movement. Note that the code will produce the result y even if the coefficients and their indices appear in an arbitrary order inside the COO data structure, since the IA, JA and AS arrays define each of their element individually.

Algorithm 3: COO SpMV kernel

Input: Row vector $IA[nz]$, Column vector $JA[nz]$, Value vector $AS[nz]$, Vector $x[n]$

Output: Vector $y[m]$ (Initialized to 0 at the beginning)

```
1 for i = 0; i < nz; i++ do
2     y[IA[i]] = y[IA[i]] + AS[i] * x[JA[i]];
3 end
```

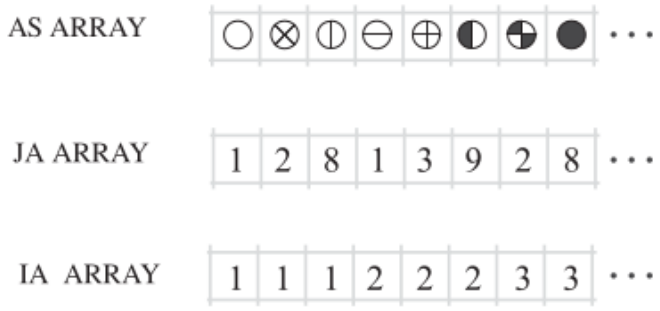


Figure 1.14: COO format of sparse matrix A in 1.13

Compressed Sparse Row format (CSR)

Next is the CSR format ,the most popular format for sparse matrix representation. It is almost the same with the COO format, with the only difference that the row indices are stored in a pointer-like fashion. Instead of having the IA vector, a vector IRP is used instead, containing the "boundaries" of each row; thus requiring only $M + 1$ elements instead of NZ. In contrast with the COO format, the two vectors JA and IRP must be sorted in order for the format to work (to be precise, the elements of JA must be in the right IRP boundary, bot could possibly be unsorted internally). The naive CSR SpMV kernel for the product $y = A \cdot x$ is shown in Algorithm 4; it requires three memory reads and two floating-point operations per iteration of the inner loop, that is, per nonzero coefficient; the cost of the access to $x[JA[j]]$ is highly dependent on the matrix pattern and on its interaction with the memory hierarchy and cache structure. That is the biggest challenge in SpMV, and is especially important in SIMD architectures like GPUs. In addition, each iteration of the outer loop, that is, each row of the matrix, requires reading the pointer values $IRP[i]$ and $IRP[i+1]$, with one of them available from the previous iteration and one memory write for the result.

Algorithm 4: CSR SpMV kernel

Input: Row ptr vector $IRP[m]$, Column vector $JA[nz]$, Value vector $AS[nz]$, Vector $x[n]$
Output: Vector $y[m]$ (Initialized to 0 at the beginning)

```
1 for i = 0; i < m; i++ do
2     ytemp = 0;
3     for j = IRP[i]; j < IRP[i + 1]; i++ do
4         ytemp = ytemp + AS[j] * x[JA[j]];
5     end
6     y[i] = ytemp;
7 end
```

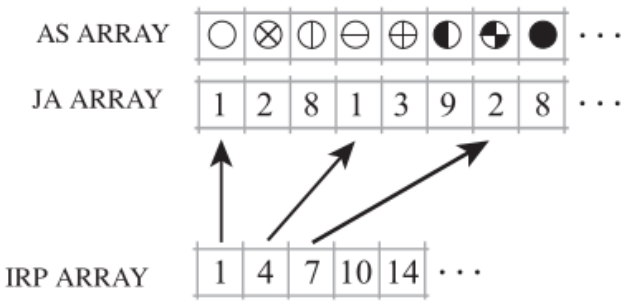


Figure 1.15: CSR format of sparse matrix A in 1.13

Almost identical to CSR is the CSC format, which is basically a column based CSR representation. Since it is rarely used in iterative solvers and also goes against the SIMD logic of GPUs, we are not going to expand further.

Diagonal format (DIA)

Since the biggest challenge in SpMV was to reduce memory traffic and utilize the hidden memory hierarchy, many storage formats focus in exactly that; These are formats that focus on performing well on specific matrices, and are usually much slower on the general case. Their existence arrised from the need to perform SpMV very fast in specific scientific problems where the input matrix was bound to have certain characteristics.

The best example of such a format is the Diagonal (DIA) format. Its name defines the kind of arrays it was designed for; arrays that have elements only (or explicitly very close) to the diagonal, which is the most common structured sparse pattern. The DIA format is defined by a two-dimensional array AS containing in each column the coefficients along a diagonal of the matrix and an integer array $OFFSET$ that determines where each diagonal starts. The diagonals in AS are padded with zeros as necessary. The DIA SpMV kernel for the product $y = A \cdot x$ is shown in Algorithm 5; it costs one memory read per outer iteration, plus three memory reads, one memory write, and two floating-point operations per inner iteration. The accesses to AS and x are in strict sequential order; therefore, no indirect addressing is required. This is the basis on which the DIA format performance boost is based; which of course works only on specific matrices.

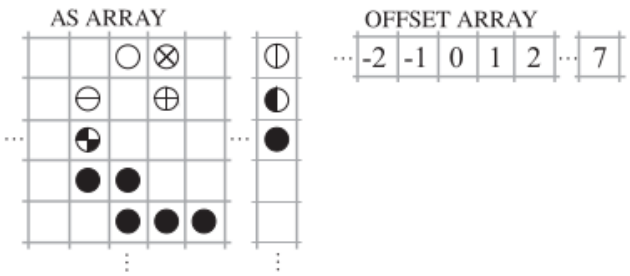


Figure 1.16: DIA format of sparse matrix A in 1.13

Algorithm 5: DIA SpMV kernel

Input: Value array $AS[nz_{padded}]$, Offset vector $OFFSET[ndiag]$, Vector $x[n]$

Output: Vector $y[m]$ (Initialized to 0 at the beginning)

```
1 for j = 0; j < ndiag; j++ do
2     if OFFSET[j] > 0 then
3         ir1 = 1; ir2 = m − OFFSET[j];
4     else
5         ir1 = 1 − OFFSET[j]; ir2 = m ;
6     for i = ir1; i < ir2; i++ do
7         y[i] = y[i] + AS[i, j] ∗ x[i + OFFSET[j]];
8     end
9 end
```

Ellpack format (ELL)

The other format designed specifically for vector machines (SIMD logic-based execution), is the ELLPACK (ELL) format. The normal ELLPACK/ITPACK format consists of two two-dimensional arrays AS and JA with M rows and $MAXNZR$ columns, where $MAXNZR$ is the maximum number of nonzeros in any row (which therefore must be known for the target array). Each row of the arrays AS and JA contains the coefficients and column indices; rows shorter than $MAXNZR$ are padded with zero coefficients and appropriate column indices (e.g., the last valid one found in the same row) [Filippone et al., 2017]. The ELLPACK SpMV kernel for the product $y = A \cdot x$ is shown in Algorithm 6; it costs one memory write per outer iteration, plus three memory reads and two floating-point operations per inner iteration. Unless all rows have exactly the same number of nonzeros, some of the coefficients in the AS array will be zeros; therefore, this data structure will have an overhead both in terms of memory space and redundant operations (multiplications by zero).

Algorithm 6: ELL SpMV kernel

Input: Value array $AS[maxnzr * m]$, Indices array $JA[maxnzr * m]$, Vector $x[n]$

Output: Vector $y[m]$ (Initialized to 0 at the beginning)

```
1 for i = 0; i < m; i++ do
2     ytemp = 0;
3     for j = 1; j < maxnzr; j++ do
4         ytemp = ytemp + AS[i, j] ∗ x[JA[i, j]];
5     end
6     y[i] = ytemp;
7 end
```

This is why the ELLPACK format performs better in "balanced" arrays, where each column's nz_c elements are close to $MAXNZR$. In such cases, while it will still probably contain more operations than a simple format, by using its structural integrity it can achieve noticeable speedups with vectorization and cache utilization. On the other hand, in the extreme case where the input matrix has one full row, the ELLPACK structure would require even more memory than the normal 2D array storage. This is why in ELL formats and variations, an affinity check is required before transforming from an other format to ELL, in order to avoid the very "bad" matrices, which would result in execution even slower than dense matrix-multiplication.

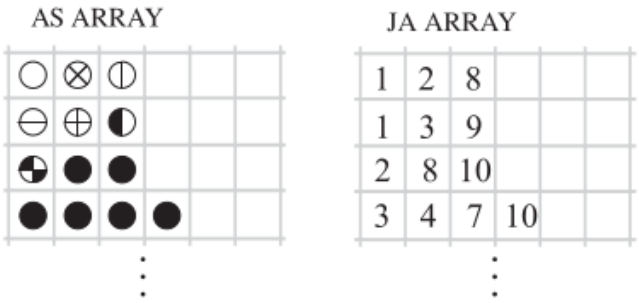


Figure 1.17: ELL format of sparse matrix A in 1.13

Block Compressed Row format (BSR/BCSR)

This format is tecnically the same with CSR, with the only difference being that it stores blocks instead of single elements; since blocked sparse arrays are very common in most scientific applications, BSR is one of the most common non-generic kernel used for SpMV [Im and Yelick, 2001]. In BSR data is stored in fixed-size aligned $r \times c$ blocks instead of single nonzero elements. The JA structure is replaced by BJA, which stores the column index of the first element of each block, while the IRP structure is replaced by $BIRP$ which points to the start of each block-row. The nonzero elements are stored block-wise in bvalues and follow a row-major ordering. Although the BSR format reduces the size of the indexing information and allows for common optimizations, such as register blocking, loop unrolling and vectorization, it usually introduces unnecessary padding in the bvalues array in order to construct full blocks. Thus, for matrices without a favorable nonzero element pattern, the final memory requirements of the matrix might exceed those of the CSR format, resulting in considerable performance degradation.

Hybrid format (HYB)

While the ELLPACK format is well-suited to vector and SIMD architectures, its efficiency rapidly degrades when the number of nonzeros per matrix row varies. In contrast, the storage efficiency of the COO format is invariant to the distribution of nonzeros per row, and the use of segmented reduction makes its performance largely invariant as well. To obtain the advantages of both, the hybrid ELL/COO format was implemented. The hybrid (HYB) format stores the typical number of nonzeros per row in the ELL data structure for average rows, and the remaining entries of exceptional rows in the COO format. Determining which rows are "regular enough" to be stored in ELL format and which will be left in COO depends on each specific hybrid implementation. In most cases, the splitting is based in some performance weight function, which tries to optimize execution time, taking also into account the ELL pre-processing cost. As a result, the HYB format is designed to perform well in all cases, but to lack the structure-targeted optimization of formats like DIA or BSR.

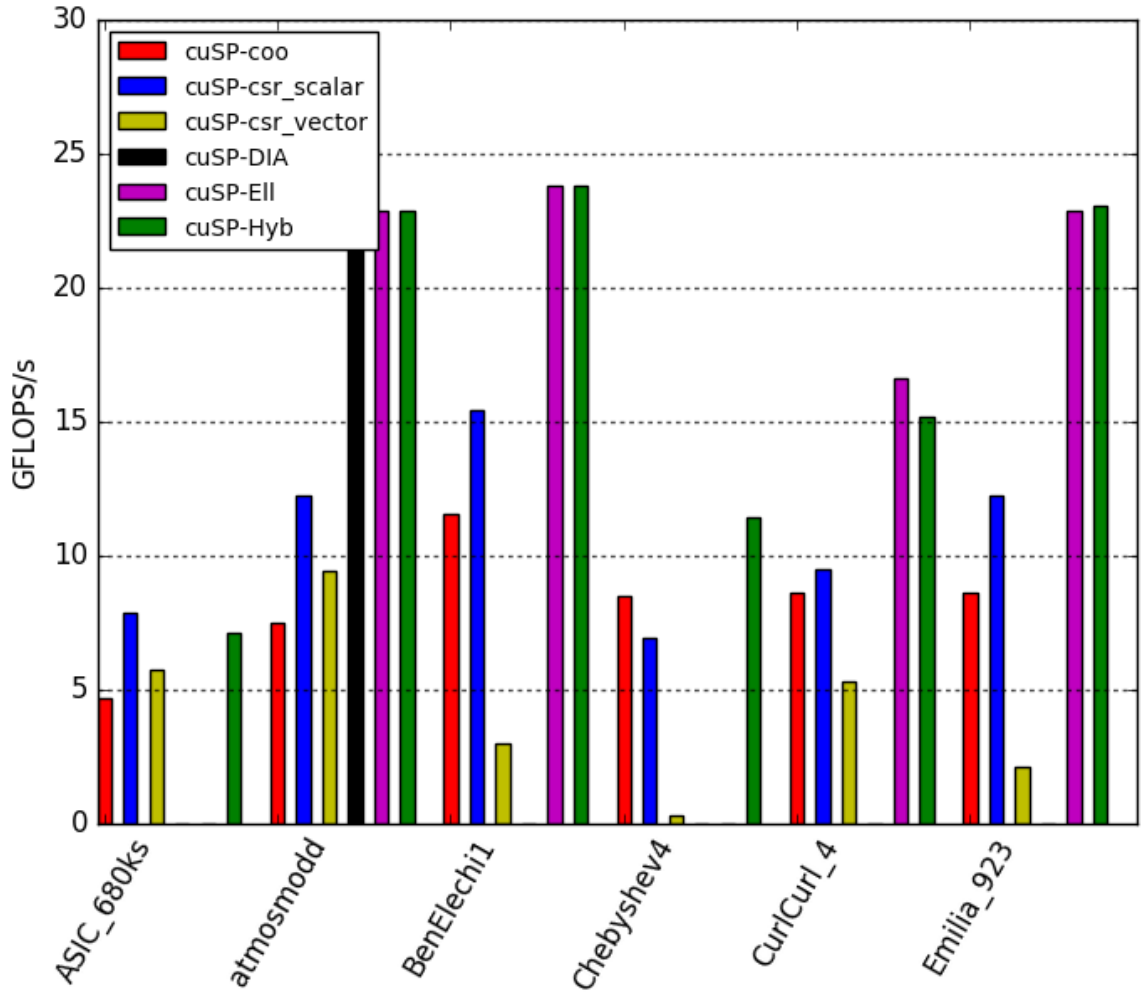


Figure 1.18: An example benchmark of some SpMV formats using the CuSP library for SpMV in GPUs.

1.3.5 SpMV in GPUs

With the popularity of GPGPU increasing, a lot of research has focused in targeted SpMV kernels for GPUs. Since GPUs can offer great parallelism, considerable bandwidths and are widely used in almost all scientific calculations and algorithms, optimizing SpMV especially for them is essential; and in some cases even specialized SpMV kernels for certain structural matrices are required. This is why during the last years, a lot of formats and techniques which try to utilize the SIMD logic of graphic processors have been implemented; in our approach, we use some of the state-of-the-art algorithms presented in this section, in order to produce results that are not only of research interest, but could have a practical use too.

The first library we are going to use is the **Nvidia CUDA Sparse Matrix library** (cuSPARSE). CUSPARSEs purpose is to provide GPU-accelerated basic linear algebra subroutines for sparse matrices and is widely used by engineers and scientists working on applications such as machine learning and natural language processing, computational fluid dynamics, seismic exploration and computational sciences. It is a closed source project by Nvidia, containing some of the most GPU-optimized implementations for the CSR, BSR and HYB formats (which we αre going to use); which are frequently updated by Nvidia when new methods and capabilities arise, and run only on Nvidia GPUs (since they require the CUDA toolkit). They support both single and double precision for all operations, and operate with the same CUDA infrastructure as most Nvidia GPU implementations.

Since cuSPARSE purpose is to focus on fast, all around good algorithms that run well on most matrices, it does not contain more specialized formats like ELL or DIA. For this purpose, and in order to also compare all methods format wise, we use the **cuSP library**. Also implemented by Nvidia, the cuSP library is an open-source cuSPARSE equivalent, focused on providing users a way to actually interact with and change their SpMV kernel device code to fit it with their needs. It is currently maintained on github and is available for all users. Unlike cuSPARSE, its performance is not state-of-the-art, and its implementations are much simpler to interact with. Apart from that, it also contains some COO, DIA and ELL implementations which we will use in our benchmarks, and one of the most advanced MMF serial readers available.

In GPUs and all vector machines in general, one of the most widely used formats is CSR. While SpMV research has expanded in all normal formats, there is a considerable amount of focused research on lightweight-preprocessing CSR implementations . Still, the CSR row pointer logic has a lot of divergence, imbalance and memory access problems. All CSR implementations for GPUs thus try to face these problems; some focusing in eliminating some of them while others trying to reduce the size of impact they have on performance. In our approach we chose 3 of them, each trying a different approach, in order to have our network recognize each one's strengths and weaknesses and train accordingly.

Compressed Sparse Row 5 format (CSR5)

The CSR5 format is among the state-of-the-art implementations for SpMV. It offers high-throughput SpMV on various platforms including CPUs, GPUs and Xeon Phi; its solid performance is based on the fact that it is insensitive to the sparsity structure of the input matrix. Thus the single format can support an SpMV algorithm that is efficient both for regular matrices and for irregular matrices. Furthermore, the overhead of the format conversion from the CSR to the CSR5 can be as low as the cost of a few SpMV operations, proving it ideal even for processes with limited SpMV iterations [Liu and Vinter, 2015].

CSR5 is a input format-based optimization. The first step is evenly partition all nonzero entries to multiple 2D tiles of the same size. Thus when executing parallel SpMV operation, a compute core can consume one or more 2D tiles, and each SIMD lane of the core can deal with one column of a tile. That leads to a group of 2D tiles composing the CSR5 core. The tuning parameters of each tile are ω and σ, where ω is a tile's width and σ is its height. In addition, for each tile there is a tile pointer $tile_{ptr}$ and a tile descriptor $tile_{desc}$. While ω is based solely on the target architecture, σ is computed after taking consideration of several parameters, like sparsity, matrix pattern and cache hierarchy. In 1.19 we can see a visual example of the CSR5 partition.

The CSR5 kernel algorithm is quite similar to the simple CSR; it includes the tileptr and tiledesc logic in the computation of the kernel, focusing on relinquishing imbalance among the warps and achieving coalesced access in the data. While using a complex logic to compute each tile, the actual arithmetic operations involved in each tile are quite simple, making them ideal for SIMD/parallel execution; and that is what makes CSR5 so efficient in all architectures.

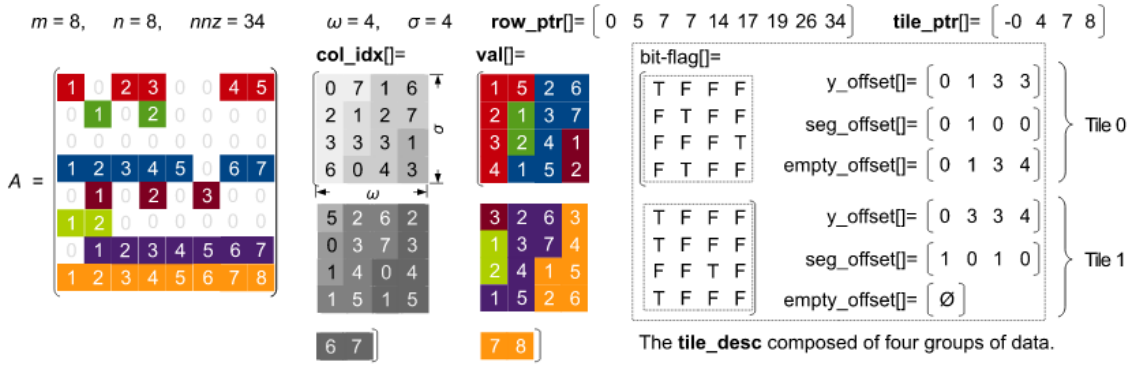


Figure 1.19: The CSR5 storage format of a sparse matrix A of size 8X8. The five groups of information include rowptr, tileptr, colidx, val and tiledesc [Liu and Vinter, 2015]

Light-SpMV

The next CSR format we chose for our network is the Light-SpMV format; which is more actually a modified algorithm using standard CSR, so its not exactly a "new format". It was chosen for its CSR-Adaptive like logic, and its very good performance in our training dataset (Multiple formats were tried). The whole focus of Light-SpMV is eliminating warp imbalance by dynamically allocating rows to warps. It features two different approaches, Light-SpMV warp and Light-SpMV vector. Both are based on a global row management (GRM) data structure, which stores the rows that need to be computed. Each vector/warp (depending on which of the two algorithms is used) is assigned a row from the GRM, computes it and then "asks" for another. This way, the imbalance is completely eliminated. To communicate with the GRM each vector/warp uses atomic operations [Liu and Schmidt, 2015].

The x vector resides in GPU texture memory, in order to speedup reads. The Light-SpMV great performance and utility comes from the fact that it requires nearly zero pre-processing; it utilizes the basic CSR format and focuses **only** in eliminating imbalance. That makes it ideal for unbalanced arrays, where large rows are present; but slower on matrices that memory accessing is also very important. The big percentage of the first kind of matrices puts it among the state-of-the-art implementations for GPU SpMV; and its simplicity determines it easier to classify.

Merge-SpMV

Merge-SpMV is another CSR algorithm for SpMV introducing a strictly balanced method for the parallel computation of sparse matrix-vector products. The algorithm operates directly upon the Compressed Sparse Row (CSR) sparse matrix format without reprocessing, inspection, reformatting, or supplemental encoding. Regardless of nonzero structure, its equitable 2D merge-based decomposition tightly bounds the workload assigned to each processing element. Furthermore, it is suitable for recursively partitioning CSR datasets themselves into multi-scale, distributed, NUMA, and GPU environments that are constrained by fixed-size local memories [Merrill and Garland, 2016].

Its focused on preventing workload imbalance (like Light-SpMV, but not dynamically), by "merging" the small nonzero rows and splitting the long ones. Each thread/warp is assigned a pre-constructed path of execution, and is invoked in parallel. It computes its asigned rows (and/or parts of large rows) independently and returns the results. This way, work is distributed among warps equally, eliminating imbalance, without the need for dynamic atomic operations like Light-SpMV.

ViennaCL

The last library we chose for our benchmarks, is the free open-source GPU-accelerated linear algebra and solver library, **ViennaCL**. ViennaCL offers implementations of the most popular SpMV formats, but we chose it mostly for the Sliced-ELL algorithm (we benchmark all the formats, but other formats are slower than their cuSPARSE/CSR5 equivalents). Sliced-ELL is an improvement of the SELL-C format, a format which focuses on reducing the zeroes that are included in the AS matrix because of imbalance in nonzero distribution in rows.The purpose of Sliced-ELL is to take advantages of the ELL format and reduce the padding overhead of the naive ELL, resulting in a more "safe" format regarding performance (much like HYB, but with a different approach).

The final Sliced-ELL (or SELL-P) kernel works as following. The kernel assigns t threads to each row in one slice with block size b. For this purpose it is necessary to convert the SELL-C/SELL-C-σ format into SELL-P by zero-padding the rows to a length divisible by t. While the number of rows in one slice as well as t may be adapted to the respective matrix, the kernel ensures that the total number of threads assigned to one block (i.e., $t \times b$) is suitable for the GPU architecture used, and enables the usage of shared memory. To allow coalescent memory access, the threads are arranged in a $b \times t$ 2D thread grid. For each slice, the kernel computes the number max of necessary multiply-add that each thread has to compute, and the threads proceed over the data. To improve the performance this loop is unrolled into chunks of two. Once all data is processed, the partial products are written into shared memory, and a fan-in algorithm with an increment of the thread count computes the sum for each row in shared memory. Accounting for the parameters α and β in the SpMV operation y =α·Ax+βy, the result is written back into the global memory. The grid necessary to launch the thread blocks has to cover the complete matrix, i.e., the number of blocks is equal to s, the number of slices the matrix is blocked into [Anzt et al., 2014].

Chapter 2

Pairing SpMV and Deep Learning

2.1 The importance of Sparse Matrix Vector Multiplication

Matrix vector multiplication is one of the most important parts of most modern HPC computations, with applications in the fields of science and economy, graph theory, discrete methods, linear algebra and a huge other variety of applications that use huge matrices for data representation. In addition, in most such cases the input matrix is sparse.

As explained previously, Sparse Matrix Vector multiplication (in which we will refer as SpMV from now on) is a part of most computational methods of linear algebra. That makes it a part of many scientific applications that utilize linear algebra; like iterative methods for solving huge linear systems and finding eigenvectors, data mining and graph representation. In addition, in such applications the SpMV kernel is usually the most computational-heavy part. This makes the optimization of the SpMV kernel a very important scientific research target, since with its speedup the performance of a lot huge scientific applications would improve greatly. This is why it has been categorized as one of the "seven dwarfs", placing it among the seven most important processes for the scientific field for the next century [Asanović et al., 2006].

Fortunately, the matrix vector multiplication can be very easily parallelised among multiple execution threads. Since there are no dependencies in its computational kernel between separate elements of the input matrix as we explained earlier, the calculation of the output vector y can be done in parallel; but the algorithm still remains memory bound. That is caused by the fact that the number of memory accesses per iteration is equivalent to the computation required in each step, for both dense and sparse matrices. In the case of the DMV, this can be partially "hidden" with the utilization of the cache hierarchy, vectorization and data prefetching.

Of course, the use of DMV for SpMV is totally out of the question, since by default DMV algorithms have $\Theta(n \cdot m)$ complexity, while SpMV has $\Theta(nnz)$. On the other side, in the case of SpMV all of the above optimization techniques become much harder to utilize. This is caused because, in order to effectively store sparse matrices without storing useless zeros, the alternative storage formats we analyzed before are used to store the matrix using much smaller vectors or complex structs. While without these formats SpMV would converge to a DMV, which would lead to much heavier computations with most of the elements being zero, the alternative SpMV formats create new problems; any possible spacial locality of the input matrix is eliminated and the kernel

suffers from workload imbalance.

The first step focusing on SpMV optimization, was implementing the multiple storage formats which exist today for sparse matrix representation. The number of formats and algorithms that exist today for SpMV is huge; each one trying a different optimization approach, trying to eliminate or reduce the problems noted above. Some promise average to good performance for all input matrices, while others try to focus on fully optimizing computation for specific matrix patterns and structures. In Fig. 2.1 we can see the problem more clearly. The red line represents the performance "roof", and the green lines the operational intensity. Their intersection point is the theoretical peak performance of each of these 46 divergent arrays. We can notice that the small operational intensity (computations/memory operations) of the SpMV kernel reveals its memory bound nature, and the small utilization of the GPU.

Despite all this research in the field, the question "which method is ideal for this matrix?" cannot be explicitly answered for most matrices. The whole purpose of this thesis is the study and research of possible ways this question can be answered. Instead of implementing one more method for SpMV, our focus is to construct a tool which can classify the best method for any input sparse matrix, without consuming too much pre-processing time for the classification.

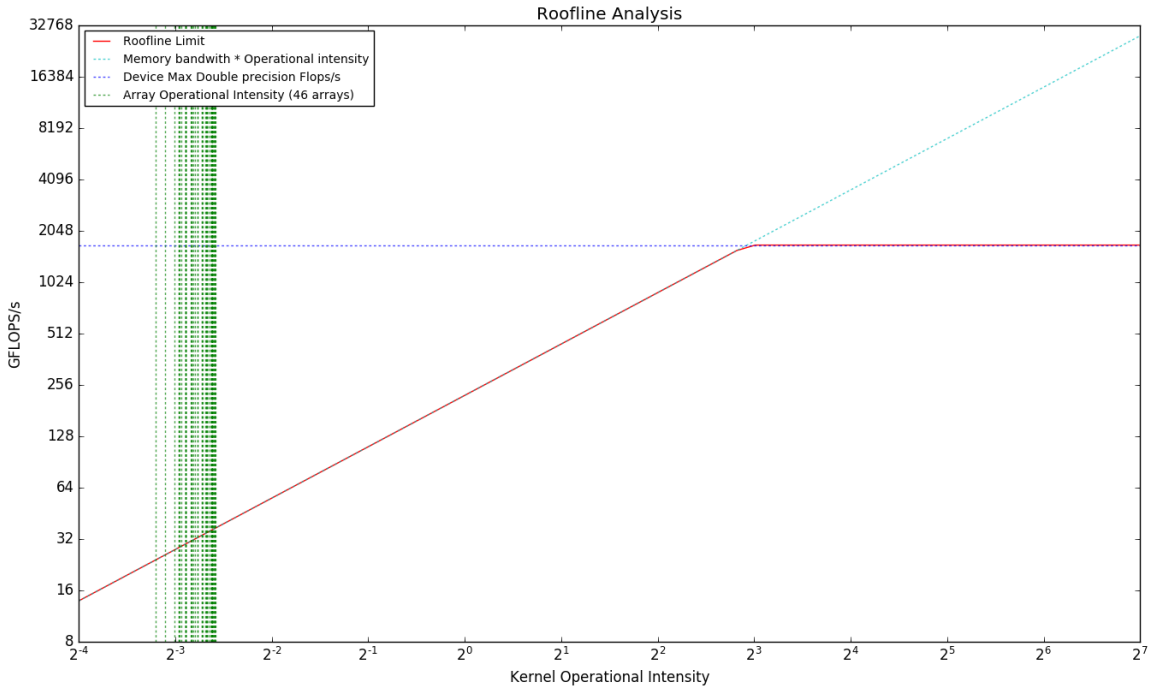


Figure 2.1: An analysis of the performance of 46 divergent sparse matrices in the GPU as per the Roofline model [Williams et al., 2009].

2.2 Previous work in SpMV format selection

The idea of format selection for SpMV is a relatively new concept. The motivation behind it is that by selecting the best format for a given SpMV execution, many applications-and especially scientific and "big data" applications where the size of the input matrices has exceeded "normal" sizes- can gain considerable speedups. It is therefore very important to increase the utilization of the given resources, by ensuring that a suitable algorithm is used in each case. Approaching the exascale era, we have to ensure that not only the hardware of HPC systems is taken into account; but the algorithms that will run on them as well. This is the reason why SpMV format selection for problems of such scale is not only of research interest; but a necessity as well.

2.2.1 Dynamic format selection methods

The first SpMV classification attempt was implemented in 2013, in the form of a Sparse Matrix-vector multiplication Auto-Tuning system (SMAT). While all research before had focused on either optimizing SpMV for a given application or target architecture, SMATs purpose was to create a programming interface which would adapt to the input matrix both structure and architecture wise. In other words, SMAT searched for the optimal storage format based on sparse matrix structures at the algorithm level while also generating the optimal implementation based on the processor architecture [Li et al., 2013]. SMAT included a pre-trained model on a dataset of 2000 matrices.

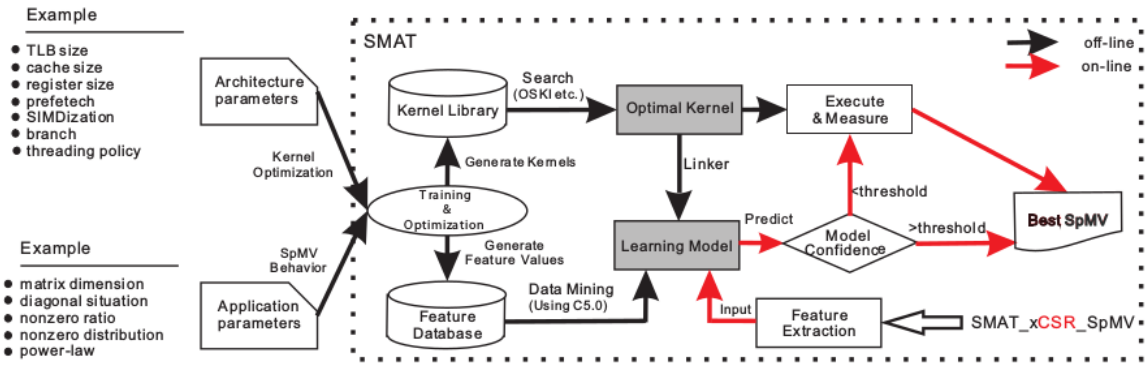


Figure 2.2: The SMAT architecture. *source: [Li et al., 2013]*

One year later, the Adaptive CSR (ACSR) format for SpMV was released; proposing a "dynamically adaptive" SpMV format. ACSR used certain characteristics of the input matrix, like max row length and row length divergence, to choose between two different algorithms (CSR-vector or CSR-Stream) for the kernel execution [Greathouse and Daga, 2014]. The next year, its creators released an improved GPU specific version named "Structural Agnostic SpMV" based on the Adaptive-CSR algorithm (A-CSR); it featured some improvements over the standard A-CSR, and also added a third algorithm for irregular matrices with very long rows [Daga and Greathouse, 2015].

At the same year with Structural Agnostic SpMV, the first generic SpMV format classification implementation was released; triggering the next stage of SpMV format prediction, decision trees.

While research until then was focused on creating an algorithm which could adapt dynamically based on the specific input and architecture, the focus of this work shifted in implementing an independent tool which could predict the best format for any given matrix using machine learning.

In that approach, three decision metrics were used; the fraction of non-zeros, the average number of non-zeros per row and the standard deviation of number of non-zeros per row [Sedaghati et al., 2015]. The difference from previous works (which also used almost the same metrics) was that these numbers were directly fed to a simple decision tree, which subsequently predicted an SpMV format. While a pretty simple classification approach, it was the milestone for most future machine learning SpMV prediction models.

In 2017 a new work featuring decision tree classification was published; but like its older precursors it focused on using the classification result not for format prediction, but for algorithm auto-tuning. It augmented standard CSR to adapt its kernel parameters based on the outcome of the machine learning decision tree classification seen in Fig. 2.3; in order to choose the ideal kernel and bin sizes for the given matrix [Hou et al., 2017].

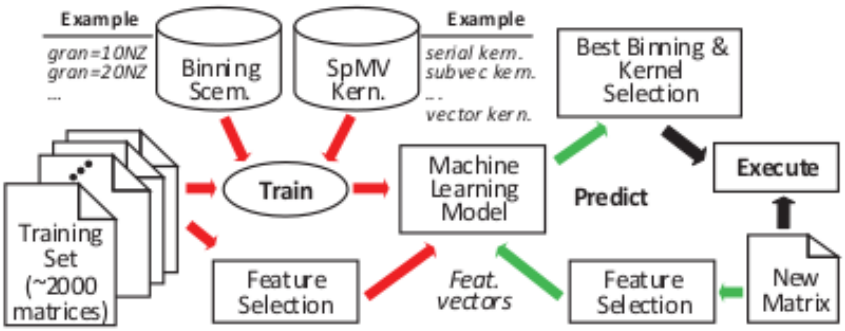


Figure 2.3: The machine learning model for kernel and bin selection. *source: [Hou et al., 2017]*

A month later the next step in SpMV format prediction was published featuring an SpMV optimizer, which focused on recognizing the possible performance bottlenecks for the input matrix and constructing the ideal optimization scheme in order to surpass them. This scheme was based on the standard CSR algorithm, and the optimizer overhead was designed to be very small, in order to make it overall ideal for a wide variety of applications. The bottleneck classification was made either through profiling or by matrix property inspection.

The possible bottlenecks for any given matrix were categorized in four classes; Memory bandwidth bound, memory latency bound, imbalanced and computational bound. Then, for each of these classes, an upper performance bound was calculated after each one's elimination, in order to find which ones degrade performance the most for each matrix. The proposed profile-guided classifier used these upper bounds to calculate the class(es) for each matrix [Elafrou et al., 2017]. The feature-guided classifier calculated 14 different features for the input matrix, and used a decision tree to predict the possible bottlenecks.

Finally, deep learning for the classification of SpMV formats is an even more recent idea. When we started this thesis there was no work whatsoever; but during the last year a very important paper was released. In this paper the first step towards " Bridging the gap between Deep Learning and SpMV format selection" was made [Zhao et al., 2018]. Its purpose was to try a CNN (Convolutional Neural Network) approach to SpMV format selection. Its impact on our work was huge; all

the steps, reflections and problems we met or came against during our approach are also seen in there, making it the most important basis for our work, and a milestone for SpMV format selection in general.

In this work, three different methods were used for input representation; a binary , a density and a histogram representation. The binary array is the most "naive" representation, but constitutes the first step to understanding input difficulties of the problem. The binary representation maps the non-zeros of the input array to an other much smaller array which the CNN will see as an image. The density representation is quite similar to the binary one; but instead of binary values a density value ranging from 0 to 1 is used for each pixel. The last and more intuitive representation creates a matrix containing a histogram of the distances of the nonzeros from the diagonal, since its the most important information for classification in most matrices.

For training, a traditional CNN was used, but an alternative late-merging structure CNN was also implemented, in order to extract information correctly from different features; for example binary plus density. Then, to eliminate the cross-architecture challenge of SpMV performance, a transfer learning option was included, featuring both continuous and top evolvement. The final results exceeded all previous state-of-the-art classification attempts in terms of plain accuracy, and included predictions for both CPU and GPU.

2.3 Neural Networks

We have already mentioned the concept of "CNNs" and "Neural Networks" without explaining them explicitly; and in our approach we will use CNNs for SpMV format selection. For this reason, in this section we will give a brief introduction to neural networks, how they work and what problems they were meant to solve; and explain further the general CNN structure. A Neural Network is a computational model that is inspired by the way biological neural networks in the human brain process information. Neural networks are widely used in machine learning and classification applications and have lead to many breakthrough results in speech recognition, computer vision and text processing. Below we will explain the basis on which they operate and their generic architecture.

The basic unit of computation in a neural network is the neuron, often called a node or unit. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated weight w, which is assigned on the basis of its relative importance to other inputs. Additionally, there is another input with weight b and constant value 1 (called the Bias), representing the constant part of the input. The node applies its **activation** function f to the weighted sum of its inputs, in order to produce the neuron output. Feedforward Neural networks (which are the simplest neural networks) usually consist of three types of nodes:

- **Input Nodes** – The Input nodes provide information from the outside world to the network and are together referred to as the "Input Layer". No computation is performed in any of the Input nodes – they just pass on the information to the hidden nodes.
- **Hidden Nodes** – The Hidden nodes have no direct connection with the outside world (hence the name "hidden"). They perform computations and transfer information from the input nodes to the output nodes. A collection of hidden nodes forms a "Hidden Layer". While a feedforward network will only have a single input layer and a single output layer, it can have zero or multiple Hidden Layers.
- **Output Nodes** – The Output nodes are collectively referred to as the "Output Layer" and are responsible for computations and transferring information from the network to the outside world.

While there are neural networks without hidden layers (called Single Layer Perceptrons), they are of no interest in this thesis; thus we will focus on Multi Layer Perceptron networks for machine learning. The process by which a Multi Layer Perceptron learns is called the Back-propagation algorithm. It is a supervised learning method, during which each neuron's weights are randomly assigned at the beginning; then for each labeled input, the neuron sequence predicts a classification label. Every time the predicted label is wrong, the error is "propagated" back to the previous layer, and the neuron weights are adjusted accordingly.

2.3.1 Convolutional Neural Networks (CNNs / ConvNets)

In this thesis, the three networks that we used belonged to a specific category of Neural Networks: Convolutional Neural Networks (CNNs). Convolutional Neural Networks are very similar to ordinary Neural Networks; they are made up of neurons that have learn-able weights and biases, they follow the same execution scheme and they also have a loss function. Their only difference is that Convolutional Neural Networks take advantage of the fact that the input consists of images in order to constrain the architecture in a more sensible way.

In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. This way, the neuron layout resembles the input pictures; which are of size $res0 \times res1 \times colour_{dim}$ ($256 \times 256 \times 3$ for our RGB input images). Unlike other neural networks, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. This allows the network to also recognize certain spacial traits of the input picture; which is critical for picture recognition. Like simple neural networks, a ConvNet is a sequence of layers, where every layer transforms one volume of activations to another through a differentiable function. ConvNet architectures are build using some extra layers along with the usual ones. A bacic CNN has the following layers:

- **Input Layer** – The pictures fed into the CNN, in a three dimensional layout. In most modern CNNs a BGR colour representation is used.
- **Convolution Layer(s)** - These are the basis of CNNs; they are used to "map" a small area of the input picture by computing dot products based on its neuron weights. By mapping pieces of the input instead of single elements, these layers "learn" to activate for certain spacial characteristics of their input.
- **Rectified Linear Unit Layer(s)** – RELU layers purpose is to apply an element-wise activation function (for example a max function), in order to extract certain characteristics from the previous layer. The RELU functions are static and not learn-able.
- **Pooling Layer(s)** – These layers perform a down-sampling operation along the spatial dimensions (width, height) of the input, in order to focus on specific areas or decrease local size.
- **Fully Connected Layer** – The layer that computes the class scores, taking into consideration **all** neurons of the previous layer.

Figure 2.4: A regular 3-layer Neural Network.

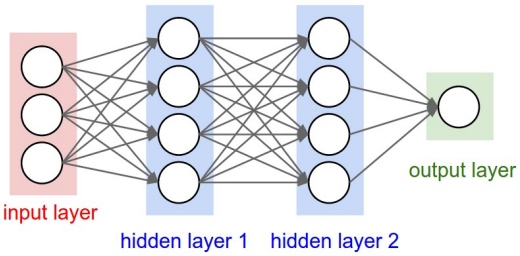
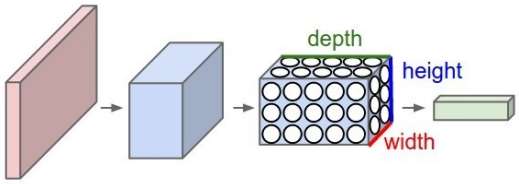


Figure 2.5: A CNN network.



source: Stanford CS231n

2.4 Our approach

While in the previous parts we explained the motivation and the basis behind our work, here we will make the targets of this thesis clear and explain the steps of our approach. As we explained, while SpMV format selection is a relatively new concept, there is a considerable amount of research on the subject; and this research also influenced our approach and targets. Our work regarding SpMV format selection consisted of three basic steps/challenges. These were:

- Creating a representative dataset for all existing sparse matrices and benchmarking it on the state-of-the-art SpMV formats
- Exploring and evaluating the options regarding the choices of input for the Neural Network
- Choosing the right Neural Network and tuning it to fit our needs

2.4.1 Target Architecture: GPU

Before we expand these three, the first thing we have to explain is our targeted architecture choice. While SpMV is part of both CPU and GPU applications, we chose in this thesis to focus on the GPU. This choice was made because in the last years, graphic processors/vector machines have gained increased popularity and are now commonly used in supercomputers, including the Top500. Since GPUs can offer great memory bandwidth and vectorization, and both are important for huge matrices and problems, we found it more appealing.

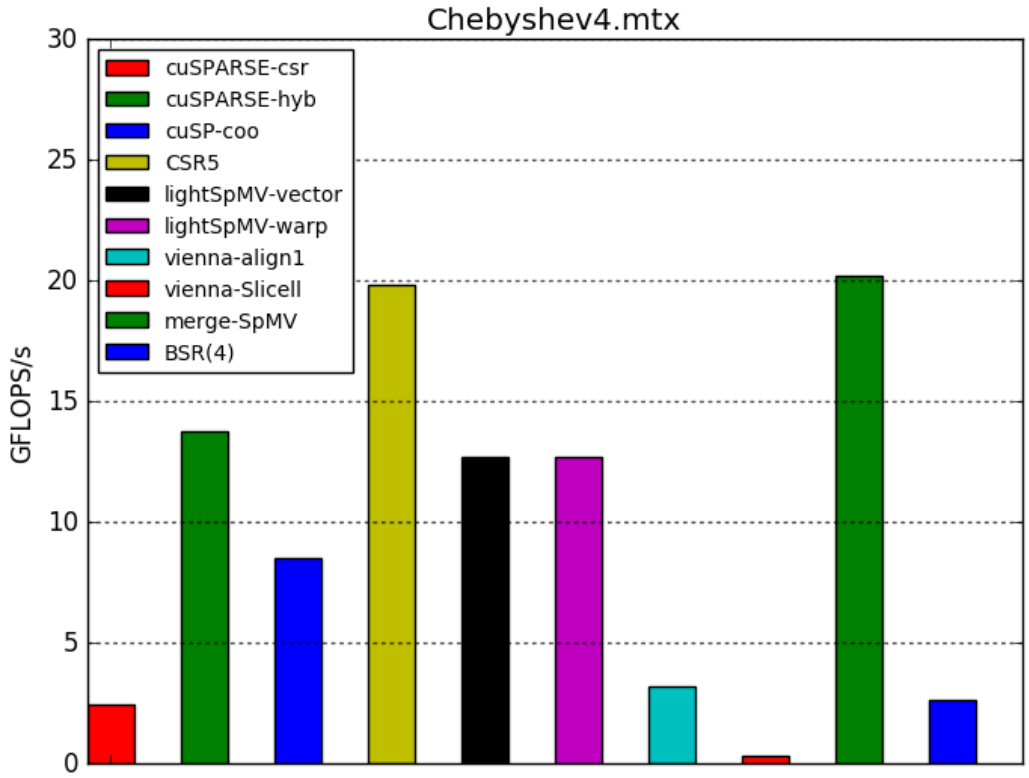


Figure 2.6: GPU benchmark results for Chebyshev.mtx. *source: SuiteSparse*

Also, another important factor was that format prediction is even more important in GPUs, since the use of the "wrong" format can result in even more drastic performance loss than CPUs. We can see in fig. 2.6 the huge difference between different implementations for a given matrix; in this case vienna-Slicell is **65** times slower than CSR5 or merge-SpMV, and the widely used cuSPARSE-csr is still **8** times slower. Also, since technically our work was centered more on the dataset and the input representation than the used Neural Networks architecture, our results could be used for CPUs too, by benchmarking our dataset in CPUs and retraining the network for the new set. These were the factors why we chose to benchmark, train and evaluate performance in a GPU. In the following section, we will give a brief explanation of our work-flow, the problems we faced and the adjustments we chose to make.

2.4.2 Training/test set

Creating a representative dataset might seem introductory for the task and of no interest, but was undeniably the biggest challenge we faced; and also unfortunately set the bounds for this thesis. While decision trees or other classifiers used until now require a solid but relatively small dataset, deep learning is a whole different thing. To train our networks, we required a huge dataset of matrices, which should be equally representative of all matrices which exist in scientific applications.

Unfortunately, this was impossible, since the Florida SuiteSparse Matrix Collection (which is the commonly used basis for SpMV research containing Sparse matrices from all scientific fields and problems) has a limited amount of matrices. While by using the whole suite with a few basic transformations as it has been proposed in previous works [Zhao et al., 2018], we could create a relatively big dataset, it would not serve our purpose. The reason is that, in order to utilize modern GPUs (which feature a huge number of threads/warps), a minimum size for the input matrix is required; and nearly 80 % of the SuiteSparse Collection matrices did not fit that criterion.

In addition, while the other 20 % contained a good (but still very small) starting basis for a representative dataset, the exact opposite problem appeared as well; some matrices were simply too big for our cluster and processing power. This did not only complicate our dataset creation, but also limited our results; since for example as we will explain in the next chapter the neural networks trained on a 10-100 MB range dataset were unable to predict the correct format for input matrices of > 500 MB, since the input images did not contain the required info for the network to recognise the difference.

That left us with a small starting dataset of 500 matrices. In order to be able to train networks requiring inputs of at least 10000 elements to converge, we had to create synthetic matrices, and also generate matrices derived from the 500 we had, by using mirroring, duplicating and some other techniques we will properly explain on the next chapter. This gave us a starting dataset to experiment on, but greatly reduced its "quality", resulting in over-fitting and spoiled predictions on the (not artificial) validation set.

2.4.3 Input representation

Our thesis original purpose was to train a deep network using an **image representation** of the input matrix. When the first research paper featuring CNN for SpMV classification was published [Zhao et al., 2018] , our focus changed in order to include and evaluate this new element as well. In

the beginning we also tried a basic binary representation for the input matrix, but later decided to focus towards making something more complex. The final form of our input was somehow similar to the density representation proposed in the above work.

We mapped each matrix to a 256×256 RGB image, where each pixel represented an equal block of the input matrix, and in the RGB channels of each pixel we mapped the density of the given block. That gave us a density resolution of 256^3, which is enough to accurately map even the largest matrices of the SparceSuite without information loss. While we explored other options; for example using only 2 channels for density and another for distance from the diagonal (similarly to the histogram representation [Zhao et al., 2018]), in the end we chose to implement only the RGB density implementation since the size of our dataset was already prohibitive, and no better prediction rate could be achieved without also improving the "quality" of our dataset (which was impossible with the given resources).

2.4.4 Neural networks used for classification

The last choice we had to make was regarding the Convolutional Neural Network (CNN) we would use. The last 20 years, a lot of CNNs have made their appearance in the scientific community; each featuring its own different depths, layers and techniques. Nevertheless, the main idea and the core architecture behind them remains the same. In Fig 2.7 we can see the top-1 accuracy and the number of operations required for training for the top modern CNNs. For our approach, we chose three implementations (among which only the 2 last were deep); Lenet-5, Alexnet and Googlenet.

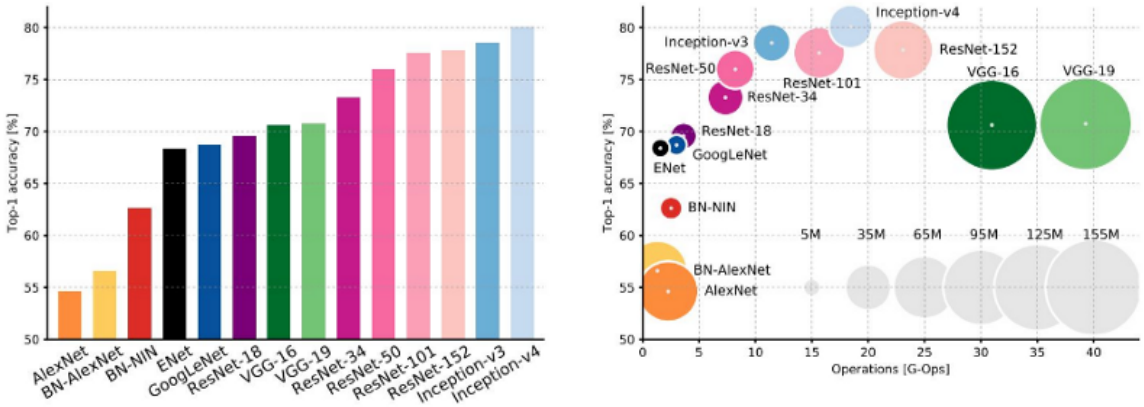


Figure 2.7: An analysis of Deep Neural Network Models. *source: Siddharth Das, Medium*

LeNet-5

Our first choice, and the network we ran most of our experiments in, was LeNet-5, the most famous simple CNN. The LeNet-5 model was first implemented to recognize digits in the form of $32 \times 32 \times 1$ gray-scale images, and was used mostly by banks in order to recognize hand-written numbers. As shown in Fig. 2.8, it consists of one input layer, two pooling and two convolutional layers , 2 fully connected layers (which might change depending on variations of the network) and of course a Gaussian output layer. While its depth and accuracy are overshadowed by modern CNNs, it is very fast compared to them, making it ideal for fast dataset quality testing, input shape evaluation and providing an accuracy baseline.

In our approach we will use a somehow modified LeNet, which takes RGB input ($256 \times 256 \times 3$). While that requires a lot more processing power and memory, it is essential for our problem; since training with smaller images (64×64 , 128×128) resulted in huge accuracy loss. Despite modifications, LeNet still was a lot faster than AlexNet or GoogleNet.

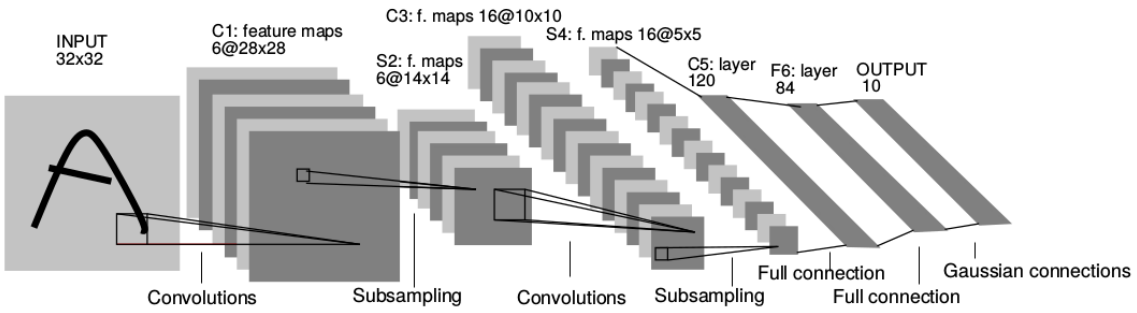


Figure 2.8: The architecture of LeNet-5 for a 32×32 input gray-scale image. *source: [Lecun et al., 1998]*

AlexNet

AlexNet was the first high-impact CNN after Lenet-5, outperforming all its previous competitors in image classification. The network had a very similar architecture as LeNet but was much deeper, with more filters per layer, stacked convolutional layers and designed to run in multiple GPUs. The net contains eight layers with weights; the first five are convolutional and the remaining three are fully-connected. The output of the last fully-connected layer is fed to a 1000-way softmax which produces distribution over the 1000 class labels of ImageNet.

AlexNet maximizes the multinomial logistic regression objective, which is equivalent to maximizing the average across training cases of the log-probability of the correct label under the prediction distribution. The kernels of the second, fourth, and fifth convolutional layers are connected only to those kernel maps in the previous layer which reside on the same GPU (if multiple are used). The kernels of the third convolutional layer are connected to all kernel maps in the second layer. The neurons in the fully-connected layers are connected to all neurons in the previous layer. Response-normalization layers follow the first and second convolutional layers. Max-pooling layers follow both response-normalization layers as well as the fifth convolutional layer. The ReLU non-linearity is applied to the output of every convolutional and fully-connected layer.

The first convolutional layer filters the $224 \times 224 \times 3$ input image with 96 kernels of size $11 \times 11 \times 3$ with a stride of 4 pixels (this is the distance between the receptive field centers of neighboring neurons in a kernel map). The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size $5 \times 5 \times 48$. The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size $3 \times 3 \times 256$ connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$, and the fifth convolutional layer has 256 kernels of size $3 \times 3 \times 192$. The fully-connected layers have 4096 neurons each [Krizhevsky et al., 2012].

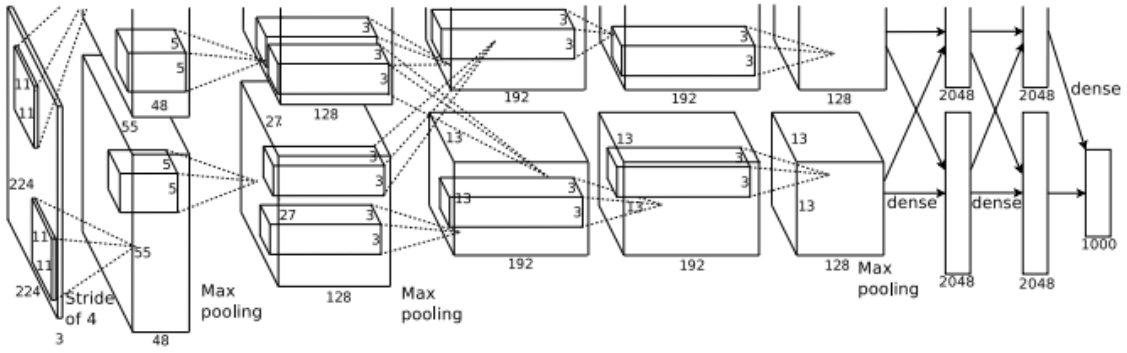


Figure 2.9: The architecture of AlexNet for a 224×224 input RGB image. *source: [Krizhevsky et al., 2012]*

In our approach, we used a somewhat modified version of the original AlexNet, called "CaffeNet". This version was implemented for the Caffe framework, which we used in our training. Its deviation from the original AlexNet is that it is not training with the relighting data-augmentation and that the order of pooling and normalization layers is switched (in CaffeNet, pooling is done before normalization). We deployed CaffeNet both with its default $224 \times 224 \times 3$ image size (by cropping our dataset) and with our $256 \times 256 \times 3$ actual images. Accuracy and training time was not influenced significantly by this small change; so in the end we used our full images in order to make comparison with the other two networks easier. The 1000 classes of the last SoftMax layer were changed to 6 to match our classification problem.

GoogleNet

GoogleNet, the most recent CNN we chose for our approach, was released in 2014. It achieved a revolutionary top-5 error of 6.67 %, putting it extremely near human error (5.1 %). Its better trait though was performance, not accuracy; while featuring 22 layers (compared to AlexNet's 8), it managed to offer a much faster training speed (paired with the improved accuracy). That was achieved by using an "Inception Layer" logic.

The idea behind the inception layer was to be able to extract both specific and more wide information from each area of the input/previous layers. So, the called "inception modules" were added between layers in order to create convolutions of 1×1, 3×3 , 5×5 paired with a 3×3 max pooling and concatenated before the next layer [Szegedy et al., 2014] . This way, the network can handle multiple scales for the same objects in the same layer, and learn separately each one of them. Instead of the normal one SoftMax output layer, GoogleNet features three different SoftMax layers, the two first in the network and the last one in its end. This way, even for problems where the big number of layers might cause over-fitting in the dataset after a number of layers, results are calculated by taking into account all three SoftMax layers; hence allowing the network to train and test much more efficiently.

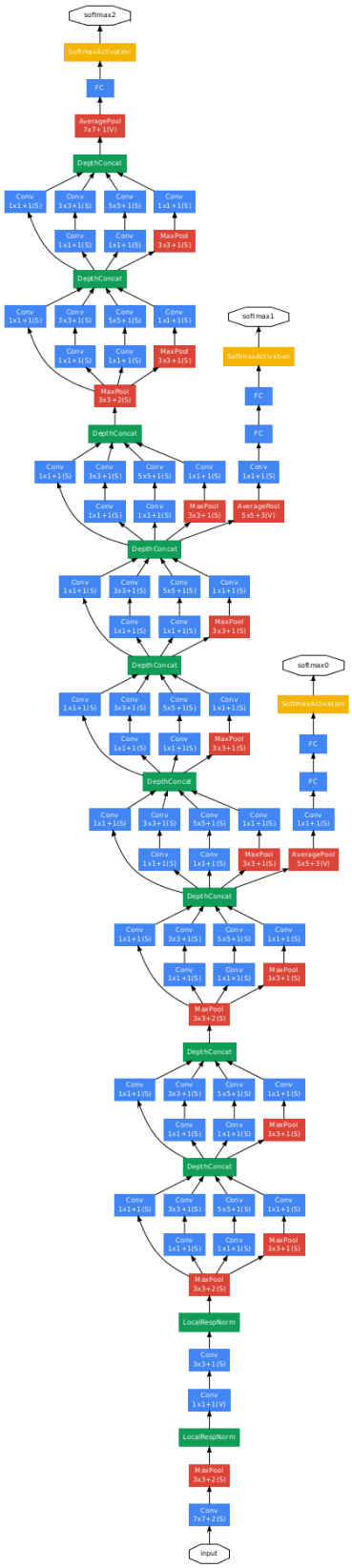


Figure 2.10: The GoogleNet Architecture. *source: [Szegedy et al., 2014]*

Chapter 3

Dataset generation

The first and most important part required to train our network was the training set. Modern CNNs require a considerable training set size in order to produce solid results; as we mentioned in the previous chapter, unfortunately the first bounds regarding our experimentation were set here. Speaking in numbers, we will explain the main problem we faced in our dataset generation. All functions for this purpose were written in python, and execution was controlled by bash scripts. These scripts were ran in 2 different architectures, utilizing each ones multicore architecture with the use of **screen** for simultaneous execution.

The first one runs the Debian GNU/Linux 7.11 (wheezy) distribution, and features an Intel(R) Core(TM) **i7-4820K** CPU @ 3.70GHz (4 cores, 8 hybrid threads), a **Tesla-K40** GPU, and an Intel Xeon Phi Knights Corner co-processor (which we did not utilize in our thesis).

The second one consisted of 4 similar devices, each featuring two NUMA nodes with Intel(R) Xeon(R) CPU **X5650** @ 2.67GHz (two 6 cores/12 threads in each NUMA node for a total of 12 cores, 24 hybrid threads for each device). They all run the Ubuntu 14.04.2 LTS (trusty) distribution.

3.1 Matrix Generation

In order to create a representative dataset for the problem we wanted to address, we had to include matrices of all sizes of interest; which range from 100 MB to many GBs. As we mentioned, after excluding all smaller matrices from the SuiteSparse collection, we were left with a small number of arrays (less than 1000). Among these, around 100 were bigger than 1 GB. Not even LeNet, our simplest CNN was not able to train with such a small dataset. That, led to an extra step in our approach that we had not considered in the beginning; we had to artificially generate a lot of matrices.

Also, as we mentioned before, we also had an upper size limit. Since matrix generation is a quite heavy operation, requiring considerable input, computation and output operations, it takes a lot of time. In order to create a representative dataset for the big matrices as well, we had to include big matrices totalling an at least 20% of the total dataset. That would mean more than 3000 matrices of size > 1 GB, which would already require at least 3 TB of hard drive (in reality for synthetics derived from the SparseSuite, around 10 TB); and this only for the $1/5$ of the total dataset. Based on our first calculations, in order to create "the ideal" dataset for our problem, a total of around 30

TB would be required.

Of course, the biggest problem for such a dataset was not the total size; it was the time required to generate it. To elaborate, generating our actual dataset took around 2 weeks running in our available devices, for a total size of 1.7 TB; so creating the ideal one with our processing power would take many months (and that is without even accounting the time it would require to benchmark the dataset for all formats; which with one GPU would take years).

The most common techniques for machine learning dataset enlargement are mirroring, cropping and random sampling rearrangement. Since the best SpMV format for a given matrix usually depends on its complete structure, automatic random cropping durimg training was out of the question; it would lead to accuracy degradation for non-artificial validation matrices and also create pictures of different resolutions. Also, while random sampling and rearrangement would create new matrices, the danger of over-fitting onto artificial sets would be much greater (since their quantity would be much bigger than that of the original matrices); and the desirable structural characteristics of the actually matrices would be corrupted.This is why most of our dataset generation was based on mirroring techniques. Below we explain in detail the sub-sets of synthetic matrices we generated, and the methods used in each one.

Power-law Cluster Graph Matrices

One of the most common appearances of sparse matrices in scientific problems is in the form of graph adjacency matrices, since they are almost always by default sparse. The SuiteSparse collection contains some, but since their exact structure would be corrupted by using transformation techniques, we chose for this part to generate **500** actual Power-law graphs ourselves. For their generation we used **networkx**, a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. The graphs were generated with the powerlaw_cluster_graph function. In Figures 3.1 and 3.2 we can see the visual binary and density representation respectively for a network of 300000 nodes with 10 edges per node.

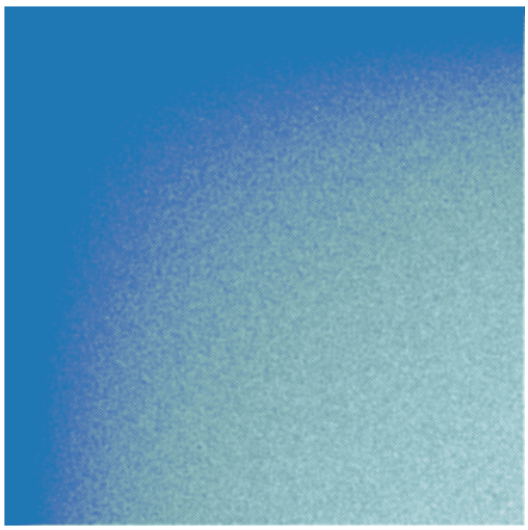


Figure 3.1: Pw_cl_graph_300000_10_4 binary

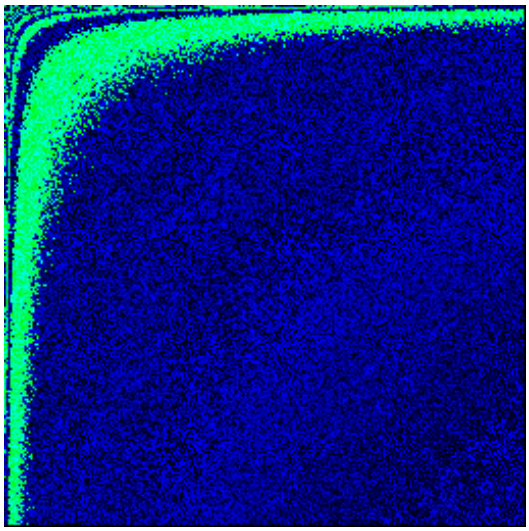


Figure 3.2: Pw_cl_graph_300000_10_4 density

Power-law Sequence Matrices

Since the networkx's powerlaw_cluster_graph function creates sorted adjacency matrices for power-law graphs based on each node's number of edges, we wanted to further enhance our dataset with unsorted power-law graphs (which are also used widely). In that end, instead of randomly scattering the results of powerlaw_cluster_graph; which would lead to the same end but would be computational intensive, we used the networkx powerlaw_sequence function to create a power-law sequence (as the name suggests), paired with the expected_degree_graph, which generates one of the possible graphs derived from this sequence.

This was much faster than powerlaw_cluster_graph , allowing us to create **1500** unsorted power-law matrices for our dataset. The difference in number between matrices generated with each power-law method (we chose to implement $3\times$ more power-law sequence matrices) was not made only for processing time reasons. In most graph problems, the adjacency matrix is unsorted in relation to the central nodes; so the SuiteSparse contains mostly such matrices. In addition, we found out that the neural network had some difficulty classifying these unsorted matrices; making it more important to increase their number.

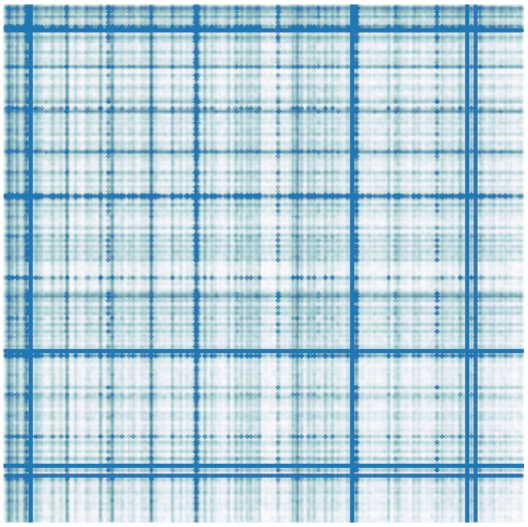


Figure 3.3: Pw_seq_graph_220000_1.8_1 binary

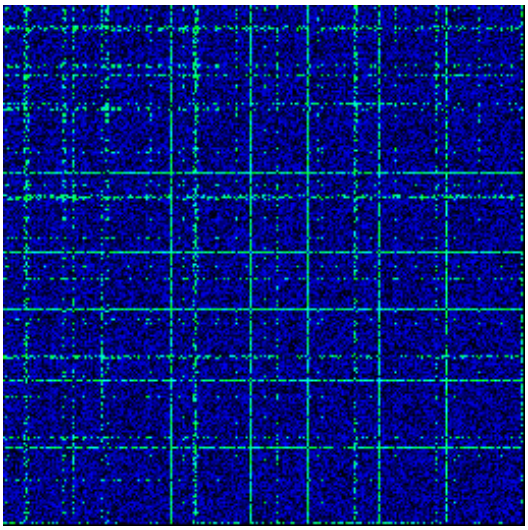


Figure 3.4: Pw_seq_graph_220000_1.8_1 density

Re-sized Suite - Div suite

As we explained, the biggest challenge in creating a considerable big dataset was the small number starting real matrices. For this reason, we had to create synthetic transformations with a high in:out ratio (meaning that many synthetic matrices should be produced from each real one). The sole purpose of this part of the dataset was to increase total dataset size considerable, in order to make it possible to run the AlexNet and GoogleNet CNNs. The foundation of this dataset consisted of 525 matrices from the SparseSuite collection; ranging from 2 to 30 MB. The first step was re-sizing these matrices to fit into the 30-500 MB range they were meant to cover; this was done using the map function in 7.

Algorithm 7: Array block resizing

Input: Matrix A in COO format, resize value res

Output: Matrix B in COO format

```
1  Create new empty COO matrix B;
2  for n = 0; i < A.nnz; n++ do
3      elem = A.data[n] ;
4      for i = 0; i < res; i++ do
5          for j = 0; j < res; j++ do
6              B.row.append(res * A.row[n] + i) ;
7              B.col.append(res * A.col[n] + j) ;
8              B.data.append(elem) ;
9          end
10     end
11 end
12 return B;
```

This function's purpose was to create a new matrix of size $res \times res \times Matrix\_size$, which would consist of $res \times res$ blocks in place of non-zeros. This would lead to both binary and density representations for the said matrix remaining **exactly** the same. While that was the initial purpose for these matrices, it ended up being destructive for the final dataset; since the network learned to train on such matrices, and as we will explain in our evaluations, our two suggested image representations gave zero information about the possible blocks in the matrix. Nevertheless, this dataset allowed us to recognize all the weaknesses of our implementation; and find ways to surpass them.

By passing our 525 matrices trough this transformation, **783** matrices were created in the desirable size range. The next step was to generate many synthetic matrices from these, which was done with a transformation we devised; diagonal distance variation transformation (**DDVT**). We used this transformation based on the fact that the nonzeros of many structured matrices are spread around the diagonal; and for some scientific problems the variation between different matrices is the distance of a specific shaped structure from the diagonal.

Following this main idea, we constructed a function that takes any input matrix and "bends" it closer to the diagonal. This bending is done either upwards or downwards; creating either a matrix whose elements have somewhat "moved away" from the diagonal, or in the opposite case, closed in. As we mentioned, in specific scientific problems, such matrices are very common; for example in 3D reconstruction changing some variables in the generation equations lead to such matrices; which was also the way we came up with this approach.

Using this function, we produced **7830** new matrices derived from the resized ones. While as we mentioned before, resizing corrupted the dataset, the diagonal distance variation transformation as a separate function is quite useful; given a normal starting dataset it can produce up to 10x matrices (or more, for divz > 5, which for some matrices might end up destroying the matrix structure completely), which will keep the most important structural traits of the input (for most matrices). While we didn't use it on our other sub-sets because of processing power limitation, it could help in the future in the creation of a much bigger dataset given the resources.

Algorithm 8: Diagonal Distance Variation Transformation

Input: Matrix A in COO format, transform value $divz$

Output: Matrix B in COO format

```
Copy COO matrix A to matrix B;
for n = 0; n < A.nnz − 1; n++ do
    i = B.row[n] ;
    j = B.col[n] ;
    if i > j: ;
    if i > j then
        zrow = i + (i-j)/divz ;
        if zrow < 0 then
            B.row[n] = 0 ;
        else if zrow < B.shape[0] then
            B.row[n] = zrow ;
        else
            B.row[n] = B.shape[0] - 1 ;
    else
        zcol = j + (j-i)/divz ;
        if zcol < 0 then
            B.col[n] = 0 ;
        else if zcol < B.shape[1] then
            B.col[n] = zcol ;
        else
            B.col[n] = B.shape[0] - 1 ;
end
return B;
```

The result of the diagonal distance variation transformation can be seen in Fig. 3.5, 3.6 and 3.7 respectively. The basic heart2 matrix is transformed on 10 different matrices (the 2 shown here are the ones for divz 3 and -3 respectively) which share most of its structural traits, but are divergent enough to create new stimuli for the network. The algorithm for this transformation is shown in Alg. 8. For a given matrix A and a transformation parameter divz, this function produces a new matrix, "shifted" divz units with the diagonal as a reference point. This way, while the structural similarity around the diagonal remains the same, a complete new matrix is created; making this method ideal for the enlargement of small datasets, without sacrificing the uniqueness of real-world application matrices.

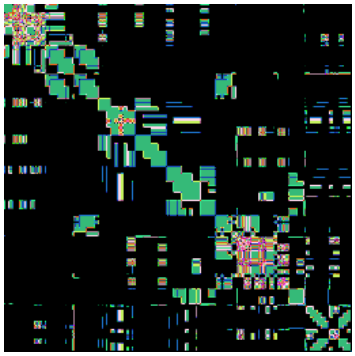


Figure 3.5: Heart2 density

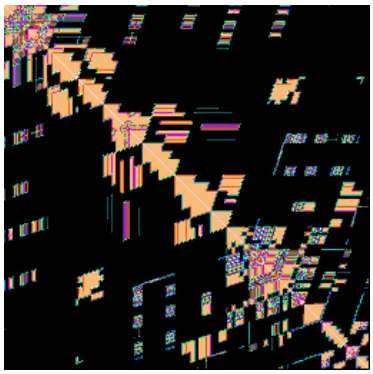


Figure 3.6: Divz_3_heart2 density

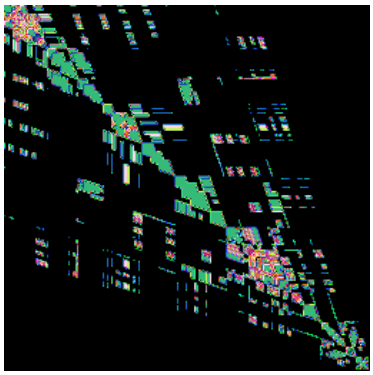


Figure 3.7: Divz_-3_heart2 density

Augmented Block Suite

A considerable percentage of the sparse matrices used in scientific applications; like 2D/3D reconstructions and specific structures, consist of blocks. Since the image representations almost always **completely fail** to pass the block nature of a matrix to the neural network, it is one of the most important bottlenecks of this approach. For this reason, we deemed important to include a good amount of blocked matrices; in order to better understand the limitations of our approach in them.

Our starting blocked matrix set consisted of **236** matrices, taken from SuiteSparse. Unfortunately we could not use the diagonal distance variation transformation for blocked matrices, because it completely corrupts blocks, transforming them into "diagonal blocks" which are not used in real world applications. To this end, we used a mirroring technique; in order to both increase the size of our dataset and not corrupt its block nature (since mirrored blocks remain normal blocks).

The idea behind the transformation algorithm was pretty straightforward; for each $N \times M$ matrix, we computed its mirror matrix, then 16 matrices of size $2 * N \times 2 * M$ were created, by using all possible quartile combinations of the original and the mirror matrix. The result was a final blocked-dataset of **1650** matrices. It is important to understand that usual mirroring, used by default in many neural networks could not be used in our case; since the mirrored matrix in our case might have a different training label. It was thus mandatory to include this matrices in the dataset-level, in order to benchmark them along the original ones.

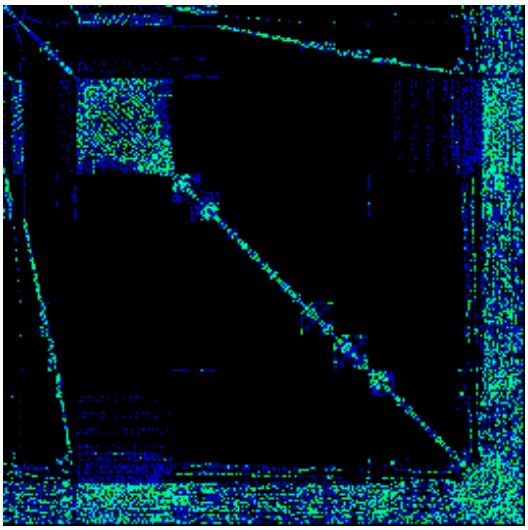


Figure 3.8: An Engine structure matrix

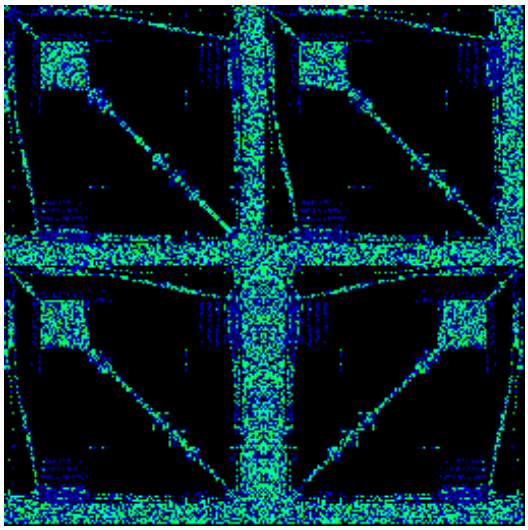


Figure 3.9: The Mirrored-Augmented matrix

Augmented Small Suite

Since as we mentioned, the core of our dataset was the diagonal distance variation transformed resized arrays, which was a relevantly "bad" dataset, training the networks was problematic; in order to surpass this problem, we created a new core for our dataset, using the mirroring we implemented for the block matrices (since it kept the structural integrity of the input matrix intact). This new dataset's purpose was to replace the 7830 matrices of the resized sub-set. It was based on **416** matrices from the SuiteSparse collection; 100 of them being among the large ones used also for the resized dataset (but without resizing this time). After quadruple mirroring, the final dataset had a size of **6656** matrices.

3.2 Picture sampling

The next important part after creating our dataset, was to decide and implement a good visual representation of the matrix; since that would be the only information we would feed the neural network with. As we explained earlier, we used two representations as an input; the naive binary implementation and a augmented density representation. For the naive binary implementation we used a tool provided by python, the **spy** function, which visualizes the nonzero distribution of a given sparse matrix.

For the density representation, we implemented an algorithm in order to accurately visualize the desirable information. The pseudo-code for the algorithm can be seen in Alg. 9. The input matrix A is mapped to a 256×256 image; this is done by splitting the input matrix in 256×256 equal parts. Each of these parts we call **datum** will be mapped to an RGB pixel. For each datum, we compute its density in a scale of 0 to 256^3. From this scale, the values of the three RGB channels are computed; with a 256-base system and the three channels as 3 number digits.

Algorithm 9: Density Mapping algorithm

Input: Matrix A in COO format

Output: 256×256 density image for A

Compute datum number and map size for A ;

Create empty $256 \times 256 \times 3$ pixel image array ;

for $n = 0; n < nnz; n{+}{+}$ **do**

    Increase nonzero count for pixel containing A.data[n] ;

end

for $n = 0; n < pixel\_num; n{+}{+}$ **do**

    temp = $den \times (256^3)/(dtx \times dty)$;

    Pixel[n] = [$temp/256^2, (temp \bmod 256^2)/256, (temp \bmod 256^2) \bmod 256)$]

end

Chapter 4

Experimental Evaluation

In this chapter we present our whole experimental walk-trough. We will start with the benchmark results in our total dataset, and then give some extra guidelines regarding benchmark results and explain our choice of the final classes for the CNNs. Finally, we will explain the training procedure and display, compare and evaluate the results.

4.1 Format Benchmarking

After creating our dataset, the next and most computational intensive part was benchmarking it on the formats we would use as labels. As we explained in the previous chapter, in the end we chose some specific implementations in order to better cover all the different techniques used in SpMV for GPUs. But, we chose to benchmark all formats nevertheless, in order to choose the best candidate for each category; and also reproduce and evaluate their performance.

All our benchmarks were run on a Tesla K40 GPU. Tesla K40 is a massive memory-centered GPU, featuring a memory clock of 6GHz GDDR5, 384-bit memory bus width and 12GB of VRAM; paired with 2880 CUDA cores, and 745 MHz clock speed. While the memory size of Tesla K40 was required for training our networks in the GPU; which was around 20 times faster than CPU training (deeming the second one nearly impossible), it was also important for our benchmarks, since we wanted to produce results for a target architecture that is still used widely. Finally, as we mentioned in the background chapter as well, Tesla K40 has a compute capability of 3.5.

In Table 4.1 we present the benchmark results. Each column contains its correspondent subset's total size distribution among the benchmarked formats (rows); and the last column shows the final percentage per benchmarked SpMV format for the total of all sets. For all benchmark inputs we used a modified version of cuSP's matrix market reader, in order to exclude some matrices and put the read data in unified GPU memory. Each SpMV kernel was launched 100 times on the GPU and the performance was counted based on the average execution time. A common timer we implemented was used for all formats and before launching each kernel, a warm-up launch was made in order to take as more precise results as possible.

The green implementations are the ones we used as labels for our neural network; the other ones were deemed too small for actual classification. For cuSP, the superior cuSPARSE versions beat CSR and HYB, the very few COO matrices performed better in vienna-Sliced-ELL, and most

Format	DDVT$_{resized}$	PWL$_{cluster}$	PWL$_{seq.}$	Block$_{aug}$	Mirror$_{aug}$	Perc (%)
cuSPARSE-CSR	320	0	0	460	676	8.11
cuSPARSE-HYB	120	322	0	322	761	8.49
cuSP-COO	0	0	0	0	0	0.00
cuSP-CSR$_{scalar}$	13	0	0	0	0	0.07
cuSP-CSR$_{vector}$	0	0	0	0	0	0.00
cuSP-DIA	88	0	0	9	8	0.59
cuSP-ELL	1	0	0	3	47	0.28
cuSP-HYB	0	0	0	0	18	0.10
bhSPARSE(CSR5)	4179	122	375	200	3159	44.73
lightSpMV$_{vector}$	114	0	0	97	102	1.74
lightSpMV$_{warp}$	380	0	0	279	1083	9.70
vienna-align1	26	0	0	3	0	0.16
vienna-align4	0	0	0	0	0	0.00
vienna-align8	5	0	0	0	0	0.03
vienna-HYB	0	0	0	23	16	0.22
vienna-Sliced-ELL	22	0	0	73	129	1.25
merge-SpMV	1433	51	975	1	320	15.48
cuSPARSE-BSR(3)	1	0	0	0	0	0.01
cuSparse-BSR(4)	1122	0	0	162	336	9.02
Total	**7830**	**495**	**1350**	**1632**	**6655**	**17962**

Table 4.1: Training Dataset Benchmark results for all sub-sets. *See text for details*

DIA matrices were claimed by CSR5. Vienna's CSR versions were not as optimal as merge-SPMV, lightSpMV$_{warp}$ and CSR5, and Sliced-Ell percentage was too small to include (especially since our network input datasets did not necessary included all 17962 matrices). LightSpMV$_{vector}$ was assimulated by merge-SpMV, which performs the same load-balancing without using atomic operations, by performing it prior exectution. All benchmarks were ran for **double precision** values.

After discarding all other implementations, the remaining matrices were also classified in the 6 "green" categories. It is interesting to note that, among all format executions, 2139 implementations were closer than 2.0%, 1053 closer than 1.0%, and 5594 closer than 5.0%. We chose to randomize the best format selection for matrices with less than 2.0 % performance difference; since such a difference is nearly indiscriminate and could be caused even by execution spikes, and such a small difference does not deem a format " better" than the other for a given matrix. To that end, if two or more implementations for a given matrix were closer than 2.0%, a random label among the competitors was used.

The most " close" formats were CSR5 and Merge-SpMV; which is normal since they target almost the same optimizations. Following we had lightSpMV$_{warp}$, which is also to be expected because of its similarities with merge-SpMV. The number of the close formats was relevantly big; which unfortunately means that it was even harder for the neural network to classify them. If we had a bigger dataset, excluding those implementations would help the network distinguish classes easier; but in our case this would reduce the size of the dataset a lot.

4.2 Network Training

In this section we will continue with the training of our networks, explain the procedure and present the results and metrics. All our training was done on 3 different train sets. All datasets contained the $PWL_{cluster}$, $PWL_{seq.}$ and $Block_{aug}$ sub-sets, and the divergence was based on the 4th sub-set. The first dataset (which we will call the DDVT dataset) was the one we initially created and used; it contained the above plus the $DDVT_{resized}$ dataset. As we explained, the DDVT dataset was a relatively "bad" dataset, because of the inability of the image representations to represent its blocked state. This was why we created the second dataset, containing the above plus the $Mirror_{aug}$ dataset,trying to make a more representative dataset for non-synthetic matrices. Finally, the third dataset contained all our sub-sets, and was used in order to see the problem development on a much bigger set; sacrificing a little bit of robustness.

As we will explained bellow, with the exception of GoogleNet, which run satisfactorily on all datasets, Lenet and AlexNet were much more problematic; for some datasets they did not manage to surpass certain bottlenecks, which we will also explain in the corresponding sections. While the results of our training lead to a different network ranking than we expected (Lenet outperformed AlexNet in both accuracy and performance), we will present the three networks with the order we trained and evaluated them. Each one's architecture can be found in the appendix 6.4.1.

4.2.1 Lenet

As we explained in the previous chapter, Lenet is "the first CNN". Although used for image classification, it is not deep; it focuses more on extracting the right information from the input picture with pooling and convolution, rather than searching for complicated relations between image locations. This in our case, was both an advantage and a bottleneck; which we are gonna explain below. Training the Lenet network took around 5 hours for each training session, for a total of half a week for training all of them. The specific training metrics we used can be found in the appendix 6.4.1. Accuracy was calculated using a Monte Carlo cross-validation of 3 iterations; each synthetic set was split in two random parts where 80% was used for training and 20% for testing during each iteration. In table 4.2 we can see the accuracy results of the Lenet training on all 3 datasets.

		Binary Lenet		Density Lenet					
Dataset		DDVT		DDVT		Mirror		DDVT + Mirror	
Sub-set	Size	Ac.	Top 2 Ac.	Ac.	Top 2 Ac.	Ac.	Top 2 Ac.	Ac.	Top 2 Ac.
$Block_{aug}$	1632	0.1	0.1	0.92	0.98	0.92	0.99	0.92	0.99
$DDVT_{resized}$	7830	0.15	0.34	0.93	0.98	0.27	0.52	0.92	0.98
$PWL_{cluster}$	495	0.0	0.11	0.83	0.97	0.85	0.98	0.83	0.95
$PWL_{seq.}$	1350	0.0	0.74	0.83	1.0	0.82	0.99	0.83	0.98
$Mirror_{aug}$	6655	0.05	0.1	0.37	0.59	0.95	1.0	0.95	0.99
$Test_{synthetic}$	2500	0.15	0.30	0.84	0.89	0.91	0.95	0.89	0.92
$Test_{real}$	416	0.1	0.1	0.48	0.65	0.47	0.69	0.47	0.66
$Test_{large}$	208	0.03	0.15	0.26	0.44	0.42	0.62	0.34	0.47

Table 4.2: Lenet test results

The gray rows for each implementation/dataset are the train set rows; meaning the data which was used in the training of the corresponding model. This means that the expected accuracy for

these rows will be much higher; since its the network's **train accuracy**. In Neural networks, a model's accuracy is instead tested on an other set, unknown to the network during the training, the called **test set**. The cyanide rows/elements are the test sets we used for each model, and constitute the network's actual final accuracy.

Binary Lenet

The first model, **Binary Lenet** faced an important problem of our binary dataset; the image size. The binary images were exported from python's **spy**, and had a size of 372×372 (after cropping them from 640×480 to remove the limits). Unfortunately, Caffe had a bug in GPU mode for specific memory layouts; which did not allowed us to run Lenet for these image sizes. Instead, we resized our images to 256×256 in order to be able to run with caffe. Nevertheless, Lenet was implemented to process images of 32×32, making our images quite large compared. While it technically ran, the results showed that it could not successfully train on the network; it kept predicting the same label *cuSPARSE-BSR(4)* for all matrices. This was something we encountered in AlexNet as well, and also in Googlenet when we didn't subtract the pixel mean from the images. Since Lenet featured no such option, it had to run on the normal images and thus "stuck" on always predicting one format label. For this reason, we only dispayed it for reference reasons, and our following evaluation is for **Density Lenet**.

Density Lenet

Unlike the binary image network, the density representation implementations lead to much more promising results. The DDVT dataset, which was our first approach, achieved satisfying training and test accuracy for synthetic matrices, but had a very low test accuracy on the large real matrices of our third test set. As we explained, this was caused by the bad quality of the DDVT dataset; which confused the network with its hidden corrupted-block nature. The same was true for the dataset containing both the Mirror and DDVT sub-sets; since it was also "tricked" by DDVT's nature. On the contrary, the Mirror-based implementation had a much better performance on this set.

		Binary Lenet		Density Lenet					
Dataset		DDVT		DDVT		Mirror		DDVT + Mirror	
Format	Size	Recall	Ac.	Recall	Ac.	Recall	Ac.	Recall	Ac.
cuSPARSE-csr	1571	0.0	0.0	0.61	0.52	0.66	0.74	0.91	0.87
cuSPARSE-hyb	1742	0.0	0.0	0.48	0.58	0.34	0.89	0.95	0.91
bhSPARSE(CSR5)	8485	0.0	0.0	0.82	0.73	0.76	0.66	0.92	0.93
lightSpMV-warp	2184	0.0	0.0	0.56	0.55	0.63	0.76	0.89	0.87
merge-SpMV	2854	0.0	0.0	0.78	0.78	0.79	0.43	0.86	0.87
cuSPARSE-BSR(4)	1750	0.09	1.0	0.69	0.85	0.87	0.36	0.89	0.93

Table 4.3: Lenet Per-format accuracy

Per-Format Accuracy

In table 4.3 we can see the accuracy and recall for each format for the sum of our train and test set. Recall is a metric which shows the percentage of right predictions for each format among all predictions for the given format. Unlike accuracy, it doesn't help evaluate the network performance; it gives useful information about where the network lacks and what are its weaknesses. In our case for example, it helps understand which formats are the most difficult to predict.

Specifically, we can see for example that cuSPARSE-HYB has a very low recall rate; which means that the model predicts incorrectly hybrid for more arrays than the actual. Paired with the much higher accuracy for the same format it means that the network has learned to recognize which arrays need cuSPARSE-HYB; but also mistakes a lot arrays to have a similar structure. This is caused by hybrid's specific nature; it is a format designed to run well -but not perfectly- on most arrays. In our case, it came first on **8.49%** and second in **43.68%** of the dataset.

On the other hand, merge-SpMV and cuSPARSE-BSR(4) have satisfying recall rates, but much lower accuracy. This in our case is caused because the test sets have a lot matrices falling into those categories, which the network is not familiar with and thus predicts incorrectly. But, the high recall shows that it doesn't usually mistake matrices of other formats as these. That shows that a better dataset would require more matrices for these formats, in order to enhance the network's ability to select them.

Train Accuracy

While the train accuracy of a model says nothing about its performance, it can give us useful information about the limits and bottlenecks of a network. We can see in the Lenet's case that the network has a difficulty fitting on the PWL sub-sets. This is to be expected since these dataset contain matrices with difficult structures for the network. Specifically, $PWL_{cluster}$ runs well only on 3 formats; cuSPARSE-HYB, merge-SpMV and CSR5; and $PWL_{seq.}$ only on merge-SpMV and CSR5. In other format (as we will see bellow) their performance is very low. The network has "recognized" this fact and classifies all these matrices on one of those algorithms only; but sometimes fails to predict the exact one. This is why they have such a huge top 2 accuracy, but their top 1 accuracy is much lower.

Test Accuracy

The test accuracy, on the other side, shows the actual performance on the network. For testing we used 3 sets as shown in table 4.2 ; A big test set containing both synthetic and real matrices, a real set and a final set containing larger matrices.The first test set's purpose is to show the performance of the network; it was generated by taking a random 20% of the input dataset and using it for testing; while the 80% left was used for training. It contains matrices from all sub-sets of the train test.

As we can see, even for Lenet the accuracy numbers for this test set are pretty high; and equivalent to the state-of-the-art [Zhao et al., 2018]. This would be the actual network accuracy for a normal neural network problem; but in our case, thing were not that simple. As we explained, our dataset 90% consists of synthetic matrices; as a result it was normal for the network to "assume"

these are the actual images it was tasked to label. While in the theoretical sense our network learned to predict very well for its given dataset, what we actually wanted was it to be able to predict only real matrices. And this was why our dataset "quality" was so important.

The second and third test sets were created for this exact reason; to evaluate the network on only real matrices. And this is where we see both the quality problem of the DDVT dataset and the accuracy bottleneck for Lenet. As we can see, the accuracy is far lower; and the top 2 accuracy is still not good enough. While this was a huge downfall for our approach, what we were tasked to evaluate was what these accuracy results mean performance-wise for real matrices; not network-accuracy wise (this was the first test set's role). For this purpose we chose cuSPARSE-CSR as a baseline and evaluated the prediction **performance increase**, shown in table 4.4.

		Density Lenet						CSR5
Dataset		DDVT		Mirror		Mirror + DDVT		-
Sub-set	Size	Pred.	Max	Pred.	Max	Pred.	Max	Pred.
Block_{aug}	1632	1.14	1.14	1.13	1.14	1.14	1.14	0.97
$\text{DDVT}_{resized}$	7830	2.2	2.21	1.71	2.21	2.2	2.21	1.99
$\text{PWL}_{cluster}$	495	1.77	1.77	1.77	1.77	1.77	1.77	1.66
$\text{PWL}_{seq.}$	1350	29.84	29.91	29.81	29.91	29.82	29.91	29.15
Mirror_{aug}	6655	1.25	1.5	1.5	1.5	1.5	1.5	1.36
$\text{Test}_{synthetic}$	2500	4.98	5.23	5.04	5.35	5.10	5.25	4.03
Test_{real}	416	1.25	1.57	1.29	1.57	1.26	1.57	1.31
Test_{large}	208	1.03	1.42	1.22	1.42	1.11	1.42	1.27

Table 4.4: Lenet Performance Results

Performance Evaluation

Table 4.4 shows the performance boost each method leads to. The "Maximum" column for each model is the maximum speedup that can theoretically be achieved by always predicting the best format. The "Predicted" column shows the speedup achieved my the given format. While we will evaluate all our networks with more detail in the next section, there are some interesting things to notice in this table. First, We can see that the $\text{PWL}_{seq.}$ dataset has a huge maximum performance speedup value; that is because as we explained it runs very slow with cuSPARSE-CSR. We can see that the $\text{Test}_{synthetic}$ maximum speedup is also very high; which is caused because around 15% of the its matrices originate from $\text{PWL}_{seq.}$. For the real test sets, the maximum speedup is much lower, since cuSPARSE-CSR is designed to run well on them.

Regarding predicted speedups, we can see that all train set speedups are almost maximum; and the same is true for $\text{Test}_{synthetic}$. On the other hand, on the real test sets, while the accuracy was pretty low, the speedups are considerably good; especially for the *Mirror* density Lenet. This shows that while the network fails to exactly understand the matrix structure and predict the ideal format, it predicts a format which is also good for the given matrix; just not optimal. If we cross these results with the fact we explained in the last section; that some implementations are very close to each other performance wise, we can see that in this case, the accuracy degradation might be a result of bad dataset quality and labeling; not a network problem.

While the speedup numbers are satisfying, we shouldn't forget that our baseline cuSPARSE-CSR is not the only state of the art algorithm for SpMV in GPUs. An other important implementation is CSR5, which was the best implementation for 40% of the matrices. For this reason, we included the speedup for CSR5. As we can see in the test sets, for the synthetic set where the network can predict with a good accuracy CSR5 is inferior; but for the real sets CSR5 outperforms the predictions by a small percentage.

4.2.2 CaffeNet (AlexNet)

The second CNN we chose to train was CaffeNet, an AlexNet implementation [Krizhevsky et al., 2012] of caffe with some small variations in order to run faster on it. Training took around 20 hours for each training session. The specific training metrics we used can be found in the appendix 6.4.1. For CaffeNet we ran each training model only once, since it was much slower than Lenet, again by splitting the dataset in two random parts where 80% was used for training and 20% for testing. Unlike Lenet, the results of AlexNet were disappointing. The table 4.5 shows the accuracy for the three datasets also used in Lenet; while AlexNet trained on all six combinations, the result was the same for all categories. The accuracy was very low not only on the test sets, but on the train sets as well; and it did not increase or decrease after any number of iterations. That lead us to understand that something was wrong with the process of the training; as we will see the network was stuck on a local minimum point.

		Binary Caffenet		Density Caffenet			
Dataset		DDVT		Mirror		DDVT + Mirror	
Sub-set	Size	Acc.	Top 2 Acc.	Acc.	Top 2 Acc.	Acc.	Top 2 Acc.
Block_{aug}	1632	0.17	0.17	0.17	0.39	0.17	0.17
$\text{DDVT}_{resized}$	7830	0.54	0.73	0.54	0.6	0.54	0.73
$\text{PWL}_{cluster}$	495	0.23	0.34	0.23	0.23	0.23	0.34
$\text{PWL}_{seq.}$	1350	0.26	1.0	0.26	0.26	0.26	1.0
Mirror_{aug}	6655	0.49	0.53	0.49	0.67	0.49	0.53
$\text{Test}_{synthetic}$	2500	0.40	0.55	0.42	0.47	0.41	0.52
Test_{real}	416	0.48	0.48	0.48	0.74	0.48	0.48
Test_{large}	208	0.36	0.47	0.36	0.43	0.36	0.47

Table 4.5: CaffeNet test results

The problem that AlexNet faced was pretty simple; in table 4.6 we can see that the network predicts always the CSR5 format. This is not something unusual in neural networks; when one class is much bigger than the others, it is a usual phenomenon. In our case, as we explained in Lenet, for most arrays the performance difference was also very small, implying a close structural form for some matrices. For the mirror and DDVT datasets the CSR5 label percentage is 39% and 42% ; which is near half the total dataset; if we include the 15% of the matrices which had a performance difference for these formats less than 2% (which were labeled randomly), we end up with more than half our dataset, having a somewhat specific structure which AlexNet classified as a "CSR5 label structure". That lead to AlexNet "believing" that classifying all matrices as CSR5 would result in the optimal accuracy. In order to correctly train AlexNet, we would require a more divergent dataset; and with more matrices (since 10000 is a small number for Deep CNNs).

		Binary Caffenet		Density Caffenet			
Dataset		DDVT		Mirror		DDVT + Mirror	
Format	Size	Recall	Ac.	Recall	Ac.	Recall	Ac.
cuSPARSE-csr	1571	0.0	0.0	0.0	0.0	0.0	0.0
cuSPARSE-hyb	1742	0.0	0.0	0.0	0.0	0.0	0.0
bhSPARSE(CSR5)	8485	0.46	1.0	0.46	1.0	0.46	1.0
lightSpMV-warp	2184	0.0	0.0	0.0	0.0	0.0	0.0
merge-SpMV	2854	0.0	0.0	0.0	0.0	0.0	0.0
cuSPARSE-BSR(4)	1750	0.0	0.0	0.0	0.0	0.0	0.0

Table 4.6: CaffeNet Per-format accuracy

4.2.3 Googlenet

The last and most promising CNN we trained was Googlenet. It outperformed both Lenet and AlexNet; but was not very far from Lenet. The specific training metrics we used can be found in the appendix 6.4.1. For GoogleNet we also ran each training model only once, since it also required a lot of time, again by splitting the dataset in two random parts where 80% was used for training and 20% for testing. Googlenet training took about 40-60 hours for each model trained, for a total of 3-4 weeks. In table 4.7 we can see the networks accuracy for Binary and Density Googlenet, ran on the *Mirror* and the *Mirror+ DDVT* dataset. Unlike Lenet, Googlenet ran on all six combinations; we will not present the *DDVT* dataset results since in Googlenet's case, the dataset which contained both sub-sets (*Mirror+ DDVT*) surpassed it, and kept all its interesting characteristics we will explain bellow. The colour representation for the table is the same with Lenet's; Cyanide for valid test accuracy and grey for train accuracy.

		Binary Googlenet				Density Googlenet			
Dataset		Mirror		DDVT + Mirror		Mirror		DDVT + Mirror	
Sub-set	Size	Ac.	Top 2 Ac.	Ac.	Top 2 Ac.	Ac.	Top 2 Ac.	Ac.	Top 2 Ac.
Block_{aug}	1632	0.81	0.95	0.87	0.96	0.84	0.94	0.91	0.97
$\text{DDVT}_{resized}$	7830	0.25	0.44	0.86	0.95	0.3	0.48	0.86	0.95
$\text{PWL}_{cluster}$	495	0.74	0.88	0.75	0.91	0.84	0.96	0.8	0.92
$\text{PWL}_{seq.}$	1350	0.82	0.96	0.81	1.0	0.82	0.96	0.81	1.0
Mirror_{aug}	6655	0.94	0.97	0.9	0.97	0.95	0.98	0.96	0.99
$\text{Test}_{synthetic}$	2500	0.90	0.97	0.85	0.95	0.91	0.97	0.85	0.94
Test_{real}	416	0.44	0.71	0.49	0.65	0.43	0.65	0.57	0.73
Test_{large}	208	0.33	0.5	0.34	0.56	0.43	0.66	0.38	0.58

Table 4.7: Googlenet test results

Binary Googlenet

The first and most important difference from Lenet is the Binary implementation; Googlenet binary managed to train and produce solid accuracy unlike Lenet which could not even train. The basic reasons Googlenet managed to surpass Lenet in this case were three; the use of mean subtraction, the much deeper architecture, and the designing of the network for images of bigger resolution. The most important was mean subtraction; without it Googlenet also failed to train on the binary images (we tried both AlexNet and Googlenet with and without the mean subtraction to find the

better combination). Apart from that, the deep nature of the network was also critical, since it helped the network surpass the 40% label prediction after a considerable amount of iterations; specifically it took 10000 iterations to break from this bottleneck, which is Lenet's total iteration number.

Density Googlenet

The Density Googlenet implementation is quite similar to Lenet Density. The main difference is that we also used mean subtraction as per the binary Googlenet; in order to normalize our data. Unlike binary Googlenet, in this case the versions with and without mean subtraction were nearly identical accuracy wise, so we used the mean one for comparison reasons. The accuracy results displayed on the table were pretty satisfying for both train sets. Unfortunately, the problem of accuracy drop on the real matrix test sets was not eliminated; but there was a considerable improvement over Lenet which we will analyze further in the next section.

		Binary Googlenet		Density Googlenet	
Dataset		Mirror	DDVT + Mirror	Mirror	DDVT + Mirror
Format	Size	Acc.	Acc.	Acc.	Acc.
cuSPARSE-csr	1571	0.55	0.79	0.69	0.85
cuSPARSE-hyb	1742	0.77	0.81	0.79	0.91
bhSPARSE(CSR5)	8485	0.61	0.89	0.61	0.91
lightSpMV-warp	2184	0.76	0.79	0.74	0.87
merge-SpMV	2854	0.4	0.85	0.41	0.85
cuSPARSE-BSR(4)	1750	0.3	0.87	0.51	0.84

Table 4.8: Googlenet Per-format Results

Per-Format Accuracy

In terms of per-format prediction accuracy, table 4.8 shows the corresponding results. In contrast with the equivalent Lenet table, we didn't use recall as a metric here, since the results were almost identical to Lenet. There are some very interesting and important things to comment on this table; which were critical for expanding and correcting our datasets. The first one is the very low accuracy of the Mirror train set on merge-SpMV and cuSPARSE-BSR. This is caused because on total calculations of format accuracy the DDVT set was used as well; which shows that the Mirror dataset is bad at predicting DDVT arrays. In contrast to what accuracy shows, this is not a bad thing; it was **the reason** the Mirror dataset was created in the first place; to diminish the DDVT dataset's "bad" influence on block matrix prediction. It is thus important that the Mirror dataset has this exact characteristic; it shows that it succeeded on eliminating the DDVT's corrupted blocks. This can also be seen in table 4.7 from the very low accuracy of the Mirror train set on the DDVT test set.

The second one is the Binary-Density Googlenet difference. Since the two implementations were pretty close accuracy wise on general results, it was impossible to isolate the difference from table 4.7. On the other hand, we can see here **where** Density Googlenet was superior; and that is the cuSPARSE implementations. This is caused by the " generally average" nature of cuSPARSE as a library; focusing on running well on most matrices, without explicitly being optimized for

certain ones. That creates a huge bottleneck for the network; being unable to extract specific structural information that makes a matrix "ideal" for these implementations. For this reason, in binary representation the network is unable to correctly predict these formats. On the other hand, the Density representation gives extra information on the network, the density per pixel, allowing the Googlenet's deep structure to utilize an extra metric resulting in accuracy increase.

Train Accuracy

Again, we use train accuracy to recognize the bottlenecks of the network, not evaluate it. As expected, the train accuracy is also high; but the important here is the binary-density implementations comparison, since in Lenet the binary one did not train correctly. The biggest difference is on the Block$_{aug}$ and PWL$_{cluster}$ train sub-sets; the binary implementation has 5-10% accuracy drop compared to the density one. The reason is quite simple; these are the most area-dense train sub-sets (meaning they have specific areas where they have many concentrated nonzeros), which makes the density information that binary fails to provide very important. Apart from that, it is important to notice the top 2 accuracy superiority of the *Mirror+ DDVT* datset compared to the *Mirror* one. Unlike Lenet, Googlenet is a deep learning network; providing extra input on the network (even if in our case this is the "bad" DDVT sub-set) increases the ability of the network to predict on different situations. While including the DDVT sub-set to the train set causes a small performance degradation for some sub-sets, in the case of Block$_{aug}$ and Mirror$_{aug}$ it increases testing performance for about 5-10 %, which is a huge improvement.

Test Accuracy

In these section we will explore the Googlenet accuracy results on the independent test sets, which were used for evaluation. The test sets were the same with Lenet; one synthetic test set and two real test sets including small and large matrices respectively. In this section the superiority of the *Mirror* train set for test accuracy becomes obvious. While the results for the only *DDVT* training are not presented here, the *Mirror+ DDVT* keeps the same behavioural elements. In the synthetic test set, the *Mirror* train set lead to 5-6% more prediction accuracy than *Mirror+ DDVT* (the difference from single *DDVT* was up to 10%); which was a huge difference (for testing at least). That meant that the *Mirror* train set lead to much better training and fitting, being able to recognize the synthetic matrices far easier and much more accurately.

Unfortunately, fitting in a synthetic train set when the target is predicting real matrices is a form of **over-fitting**. The *Mirror* train set faced this exact problem; it fit so well on predicting synthetic matrices that the prediction rate on real ones dropped. The same was true for the *DDVT* train set, which suffered far more from over-fitting leading to even lower accuracy (7% lower from *Mirror*) for real matrices. On the contrary, the total *Mirror+ DDVT* train set was better at predicting real matrix labels; since by combining the "bad" DDVT with the much better *Mirror* train set, the network became better at predicting matrices with structures fully unknown to it (which shouldn't be the case for any matrix, but as we said was impossible for our resources).

Performance Evaluation

In table 4.9 we show the corresponding speedups for our models. The percentage displayed is again the speedup from using cuSPARSE-csr. Here we can see **why** Googlenet is considerably better than Lenet. Regarding train subset speedup, the results are a little better that Lenet, but the difference is irrelevant since we do not want to use them for evaluation. In the synthetic test set, Googlenet gives a far better speedup than Lenet, because of the much higher accuracy achieved by the *Mirror* train set. In the real test sets, the total *Mirror+ DDVT* train test leads to better results than Lenet for the small matrices (since in the large matrices, the problem of not having enough available classification information doesn't change by using a deeper network).

It is interesting to note here the performance speedup over the state of the art CSR5 implementation, since now there is an actual difference (with the exception of the large matrix test suite where the results are almost identical to Lenet). In Lenet, the *Mirror+ DDVT* density Speedup on the $\text{Test}_{synthetic}$ set compared to CSR5 was **25%**, while in Googlenet it is **31%**, which is a considerable difference. Also, for the Test_{real} set, Googlenet manages to outperform CSR5 for **5%**, which is a very small speedup, but an improvement nonetheless. It is important to remember that in our case, the maximum speedup over CSR5 is not very big for the real matrix suites; the max speedup for Test_{real} is **20%** and for Test_{large} it is only **12%**.

		Binary GoogleNet				Density GoogleNet			
Dataset		Mirror		DDVT + Mirror		Mirror		DDVT + Mirror	
Sub-set	Size	Pred.	Max	Pred.	Max	Pred.	Max	Pred.	Max
Block_{aug}	1632	1.11	1.14	1.12	1.14	1.12	1.14	1.13	1.14
$\text{DDVT}_{resized}$	7830	1.62	2.22	2.18	2.22	1.68	2.21	2.16	2.21
$\text{PWL}_{cluster}$	495	1.76	1.77	1.76	1.77	1.77	1.77	1.77	1.77
$\text{PWL}_{seq.}$	1350	29.77	29.91	29.81	29.91	29.8	29.91	29.81	29.91
Mirror_{aug}	6655	1.46	1.5	1.48	1.5	1.47	1.5	1.5	1.5
$\text{Test}_{synthetic}$	2500	5.28	5.35	5.28	5.41	5.32	5.35	5.29	5.41
Test_{real}	416	1.23	1.57	1.29	1.57	1.27	1.57	1.35	1.57
Test_{large}	208	1.19	1.42	1.14	1.42	1.23	1.42	1.2	1.42

Table 4.9: GoogleNet Performance Results

Chapter 5

Conclusions

In this thesis, we have explored the optimization of Sparse Matrix vector Multiplication (SpMV) in GPUs with format prediction using CNNs. We started by extensively analyzing the background on SpMV optimization techniques for GPUs and all the previous work on format prediction, and selected 6 state-of-the-art GPU SpMV implementations as classification labels for our problem. Then, we created 5 divergent datasets for the total of 18000 synthetic matrices for the classification problem and ran the SpMV format benchmarks for all matrices to create the dataset labels. In addition, we implemented 2 different image types for the dataset's matrices; a binary and an RGB density representation. Finally, we trained 3 different CNNs, Lenet, AlexNet and Googlenet and evaluated and compared their individual performance. In this chapter we will present the possible future work or research on the subject.

5.1 Future Work

Regarding future work on the subject, there are three basic directions we believe that must be explored. The first is regarding the experimental layout of this approach. While with our experiments we were lead to believe that using CNNs for SpMV format selection is an interesting approach of the problem, and could lead to considerable speedups even over state of the art algorithms, our inability to create the ideal dataset due to lack of resources still leaves this question open. That means that, more research regarding input transformation and matrix generation is required to create a more accurate model for real matrices, because while our models showed exceptional accuracy on synthetic test sets, they had a huge accuracy drop on real ones; meaning they were not representative enough.

The second one, which we think is the most interesting for future research (since the other two are more experimental and technical, not theoretical), is the matrix representation used for the networks input. In our case, we used a picture representation of the nonzeros location(binary) and sparsity pattern (density). These representations face some major problems we already mentioned; the inability to extract certain structural characteristics like blocks or very long rows. To surpass these problems, extra information could be given to the network; in our case through images. A similar idea has been proposed before in [Zhao et al., 2018], by using the histogram of a given matrix as input; but we believe further improvement is possible. Sticking to the image representation, we would propose using one of the three RGB channels to pass information about the blocks, av-

erage low length and/or other useful metrics for classification proposed on previous works. Or, as also suggested in [Zhao et al., 2018], training another network with completely different input and late-merging the prediction results.

The third one is how our work (or any other work of this nature) could be integrated in actual scientific libraries. As we explained, we assumed for our approach that the training is not part of this pre-processing; the only time we count as part of this is the network deploy time, since in a real world scenario the network would be pre-trained. But even with this assumption, there are still a lot of challenges to face. Using this classifier implies that the user has access to all libraries used as labels, with each one of them ready to be deployed after prediction. That requires the creation of a global infrastructure containing all these implementations, which should be easily re-trainable and adjustable to the now possible state of the art SpMV implementations; which is not an easy task. Even in this case, pre-processing time for a given matrix should be taken into account, since generating the image representation also takes time; so a fast image creation scheme is required as well. And all this should be integrated in an existing linear algebra library in order for the method to be widely used.

Chapter 6

Appendix

In this chapter we will include the most important parts of our code, split in three categories:

- Dataset and image generation
- Benchmarking Formats
- CNN related data augmentation and evaluation

6.1 Required Datatypes

In this section we will list the most basic object classes used in our scripts, so we can carelessly reference them later.

6.1.1 scipy.sparse.coo_matrix (arg1, shape=None, dtype=None, copy=False)

Attributes:

- *dtype* dtype : Data type of the matrix.
- *2-tuple* shape : Shape of a matrix.
- *int* ndim : Number of dimensions (always 2).
- *int* nnz : Number of stored values, including explicit zeros.
- *dtype array* data : COO format data array of the matrix
- *int array* row : COO format row index array of the matrix
- *int array* col : COO format column index array of the matrix

6.2 Dataset generation-Related scripts

This part contains all functions used to create, augment or delete synthetic matrices, as well as all create and extract images derived from them. All code regarding the dataset augmentation was done using python. The basic scripts responsible for calling them were the following:

- **powerlaw_cluster_graph-gen.py** : Generates the PWCL dataset used in training.
- **powerlaw_seq-gen.py** : Generates the PWS dataset used in training.
- **spyplot.py** : Generates the DDVT dataset + spyplot images by resizing and then transforming all matrices in given directory.
- **Crop_crop.py** : Crops all 640×480 images in the input directory to 372×372.
- **RGB_Creator.py** : Creates the RGB density images for all matrices in given directory.

Below we will explain the arguments and outputs of the functions used in these scripts.

6.2.1 multgen (int sz)

Checks and returns the maximum resize possibility for a matrix in order to keep nnz < 2000000.
Parameters

Input	sz	nonzeros of input matrix

Returns
mult: Maximum resize parameter (mult < 5)

6.2.2 spyplot (scipy.sparse.coo_matrix x, string str1)

A funtion which plots the given sparse matrix using the spy python function.
Parameters

Input	x	The sparse matrix to be ploted
Input	str1	The name of the given matrix

6.2.3 resize (scipy.sparse.coo_matrix A, int mult)

Resizes the input matrix A of size $N \times M$ containing NNZ nonzeros to a new matrix of size $(mult \cdot N) \times (mult \cdot M)$ with $mult^2 \times NNZ$ nonzeros.
Parameters

Input	A	The sparse matrix to be resized
Input	mult	The resize scale

Returns

B: A COO sparse matrix derived from A by placing $mult \times mult$ blocks in the nonzero positions of A.

6.2.4 dg_dist (scipy.sparse.coo_matrix x, divz)

Computes the diagonal distance variation transformation (DDVT) for matrix x.
Parameters

Input	x	The sparse matrix basis for the DDVT
Input	divz	The distance parameter for the transformation (See alg. 8)

Returns
z: scipy.sparse.coo_matrix of same size with A after applying the DDVT transformation.

6.2.5 dg_dist_ng (scipy.sparse.coo_matrix x, divz)

Computes the reverse diagonal distance variation transformation (DDVT) for matrix x. Can be used for either resetting a modified matrix or creating a reversely-transformed one.

Parameters

Input	x	The sparse matrix basis for the DDVT
Input	divz	The distance parameter for the transformation

Returns
nz: scipy.sparse.coo_matrix of same size with A after applying the reverse DDVT transformation.

6.2.6 crop_binary (string fpath)

Crops the binary image to 372×372.
Parameters

Input	fpath	Path to the image to be cropped

Returns
0 or 1: 1 for success, 0 for failure

Possible error conditions
Returned 0: Wrong input image resolution

6.2.7 is_non_zero_file (string fpath)

Checks if given fpath is a valid file.
Parameters

Input	fpath	Path to the file

Returns
True or False: True if fpath points to a nonzero file, False otherwise

6.2.8 density_RGB (int den, int dtx, int dty)

Calculates the RGB pixel values for given mapped datum.
Parameters

Input	den	Nonzeros of the given datum
Input	dtx	Dimension x of the datum
Input	dty	Dimension y of the datum

Returns
(R,G,B): A 3 value-tuple containing the RGB pixel values of the mapped datum.

Possible error conditions
Returned (-1,-1,-1): Density calculation lead to infinity-overflow.

6.2.9 datum_mapping(scipy.sparse.coo_matrix A)

The core function for the RGB density pixel mapping. Calculates the mapping datum sizes and nonzero counts and invokes *density_RGB* for each datum to calculate the RGB image array.
Parameters

Input	A	The sparse matrix whose image will be calculated.

Returns
pic: An $256 \times 256 \times 3$ array containing the RGB density representation of A.

Possible error conditions
Returned 0 or 1: 0 for Uneven input matrix, 1 for $\text{datum}_{dim} = 0$

6.3 Benchmark Scripts

This section contains all functions used during benchmarking the dataset for all formats and extracting useful benchmark metrics information from the outputs. It contains C, python and bash script files. The important files for each category are listed below:

C files

- **rootdir/main-cuSPARSE.cu** : Code we implemented for invoking the cuSPARSE kernels.
- **rootdir/Ext_lib/input.cu** : Our functions for Matrix Market File input
- **rootdir/Ext_lib/gpu_util.cu** : Some helper functions for main-cuSPARSE.

- **rootdir/Ext_lib/dmv.c** : Vector creation and serial SpMV for verification.
- **rootdir/Ext_lib/alloc.c** : Memory allocation helper functions.
- **rootdir/Ext_lib/timer.c** : The universal timer we implemented for benchmarking all formats.
- **rootdir/cusplibrary-0.5.1/performance/spmv/spmv.cu** : The cuSP library kernel invoker.
- **rootdir/bhSPARSE/CSR5_cuda/main.cu** : The CSR5 library kernel invoker.
- **rootdir/lightSpMV/lightspmv/src/main.cu** : The lightSpMV library kernel invoker.
- **rootdir/ViennaCL-1.7.1/examples/benchmarks/sparse.cu** : The ViennaCL library kernel invoker.
- **rootdir/merge-spmv-master/gpu_spmv.cu** : The Merge-SpMV library kernel invoker.

Bash Scripts

- **rootdir/run_benchmark.sh** : Executes all format benchmarks (except BSR) for all files in given dir.
- **rootdir/run_BSR.sh** :Executes cuSPARSE BSR for block_size 3 and 4 for all files in given dir.

Python Scripts

- **plot_gpu_gflops.py** : Generates a GFLOPs per format graph per array graph.
- **plot_gpu_roofline.py** : Generates a roofline graph for given arrays [Williams et al., 2009].
- **plot_gpu_trans_time.py** : Generates a preprocessing time per format graph per array graph.

Following are the C functions used for this part:

6.3.1 double csecond(void)

The universal timer we used for all benchmarks.

Returns
The exact system time in seconds.

6.3.2 void dmv_csr(int * csrPtr, int *csrCol, double * csrVal, double *x, double *ys, int n)

A serial SpMV csr implementation for result validation.

Parameters

Input	csrPtr	A pointer to the CSR row pointer vector.
Input	csrCol	A pointer to the CSR column index vector.
Input	csrVal	A pointer to the CSR values vector.
Input	x	A pointer to the input vector.
Input, output	ys	A pointer to the output vector.
Input	n	The nonzero count of the input matrix.

Returns

The output vector ys where $ys = A \cdot x$.

6.3.3 int vec_equals(const double *v1, const double *v2, size_t n, double eps)

Compares two vectors of same size.

Parameters

Input	v1	A pointer to the first vector.
Input	v2	A pointer to the second vector.
Input	n	The size of the vectors.
Input	eps	The element-wise comparison precision.

Returns

k: The number of different elements between the two vectors.

6.3.4 void vec_init(double *v, size_t n, double val)

Initialize all elements of the input vector to val.

Parameters

Input, output	v	A pointer to the input vector.
Input	n	The size of the vector.
Input	val	The element-wise initialization value.

6.3.5 void vec_init_rand(double *v, size_t n, double max)

Initialize all elements of the input vector to a random value in the range (0,max).

Parameters

Input, output	v	A pointer to the input vector.
Input	n	The size of the vector.
Input	max	The max value the output vector can contain.

6.3.6 void vec_init_rand_p(double *v, size_t n, size_t np, double max)

Initialize the first n elements of the input vector to a random value in the range (0,max) and the rest np to 0 (vector padding).

Parameters

Input, output	v	A pointer to the input vector.
Input	n	The size of the original vector.
Input	np	The size of the padding to be added.
Input	max	The max value the output vector can contain.

6.3.7 static void check_result(double *test, double *orig, size_t n)

Tests if two vectors of size n are equal (prec = 0.0001) by invoking vec_equals.

Parameters

Input	test	Vector to be tested.
Input	orig	The correct vector.
Input	n	Vector Length.

6.3.8 static void report_results(double timer, int flops, int bytes)

Prints time (ms), performance (GFLOPs) and bandwidth (GBs).

Parameters

Input	timer	Total execution time.
Input	flops	Total algorithm floating point operations.
Input	bytes	Total algorithm memory movement in bytes.

6.3.9 void cudaCheckErrors(const char * msg)

Print msg and exit(1) if there is any CUDA error on the stack.

Parameters

Input	msg	Pointer to the error message string.

6.3.10 void *gpu_alloc(size_t count)

Allocate count bytes in the GPU memory.

Parameters

Input	count	Nymber of bytes to be allocated.

Returns

ret: A pointer to *count* bytes of GPU memory.

6.3.11 void gpu_free(void *gpuptr)

Free the GPU memory pointed by gpuptr.

Parameters

Input	gpuptr	A pointer to a GPU-memory address.

6.3.12 int copy_to_gpu(const void *host, void *gpu, size_t count)

Copy count bytes from the host memory to the GPU memory.

Parameters

Input	host	A pointer to a host-memory address.
Input, output	gpu	A pointer to a GPU-memory address.
Input	count	Nymber of bytes to be copied.

Returns

1 if sucess

Possible error conditions

cudaError_t: If copy fails the error is printed and the program exits.

6.3.13 int copy_from_gpu(void *host, const void *gpu, size_t count)

Copy count bytes from the GPU memory to the host memory.

Parameters

Input	host	A pointer to a host-memory address.
Input, output	gpu	A pointer to a GPU-memory address.
Input	count	Nymber of bytes to be copied.

Returns

1 if sucess

Possible error conditions

cudaError_t: If copy fails the error is printed and the program exits.

6.3.14 double gpu_memory_start_count(void)

Calculate size of used GPU memory.

Returns

GPU memory usage (GB)

6.3.15 double gpu_memory_stop_count (double used)

Calculate difference in used GPU memory from given value (intended use: last gpu_memory_start_count(void) call as input).

Parameters

Input	used	The output of the last gpu_memory_start_count(void) call.

Returns

GPU memory usage (GB) in an interval.

6.3.16 void gpu_memory_print()

Prints total, free and used GPU memory values.

Possible error conditions

cudaError_t: If cudaMemGetInfo fails the error is printed and the program exits.

6.3.17 int mtx_read1(int ** csrRow, int ** cooCol, double ** cooVal, int * n, int * m, int * n_z, char * name)

Read a sparse matrix for file "name" into unified memory.

Parameters

Input, output	csrRow	A pointer to the pointer which will point to the CSR row pointer vector.
Input, output	cooCol	A pointer to the pointer which will point to the CSR column index vector.
Input, output	cooVal	A pointer to the pointer which will point to the CSR value vector.
Input, output	n	A pointer to the $\dim_x$ of the input matrix.
Input, output	m	A pointer to the $\dim_y$ of the input matrix.
Input, output	n_z	A pointer to the nonzero count of the input matrix.
Input	name	A string containing the .mtx file path.

Returns

1 if sucess

Possible error conditions

cudaError_t: If unified memory allocation fails.

6.3.18 void get_nz_symmetric(int * n_z, char* name)

Get the nonzero count of elements from a symmetric matrix.

Parameters

Input, output	n_z	A pointer to the nonzero count of the input matrix.
Input	name	A string containing the .mtx file path.

Possible error conditions

Self-generated Error in symetric read pass.

6.3.19 void csr_transform(float ** A, int n, int m, int n_z, float *csrValA, int *csrRowPtrA, int *csrColIndA)

Transforms a full matrix to CSR format (Used for function testing purposes).

Parameters

Input	A	An input matrix in dense format.
Input	n	The x-dimension of the input matrix.
Input	m	The y-dimension of the input matrix.
Input	n_z	The nozero count of the input matrix.
Input, output	csrValA	A pointer to the CSR value vector.
Input, output	csrRowPtrA	A pointer to the CSR row pointer vector.
Input, output	csrColIndA	A pointer to the CSR column index vector.

Possible error conditions

Error in n_z count in the A matrix.

6.3.20 void quickSort (int *a, int * b, double * c, int l, int r)

A quickSort implementation for sorting the COO vectors.

Parameters

Input, output	a	A pointer to the COO row index vector.
Input, output	b	A pointer to the COO column index vector.
Input, output	c	A pointer to the COO value vector.
Input	l	QuickSort length parameter.
Input	r	QuickSort recursion parameter.

6.3.21 int partition (int *a, int * b, double * c, int l, int r)

The partition helper function for quicksort.

Parameters

Input, output	a	A pointer to the COO row index vector.
Input, output	b	A pointer to the COO column index vector.
Input, output	c	A pointer to the COO value vector.
Input	l	QuickSort length parameter.
Input	r	QuickSort recursion parameter.

6.4 Network training Scripts

This sections contains all scripts used in initializing, training, testing and evaluating the neural networks. It consists mostly of Python and bash script files, along with the required network initialization files. While for training we used the caffe framework, for testing and validation we used pycaffe, a python library for caffe. We will not include their functions here since they are not our work.

Bash scripts

- **traindir/create_imagenet.sh** : Generates from a given label file the lmdb database for all classification images in the file.
- **traindir/make_imagenet_mean.sh** : Computes the pixel mean for a given LMDB train and test set.
- **traindir/train_caffenet.sh** : The script which invokes the neural network (based on the example caffe script).
- **traindir/resume_training.sh** : Resumes training from a given solverstate snapsot.

Network Initialization files

- **traindir/Arch/lenet/lenet_solver.prototxt** : The Solver initialization file for lenet.
- **traindir/Arch/lenet/lenet_train_test.prototxt** : The model architecture file for lenet.
- **traindir/Arch/lenet/lenet.prototxt** : The deploy file for lenet (used for validation).
- **traindir/Arch/Googlenet/solver.prototxt** : The Solver initialization file for Googlenet.
- **traindir/Arch/Googlenet/train_val.prototxt** : The model architecture file for Googlenet.
- **traindir/Arch/Googlenet/deploy.prototxt** : The deploy file for Googlenet (used for validation).
- **traindir/Arch/caffenet/solver.prototxt** : The Solver initialization file for Alexnet(caffenet).
- **traindir/Arch/caffenet/train_val.prototxt** : The model architecture file for Alexnet(caffenet).
- **traindir/Arch/caffenet/deploy.prototxt** : The deploy file for Alexnet (used for validation).

Python Scripts

- **bench_stats.py** : Prints some basic metrics for input .out benchmark file.
- **Lmdb.py** : Loads an lmdb file and passes it to the network (for image representation validation).
- **import_vienna.py** : Compacts a vienna benchmark file with a general benchmark file (.log -> .out1)
- **import_merge.py** : Compacts a merge benchmark file with a 1st-level pre-processed benchmark file (.out1-> .out).
- **import_bsr.py** : Compacts a bsr benchmark file with a 2nd-level pre-processed benchmark file (.out -> .outf)
- **convert_data_layout.py** : Compacts and converts any number of input final benchmark files (.outf) to an image label file. Can be used for any number of labels.
- **convert_data_layout_true_max.py** : Compacts and converts any number of input benchmark files to an image label file without random choice for close implementations.
- **Split_train_test.py** : Splits an image label file to train and test sub-sets (using a random $1/5$ as the test set).
- **validate.py** : Launches a trained network and tests it on any number of test sets, outputs training results.

6.4.1 CNN architecture and train variable scripts

In this section we will present the specific initialization files for all 3 networks. The network architecture was not changed (except the input size and output label number) so we will not include these here; they can be found either in the original Caffe framework (`http://caffe.berkeleyvision.org/`) or in our thesis individual repository.

Lenet lenet_solver.prototxt

The train/test net protocol buffer definition
net: ”/path/to/lenet/lenet_train_test.prototxt”
test_iter: 100
test_interval: 500
The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
The learning rate policy
lr_policy: ”inv”
gamma: 0.0001

power: 0.75
display: 100
max_iter: 10000
snapshot: 5000
snapshot_prefix: ”/path/to/Solverstates/Density_Lenet”
solver_mode: GPU

CaffeNet solver.prototxt

net: ”/path/to/caffenet/train_val.prototxt”
test_iter: 1000
test_interval: 1000
basevlr: 0.01
lr_policy: ”step”
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 10000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: ”/path/to/Solverstates/Density_Caffenet”
solver_mode: GPU

GoogleNet solver.prototxt

net: ”/path/to/Googlenet/train_val.prototxt”
test_iter: 1000
testvinterval: 4000
test_initialization: false
display: 40
average_loss: 40
basev_lr: 0.01
lr_policy: ”step”
stepsize: 100000
gamma: 0.96
max_iter: 200000
momentum: 0.9
weight_decay: 0.0002
snapshot: 10000
snapshot_prefix: ”/path/to/Solverstates/Density_Googlenet”
solver_mode: GPU

List of Figures

List of Tables

List of Algorithms

Bibliography

[Anzt et al., 2014] Anzt, H., Tomov, S., and Dongarra, J. J. (2014). Implementing a sparse matrix vector product for the sell-c/sell-c-σ formats on nvidia gpus.

[Asanović et al., 2006] Asanović, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[Barrett et al., 1994] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA.

[Daga and Greathouse, 2015] Daga, M. and Greathouse, J. L. (2015). Structural agnostic spmv: Adapting csr-adaptive for irregular matrices. pages 64–74.

[Davis, 2006] Davis, T. (2006). *Direct Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics.

[Duff et al., 1989] Duff, I. S., Erisman, A. M., and Reid, J. K. (1989). *Direct Methods for Sparse Matrices.* Clarendon Press, New York, NY, USA.

[Elafrou et al., 2017] Elafrou, A., Goumas, G. I., and Koziris, N. (2017). Performance analysis and optimization of sparse matrix-vector multiplication on modern multi- and many-core processors. *CoRR*, abs/1711.05487.

[Filippone et al., 2017] Filippone, S., Cardellini, V., Barbieri, D., and Fanfarillo, A. (2017). Sparse matrix-vector multiplication on gpgpus. *ACM Trans. Math. Softw.*, 43(4):30:1–30:49.

[Greathouse and Daga, 2014] Greathouse, J. L. and Daga, M. (2014). Efficient sparse matrix-vector multiplication on gpus using the csr storage format. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 769–780.

[Hestenes and Stiefel, 1952] Hestenes, M. R. and Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409–436.

[Hou et al., 2017] Hou, K., c. Feng, W., and Che, S. (2017). Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi- and many-core processors. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 713–722.

[Im and Yelick, 2001] Im, E.-J. and Yelick, K. (2001). Optimizing sparse matrix computations for register reuse in sparsity. In Alexandrov, V. N., Dongarra, J. J., Juliano, B. A., Renner, R. S., and Tan, C. J. K., editors, *Computational Science — ICCS 2001*, pages 127–136, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Karakasis, 2012] Karakasis, V. (2012). *Optimizing the Sparse Matrix-Vector Multiplication kernel for Modern Multicore Computer Architectures*. National Technical University of Athens, New York, NY, USA.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. pages 1097–1105.

[Lecun et al., 1998] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Li et al., 2013] Li, J., Tan, G., Chen, M., and Sun, N. (2013). Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication. *SIGPLAN Not.*, 48(6):117–126.

[Liu and Vinter, 2015] Liu, W. and Vinter, B. (2015). CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication. *CoRR*, abs/1503.05032.

[Liu and Schmidt, 2015] Liu, Y. and Schmidt, B. (2015). Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus. pages 82–89.

[Merrill and Garland, 2016] Merrill, D. and Garland, M. (2016). Merge-based parallel sparse matrix-vector multiplication. pages 678–689.

[Saad, 2003] Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition.

[Saad and Schultz, 1986] Saad, Y. and Schultz, M. H. (1986). Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869.

[Sedaghati et al., 2015] Sedaghati, N., Mu, T., Pouchet, L.-N., Parthasarathy, S., and Sadayappan, P. (2015). Automatic selection of sparse matrix representation on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 99–108, New York, NY, USA. ACM.

[Singer, 2013] Singer, G. (2013). The history of the modern graphic processor.

[Szegedy et al., 2014] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. *CoRR*, abs/1409.4842.

[Williams et al., 2009] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76.

[Zhao et al., 2018] Zhao, Y., Li, J., Liao, C., and Shen, X. (2018). Bridging the gap between deep learning and sparse matrix format selection. *SIGPLAN Not.*, 53(1):94–108.