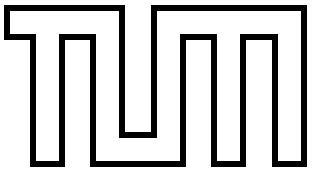


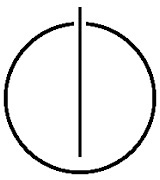
DEPARTMENT OF INFORMATICS

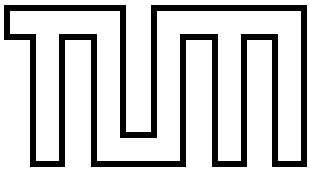
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

GPU-optimised implementation of high-dimensional tensor applications

Thomas Hörmann





DEPARTMENT OF INFORMATICS

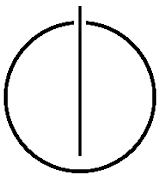
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

GPU-optimised implementation of high-dimensional tensor applications

GPU-optimierte Implementierung hochdimensionaler Tensor-Anwendungen

Author:	Thomas Hörmann
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Dipl.Inf. Christoph Riesinger
	Dipl.Math. Konrad Waldherr
Date:	December 15, 2014



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, the 15. December 2014 Thomas Hörmann

Acknowledgments

I want to thank my advisors Christoph Riesinger and Konrad Waldherr for their great advice and assistance. I would also like to extend my thanks to the chair of scientific computing, especially Kilian Röhner, for letting me use their hardware. I also wish to acknowledge the help provided by Melanie Fiener.

Abstract

Tensor Trains are a sparse representation of high dimensional tensors. They are used in quantum many body physics in order to find their ground state. For the numerical computation, a lot of inner products are needed. The computation of the inner product of two tensor trains can be done on parallel machines like multicore CPUs, CPU clusters and GPUs. Implementations for GPUs are possible in OpenCL as well as CUDA. Also vector extension like SSE and AVX would speed up the computations on CPUs. We propose a fast and efficient implementation for tensor train contractions in CUDA. The algorithm reaches up to 88% of the theoretical peak performance on the latest Nvidia architecture. The proposed algorithm can also profit from the use of multi GPU systems.

Contents

Part I.

Introduction and Theory

1. Introduction

In quantum many-body physic, the computation of the ground state is an important task[3]. This can be done by finding the smallest eigenvector x of a tensor called the Hamiltonian H. The Hamiltonian contains information about the spin of each particle and their correlation. The spin of each particle is described by a 2×2 Pauli matrix [10]. Since the size of x and H grows exponentially with the number of particles n, the values can't be stored even on large computers for bigger n. For the handling of such big tensors, some sparse representations, like the Tucker decomposition[4] and the Tensor Train decomposition[19], have been developed. The smallest eigenvector can be found by the minimization of the Rayleigh quotient.

$$\min_{x \in \mathcal{H}} \frac{x^H H x}{x^H x} \tag{1.1}$$

For numerical algorithms, the inner product of two Tensor Trains is needed. In this thesis, we will concentrate on the efficient implementation of the inner product of two Tensor Trains on GPUs, with and without cyclic boundary condition. Those are needed for computing the ground state of one dimensional particle chains. We use an optimal ordering of the tensor contractions in the sense of minimizing the required floating point operations[10].

The matrix-matrix multiplication is an example for a procedure, which runs efficient on a GPU. So algorithms who massively uses matrix products can be accelerated by GPUs. The contraction of two tensors from a Tensor Train can be computed with the same or similar code as for the matrix-matrix multiplication. Therefore we expect that the algorithms for the Tensor Train contraction can be accelerated with GPUs.

The outline of this thesis is structured as follows. Chapter 2 introduces some basics about tensors and operations on tensors. In Chapter 3 we discuss some principles for programming on GPUs by means of CUDA. We explain the hardware and show some guidelines for writing efficient code. In Chapter 4 and 5 we present different algorithms for the contraction of Tensor Trains. While we concentrate on the performance of different coding techniques in Chapter 4, we focus on the memory requirement in Chapter 5. In Chapter 6 we present experimental results of our proposed algorithms. We compare our algorithms and the different architecture generations. Finally, we conclude the thesis in Chapter 7.

2. Tensors

In the last few years tensors, as the d-dimensional generalization of matrices, have become very important. The treatment of dense $n \times n$ matrices can easily exceed the memory for large n. This problem is even worse for high dimensional ($d > 3$) tensors. The memory requirement increases exponential to the dimension (n^d). This is sometimes referred as the curse of dimensions. There have been a huge research on low rank tensor approximations[8]. Different representations, like the Tucker decomposition[4] and Tensor Trains[19], has relaxed the issue. A Tensor Train is a numerical approximation of a high dimensional tensors, which reduces the memory requirement drastically. Even problems of the size 1000^{1000} have become computable [9].

2.1. Indices

A tensor is a linear block of special ordered numbers over a field $\mathbb{K}$, like a vector v_i or a matrix $M_{i,j}$. A tensor can have an arbitrary number $d \in \mathbb{N}_0$ of indices. We call the number d dimension of the tensor. The zero dimensional tensor is equivalent to a scalar value. Each dimension of the tensor is defined by an index set. Those sets can be any finite or infinite set. For our application we only consider finite sets, in particular only subsets of $\mathbb{N}$. The index of a tensor is therefore the Cartesian product of the index sets of each dimension.

$$I = I_1 \times I_2 \times ... \times I_d \tag{2.1}$$

Notation 1: (Indices) We identify the entries of the tensor by a d-tuple $i = (i_1, i_2, ..., i_d) \in I$. We use the notation

$$v_i = v[i] = v[i_1, i_2, ..., i_d] \tag{2.2}$$

for the entries of the tensor v.

A tensor v is uniquely defined by its entries $v := (v_i)_{i \in I}$. We write

$$\mathbb{K}^I = \{v = (v_i)_{i \in I} : v_i \in \mathbb{K}\} \tag{2.3}$$

for the set of identical shaped tensors.

2.2. Tensor Product

An equivalent definition can be made using the tensor product of vector spaces.
Definition 1: (Tensor Product) Let $v \in \mathbb{K}^I$ and $w \in \mathbb{K}^J$ be two vectors. The tensor product

is defined as

$$v \otimes w := \begin{pmatrix} v_1 w \\ v_2 w \\ \vdots \\ v_n w \end{pmatrix} \tag{2.4}$$

A tensor created by the tensor product is called an elementary tensor. In the special case of $d = 2$ this leads us to a rank 1 matrix. The tensor product of two matrices is also called Kronecker product. Let D be the set of index sets $D = \{I_j : 1 \leq j \leq d\}$. Every tensor v can be written as linear combination of elementary tensors where r is the rank of the tensor.

$$v = \sum_{k=1}^{r} \bigotimes_{I_j \in D} v_j^{(k)} : v_j^{(k)} \in \mathbb{K}^{I_j} \tag{2.5}$$

Definition 2: (Tensor Space) Given a set of vector spaces V_j, the tensor space is defined as

$$V_D = \bigotimes_{I_j \in D} V_j = span \left\{ \bigotimes_{I_j \in D} v_j : v_j \in V_j \right\} \tag{2.6}$$

The tensor space is also a vector space.

Notation 2: For tensor spaces over the common index sets $[n] := \{1, 2, ..., n\} \subset \mathbb{N}$, we omit the brackets.

$$\mathbb{K}^n := \mathbb{K}^{[n]} \tag{2.7}$$

2.3. Contraction

The operation most needed for our application is the tensor contraction. A contraction creates a new tensor from two tensors with at least one identical index set. The contraction is formally defined as

$$\mathcal{C}_J : V_{D_1} \otimes V_{D_2} \to V_{D_1 \setminus \{J\}} \otimes V_{D_2 \setminus \{J\}}$$
$$\mathcal{C}_J \left(\left(\bigotimes_{k \in D_1} v_k \right) \otimes \left(\bigotimes_{k \in D_2} w_k \right) \right) := \langle v_j, w_j \rangle \left(\bigotimes_{k \in D_1 \setminus \{J\}} v_k \right) \otimes \left(\bigotimes_{k \in D_2 \setminus \{J\}} w_k \right) \tag{2.8}$$

Some examples for well known tensor contractions are:

- Scalar product of two vectors. ($v \in \mathbb{R}^n, w \in \mathbb{R}^n, \mathcal{C}_{[n]}(v, w) = \langle v, w \rangle$)
- Matrix-Vector multiplication. ($M \in \mathbb{R}^{m \times n}, v \in \mathbb{R}^n, \mathcal{C}_{[n]}(M, v) = Mv$)
- Matrix-Matrix multiplication. ($M \in \mathbb{R}^{m \times n}, N \in \mathbb{R}^{n \times k}, \mathcal{C}_{[n]}(M, N) = MN$)
- Matrix-Matrix multiplication with transposition. ($M \in \mathbb{R}^{n \times m}, N \in \mathbb{R}^{n \times k}, \mathcal{C}_{[n]}(M, N) = M^T N$)

2.4. Tensor Trains

A Tensor Train is a set of d lower dimensional tensors, that can be used to reconstruct the original tensor. The Tensor Train decomposition is not necessarily unique so it is sometimes referred as Tensor Train representation[9]. A Tensor Train can also be used for an approximation of a high dimensional tensor minimizing the error in some tensor norm. Tensor Trains are known for drastic reduction of the memory requirement for a given tensor.

Definition 3: (Tensor Trains) The Tensor Train space is defined by the tensor space V and a tuple $\rho \in \mathbb{N}^{d-1}$ as

$$\mathbb{T}_\rho(V) := \left\{ v \in V : v = \sum_{\substack{k_i \in K \\ (0 \leq i \leq d)}} \bigotimes_{j=1}^{d} v^{(j)}_{k_{j-1}k_j} \text{with } v^{(j)}_{k_{j-1}k_j} \in V_i \text{ and } \#K_j = \begin{cases} 1 \text{ for } j=0 \text{ or } j=d \\ \rho_j \text{ for } 1 \neq j \neq d-1 \end{cases} \right\} \tag{2.9}$$

A tensor in the Tensor Train format can be represented as

$$v = \sum_{k_1=1}^{\rho_1} \sum_{k_2=1}^{\rho_2} \dots \sum_{k_{d-1}}^{\rho_{d-1}} v^{(1)}_{1,k_1} \otimes v^{(2)}_{k_1k_2} \otimes v^{(3)}_{k_2k_3} \otimes \dots \otimes v^{d-1}_{k_{d-2}k_{d-1}} \otimes v^{(d)}_{k_{d_1},1} \otimes v^{(d)}_{k_{d_1},1}. \tag{2.10}$$

2.5. Visual Representation

For the further use of tensors we use a simple visual representation to explain the algorithms. A tensor will be represented as a rectangle. Every index set of the tensor is shown as line joining the rectangle. Figure 2.1 shows an example for a three dimensional tensor $v \in \mathbb{R}^{m \times n \times k}$. The contraction of two tensors is illustrated by the connection of the two lines for the index we are summing over. Figure 2.2 shows a four dimensional Tensor Train with the corresponding contractions and an equivalent tensor represented by the Tensor Train.

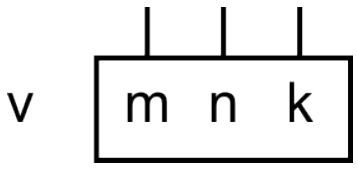


Figure 2.1.: A three dimensional tensor v with the index sets $[m]$, $[n]$ and $[k]$.

2.6. Memory Representation

Since the RAM of a common computer is linear, two or higher dimensional objects have to be structured. In case of matrices there are two major representations for mapping them into the one dimensional linear address space L. While C++ uses Row-Major order the CuBLAS framework uses the Column-Major order. One can easily switch from one representation to each other by transposing the matrix.

$$A^T = (B * C)^T = C^T * B^T \tag{2.11}$$

In case of a tensor with $d > 2$ there are d! representations. Let $I_k = [n_k]$ be the index set of our tensor T. Then our mapping into memory space is

$$f(x_1, ..., x_d) = \sum_{i=1}^{d} x_i \prod_{k=1}^{i-1} n_k \tag{2.12}$$

Note, that the empty product is equal to one. Let $\sigma \in S_d$ be a permutation. Then

$$f_\sigma(x_1, ..., x_d) = \sum_{i=1}^{d} x_{\sigma(i)} \prod_{k=1}^{i-1} n_{\sigma(k)} \tag{2.13}$$

is another mapping of T into L. Using this, a matrix is stored Column-Major-Order using f_{id} and Row-Major-Order using $f_{(21)}$. In our algorithm we use for every tensor f_{id}. By using f_{id} we gain the advantage to compute a tensor contraction with the first or last index set by using ordinary matrix multiplication.

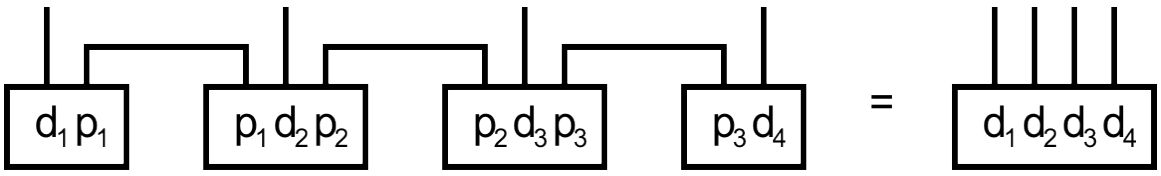


Figure 2.2.: On the left side there is a Tensor Train representing a four dimensional tensor. The indices p_i are connected. Only the d_i indices are free. On the right side there is an equivalent four dimensional tensor, which represents the left Tensor Train after the contractions.

3. CUDA

In the year 1965 Gordon E. Moore predicted, that the number of transistors on a CPU doubles every two years[11]. This has proven true and will continue at least for a few years. The computational power followed a similar trend like the transistors. The increase of computational power was mainly achieved by increasing the clock rate until the year 2000. By using even higher clock rates than about 3 GHz the power consumption rise and the heat development reaches critical level[2]. As an answer to this problem the manufacturer increased the amount of bits processed at each clock. This was done in two ways.

The first one is the *Single Instruction Multiple Data* (SIMD) architecture described by Flynn. Well known examples for SIMD are processor extensions like MMX and SSE.

The other way is the use of multicore CPUs. This means, that the processor consists of multiple physical cores, which can execute some code independently to each other. This technique is called *Multiple Instruction Multiple Data* (MIMD). Common CPUs has up to 4 cores in personal computers and up to 16 cores for industry and high performance computing.

Despite to the CPUs GPUs started earlier to increase the amount of concurrent threads. At the beginning they used specialized shaders. Due to the gain of complexity these shaders got unified and programmable.

As part of the *General Purpose Computation on Graphics Processing Unit* (GPGPU) Nvidia[7], one of the biggest graphics card manufacturer, created CUDA. CUDA was introduced in November 2006[17] and initial released in June 2007[6]. CUDA is restricted to Nvidia GPUs.

Another way for the GPGPU is OpenCL[12]. The OpenCL specification was initially released 2008 by the Krohnos Group. There exists implementations for different systems like AMD and Nvidia GPUs as well as for integrated Graphics Chips from Intel and AMD processors. There also exists some OpenCL driver for homogenous multi-processor clusters.

3.1. Execution Model

Beside to other programing languages CUDA programs can be written in C. The CUDA C language extension is a small set of keywords to use these special features of the GPUs[17]. The most important are the `__global__` declaration specifier to define a kernel and the execution configuration syntax`<<<>>>` for launching it. A kernel is a function which is executed on the GPU. The minimum is two arguments of the execution configuration. Those arguments are either of type `int` or `dim3`. The first argument defines the size of the grid, while the second argument defines the block size. In case of the type `dim3`, the total amount of threads or thread blocks is the product of each component. The size of a block must not exceed the size of 1024. Listing 3.1 shows an example for a kernel call. More

arguments are needed, when using streams.

On the Tesla and Fermi architecture only the host system can invoke a kernel execution. Kepler in revision 3.5 and Maxwell allow kernel calls from the device itself. This is called *dynamic parallelism*.

```
1  // Copies Data from A to B
2  __global__ void copy(double* A, double* B){
3    int inx = threadIdx.x + blockIdx.x * blockDim.x;
4    B[idx] = A[idx];
5  }
6  void main(){
7    ...
8    copy<<<128,1024>>>copy(A, B);
9    ...
10 }
```

Listing 3.1: An example for a kernel call. The kernel copies 1MB of data from A to B.

A kernel is concurrently executed multiple times by a grid of thread blocks on the GPU. The kernel code describes the work of exactly one thread. Each thread has a unique id within a block. This id can be retrieved by the built in variable `threadIdx`. The size of the `threadIdx` index set can be one, two or three dimensional. The maximum number of threads is limited by the hardware, so for larger problems, threads have to be split into thread blocks. Each block has also a unique id within the grid. The id of the block can also be one, two or three dimensional and it can be retrieved by the variable `blockIdx`. There is also a limit of blocks that can be concurrently executed, so the size of a block should not be too small.

The ordering when thread blocks are executed have to be independent. This allows CUDA applications to scale with the number of processors provided by the GPU. However this paradigm is not appropriate for every algorithm. For those problems who allows such algorithms, the use of a GPUs can be a huge improvement.

CUDA follows the Single Instruction Multiple Thread (SIMT) architecture. A thread block is executed in so called warps. Warps are groups of 32 threads. Each thread in a warp has its own registers and one address counter. Also, each thread can branch independently. However, divergent branches are executed in a serial way. Threads who have not taken this path are disabled until all threads are returned to the same path. For efficient code, divergent branches should be minimized.

3.2. Memory

There are a lot of different Memory types that a CUDA application can use. Main differences are the visibility, latency and memory bandwidth.

Registers are the fastest memory. They are directly on the chip and do not have any latency. A register can only be seen by one single thread. Registers are limited in their amount, so they have to be shared by the threads of all running blocks. It is not possible to influence the usage of Registers used by a Thread in normal C code. By using PTX, an Assembler like language, one could directly address registers. A maximum number

of registers per thread can be set as a compiler option. After finishing a thread block the values are released and the registers can be used for the next thread block.

Also on the chip is the Shared Memory. Shared Memory can be accessed by all threads in the same block. Shared Memory has a very low latency and can therefore be critical for most performance improvements. The Shared Memory is also very limited. An array can be defined as a block of Shared Memory by using the declaration `__shared__`. Single valued variables can also be defined as Shared Memory. Shared Memory data is only valid during the lifetime of a thread block.

Global Memory is the largest Memory on the GPU. It is off chip memory and it can be accessed by all blocks of the grid. Compared to Shared Memory Global Memory has a very high latency. Global Memory has the same lifetime as the application. Memory allocated by the host using `cudaMalloc` is automatically Global Memory. Allocating Global Memory on the device can be done by using the keyword `__global__` or using `cudaMalloc` from the `cuda_static` library on the device. Global Memory allows to use pointer arithmetic.

Local Memory is also off chip memory and therefore its latency can be compared with Global Memory. Local Memory can only be accessed by one single thread. Like Shared Memory, it will be released after a thread block finishes. Arrays allocated on the device without a declaration specifier are usually Local Memory. Also the keyword `__local__` can be used. Small arrays of Local Memory that are used frequently are usually kept in the cache.

3.3. Performance

In order to reach the peak performance there are some guidelines and metrics that should be improved. There are different types of performance leaks. The following sections describes an overview of topics that should be considered, when writing high performance applications.

3.3.1. Branches

CUDA follows the Single Instruction Multiple Thread architecture. This means that there are running several threads executing the same code. Each thread can operate on its own data and has its own address counter. They are free to use each data dependent path. But also each thread is executing the same operation at the same time. When a thread within a warp branches differently the other threads get deactivated. This can be described by the following listing and illustrated by Figure 3.1.

```
1 __global__ void kernel(int* out){
2     int idx = threadIdx.x;
3     int result;
4     if(idx == 0){
5         result = foo();
6     } else {
7         result = bar();
8     }
```

```
9       out[idx] = result;
10  }
```

Listing 3.2: A branching kernel.

Different to multithreaded CPU code where the functions `foo()` and `bar()` are called concurrently, on the GPU the code runs sequential. After reading the index only the thread 0 runs the `foo()` code. Then all other threads, except thread 0, execute the `bar()` code. Finally the paths converges and the result will be written by all threads. Dependent on the workload of the functions `foo()` and `bar()` the execution time can be heavily delayed by single thread execution. For the correctness of the code, branching can be ignored, but for performance divergent branches should be held to a minimum.

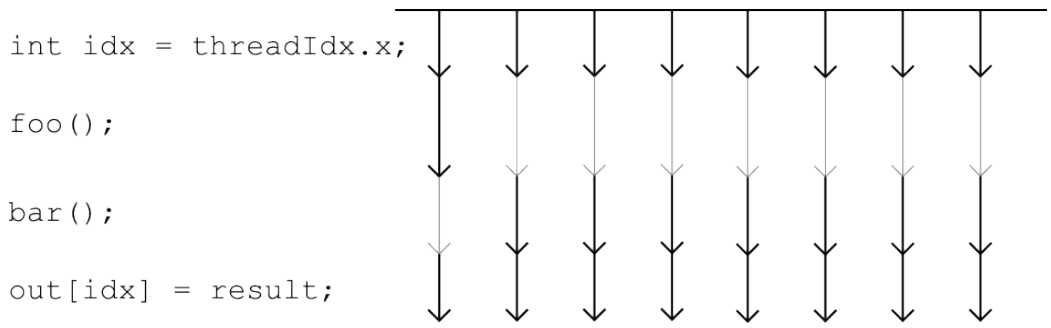


Figure 3.1.: The execution flow of a branching code, with warp size 8. Black arrows are active threads, and the grey ones are disabled.

3.3.2. Occupancy

Occupancy describes the ratio between active warps to the theoretical maximum number of active warps. Some applications can have lower theoretical maximum active warps, than 100%. The resources on the hardware are limited, so they have to be shared between the blocks. In general, there are three reasons not to reach full occupancy. First of all, every Streaming Multiprocessor (SM) has a limited amount of registers. Those registers have to be shared between all threads in a block and SM. The number of registers per thread is limited to a maximum of 63 on Fermi and 255 on Kepler and Maxwell. The maximum number of used registers in a specific application can be set by the compiler option `--maxrregcount`. Table 3.1 shows some differences of the occupancy between the different architectures. Fermi has a total of 32K registers and a maximum of 1536 threads per SM. The Kepler and Maxwell architecture increased the number of registers up to 64K compared to a maximum number of threads of 2048. Detailed information about architecture dependent differences is explained in section 3.4.

$$\text{Occupancy} = \frac{\text{Registers per SM}}{\text{Registers per Thread} * \text{Threads per SM}} \tag{3.1}$$

Another limiter is the Shared Memory. Shared Memory is a very fast memory on the chip that can be used by all threads within a block. On Fermi and Kepler the size of the on

Registers per Thread	Fermi	Kepler & Maxwell
20	1.	1.
63	0.333	0.5
255	—	0.125

Table 3.1.: The theoretical maximum occupancy using different numbers of registers per thread. Note that there is one implicit register, that can't be used by the programmer. The values are truncated by 3 digits.

chip Memory is 64KB used for Shared Memory and L1 Cache. There are two configurations for the on chip memory. By default the memory is divided into 48KB Shared Memory and 16KB L1 Cache. For applications not using much Shared Memory there is a mode with 16KB Shared Memory and 48KB L1 Cache. The Kepler Architecture introduced a third configuration, dividing the Memory by a ratio of 1:1. On Maxwell there is a dedicated 64KB Shared Memory. On all three architectures there is a limit of 48KB Shared Memory per Block.

There is also a maximum number of blocks that can be handled by one SM. Using small block sizes can also limit the occupancy. On Fermi only 8 blocks can run per SM, Kepler can handle 16 blocks and Maxwell 32. For example a kernel on Fermi uses a block size of 64 threads, the Streaming Multiprocessor work with 512 threads compared to a maximum of 1536. This results in an occupancy of 0.33. Using the same application on a Maxwell card 2048 threads can be used and we can reach an occupancy of 1. For full occupancy the block size should be at least 192 on Fermi, 128 on Kepler and 64 on Maxwell. The maximum block size for all three architectures is 1024. One should also consider the dependence of number of active warps and block size. On all three Architectures the warp size is 32. On block sizes multiple to 32 the threads are equally divided into the warps, otherwise some threads remain unused.

$$\#\text{Active Warps} = \left\lceil \frac{\text{Block Size}}{\text{Warp Size}} \right\rceil \tag{3.2}$$

Even so that large block sizes are desirable, on Fermi GPUs the block size of 1024 will reduce the maximum theoretical occupancy to 0.66. This is caused by the fact, that the threads of a block cannot be split into smaller blocks running on different streaming processors or at different time slots. The second block on the processor will need 1024 threads, but there are only 512 remaining. The Kepler architecture increased the maximum thread size per Streaming Multiprocessor to 2048. Since then a full occupancy can be achieved with block sizes of 1024.

3.3.3. Memory Access

Introduced for the Fermi Architecture, CUDA Applications uses a cache hierarchy known from CPUs. Figure 3.2 shows a schematic plan of the cache hierarchy of CUDA GPUs. Caches are small but fast memories directly on the chip. The latency of a L1 Cache is typically 3 to 10 cycles compared to over 200 and up to 800 cycles for off chip memory[17]. While the size of the L1 Cache is fixed by the architecture and reaches up to 64KB per SM,

the L2 Cache varies on the specific product and reaches values from 128KB up to 2MB, strongly dependent on the number of Streaming Multiprocessors.

A cache stores copies of the global memory and provides lower latencies. It loads the data block wise from the next memory in the hierarchy. This means, when one core requests a single datum, also the nearby memory space will be copied into the cache. So when the processor requests a larger memory area in ascending order, only the first request has high latency. Also when a small amount of memory is frequently used, it can be kept in registers or L1 cache, which will also lead to a fast memory access.

On the other hand when a program accesses the memory in a random way or with large step sizes, every read and write has a large latency even if the prefetched data will be used later on. This results in two drawbacks. Both values will be load multiple times and for each request the processor has to wait. Considering this, the L1 and L2 cache overhead is a good indicator for improvements.

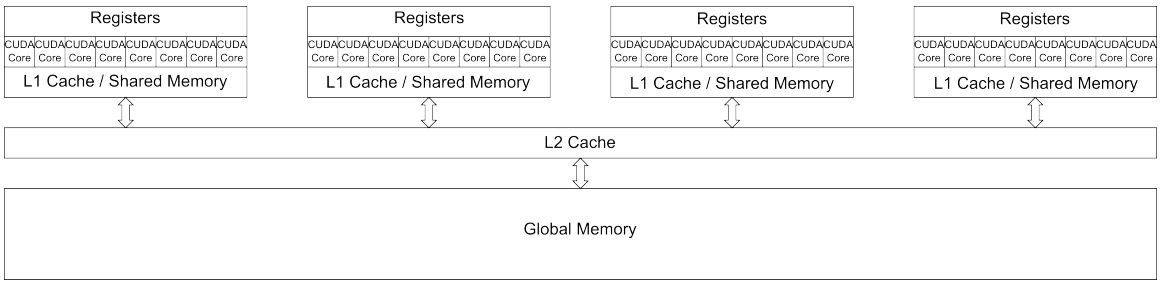


Figure 3.2.: The schematic cache hierarchy of a CUDA GPU with 4 Streaming Multiprocessors and 8 CUDA Cores each.

3.3.4. Instruction Level Parallelism

On current Nvidia GPUs up to 64 different warps can be active per Streaming Processor. Instructions are broadcasted to the specific cores by a warp scheduler. *Instruction Level Parallelism* (ILP) describes the way, the code can be parallelized within a single thread. Little dependencies within the code helps the warp scheduler to take an efficient ordering for execution.

There are only a few warps active at a time compared to the warps that are managed on the hardware. When a warp cannot execute the next instruction, the scheduler tries to find another warp that can be executed. If no warp can be executed, the processor runs idle. A common reason for this is memory dependence. Examples for ILP are shown in Listing 3.3. Even if the load operation of `a` has not finished, the arithmetic operations can be executed. Also the multiplication has not to wait for the outcome of the addition since the multiplication do not depend on the value of `b`.

```
1 a = A[1];
2 b = b1 + b2;
3 c = b1 * c1;
```

Listing 3.3: An example for Instruction Level Parallelism.

The Fermi Architecture has two warp scheduler. Kepler and Maxwell has four. The Fermi architecture (Compute Capability 2.1) has 48 CUDA cores and broadcasts instructions in packages of 16, so it can handle 3 warps at a time, but has only two warp schedulers. Therefore a warp should always be able to compute two instructions without dependence. Otherwise the warp will stall. On Kepler and Maxwell every warp has its own warp scheduler.

3.4. Architecture Differences

Table 3.2 shows some architecture dependent technical differences of Nvidia GPUs. During CUDA development a lot of internal features have been improved, but most paradigms for the programmer stayed the same. For example a streaming processor can now handle 2048 threads at a time, but the maximum block size stayed at 1024. This results in a 100% theoretical occupancy for block sizes of 1024 compared to 66% of Fermi. Another example is the use of Shared Memory. Maxwell has 64KB dedicated Shared Memory. The maximum amount of Shared Memory per Block is 48KB for all three architectures.

Name	Fermi		Kepler		Maxwell
Compute Capability	2.0	2.1	3.0	3.5	5.0
Single Precision Operation per Clock and SM	32	48	192		128
Double Precision Operation per Clock and SM	4/16[1]	4	8	8/64[2]	1[3]
Load & Store Units / SM	16		32		
Maximum Number of Threads per SM	1536		2048		
Maximum Number of Registers per Thread	63		255		
Maximum Number of Threads per Block	1024				
Active Thread Blocks per SM	8		16		32
Maximum Warps per Multiprocessor	48		64		
Registers / SM	32 K		64 K		
Level 1 Cache	16/48 KB		16/32/48 KB		64 KB
Shared Memory / SM	16/48 KB		16/32/48 KB		64 KB
Warp Size	32				

[1] 4 for GeForce and 16 for Tesla GPUs
[2] 8 for GeForce and 64 for Tesla GPUs
[3] According to the CUDA C Programming Guide. We measured a higher rate of 4 operations per clock per SM on GeForce GTX 750 Ti.

Table 3.2.: Architecture specific technical data[13] [14] [15] [17].

3.5. Profiler

For analyzing the written code, Nvidia provides a toolkit called CUDA Profiler. The Profiler can provide different data of the Kernels running on the GPU. Those data blocks are called Metric. The capability which Metric can be tested are dependent on the architecture. The Profiler isn't a tool for debugging.

The profiler comes in three different types. The first one is *nvprof*. It prints the result

in plain text using the console. The second one is the *Visual Profiler* that provides a GUI based on Eclipse or Visual Studio. The last one is the *Command Line Profiler*. It is based on Environment Variables. In some literature nvprof is usually called Command Line Profiler and the Profiler based on Environment Variables is being ignored. The Profiler supports some features like remote profiling or MPI profiling. For further information see [18].

The program nvprof can be started with the command `nvprof`. By default it writes its results direct into the console. The output of nvprof will be a text based table showing all requested data. It also supports the comma separated output like the `csv` format. This can be used to import the data into a spreadsheet program. Another possibility is to export the profiling results into a non-human readable file that can afterwards be imported into the visual profiler using the flag `--output-profile [file-name]`.

The nvprof profiler runs with the following configuration:

```
nvprof [options] [cuda-application] [application-arguments]
```

The profiler can run in various modes. The *summary Mode* is the default one and shows statistics about all kernel calls like their amount and runtime. It also shows a summary about all memory copies. The *GPU-Trace Mode* shows a list of each kernel call or memory copy individually. It provides more detailed data about the kernel calls like grid and block size or used registers that could not be displayed in summary mode. The *API-Trace Mode* lists similar to the GPU-Trace all API-Calls in chronological order. The GPU-Trace mode can be enabled by the flag `--print-gpu-trace` and `--print-api-trace` for the API-Trace mode.

More detailed information can be retrieved by the *Event/Metric Mode*. Those are also available in Summary and in Trace Mode. While Summary Mode is the default one, Trace Mode can be enabled by the `--aggregate-mode off` flag. The events can be chosen by the `--events [event]` flag and for the metrics `--metrics [metric]`. It is possible to collect data of multiple events and metrics in one run. For multiple metrics and events one can write those as coma separated flag. For all metrics or events there is the option to write `all`. A list of available metrics and events, that can be profiled with nvprof and the Visual Profiler can be found in [18].

The Visual Profiler is a tool that visualize the collected data from profiling. For starting the Visual Profiler the command `nvvp` can be used. It is able to create a new session and collect the required data on the fly or to import a session created by the Command Line Profiler and nvprof. The Visual Profiler is able to collect the same data as nvprof, but in addition it visualize the data using different charts for a possible faster analysis of the data.

3.6. CuBLAS

CuBLAS is an implementation for CUDA GPUs of the Basic Linear Algebra Subprograms (BLAS)[16]. BLAS is a quasi-standard interface providing efficient implementations for some operations frequently used in linear algebra. CuBLAS uses the advantage of multi-threading to improve performance.

There exists other implementations for BLAS routines on CUDA GPUs. For example MAGMA is a library providing various algorithms used in linear algebra for CUDA, OpenCL and Intel Xeon Phi [1].

BLAS is divided into three levels.

- Level 1: $\mathcal{O}(n)$ Memory for $\mathcal{O}(n)$ Operations
- Level 2: $\mathcal{O}(n^2)$ Memory for $\mathcal{O}(n^2)$ Operations
- Level 3: $\mathcal{O}(n^2)$ Memory for $\mathcal{O}(n^3)$ Operations

Examples for Level 1 BLAS routines are vector addition or the dot product. A matrix vector multiplication belongs to Level 2 BLAS and Level 3 BLAS includes operations like matrix-matrix multiplication and the solving of a linear Equation $Ax = b$.

For this project Level 3 BLAS routine for the matrix-matrix multiplication is of special interest.

$$C \leftarrow \alpha \cdot op_A(A) \cdot op_B(B) + \beta \cdot C. \tag{3.3}$$

It has the function name XGEMM, where the first letter have to be replaced by S, D, C or Z. S and D are real valued single and double precision numbers, C and Z are used for complex single and double precision values.

```
void dgemm(char transa, char transb, int m, int n, int k,
    double alpha, double* a, int lda, double* b, int ldb,
    double beta, double* c, int ldc)
```

Listing 3.4: Standard declaration of a BLAS matrix-matrix multiplication using real valued double precision floats.

Listing 3.4 shows the BLAS interface for the equation 3.3, where `a` is the pointer to matrix A, `b` is the pointer to matrix B and `c` is the pointer to matrix C. The variables `alpha` and `beta` define α and β where typical values are $\alpha = 1$ and $\beta = 0$. The values of `transa` and `transb` should be either "n", "t" or "c" defining the operation of op_A and op_B. Those operation can be the identity function for "n", the transposition for "t" or the hermitian transposition for "c". In case of real valued matrices "t" and "c" are the same. `m`, `n` and `k` are defining the size of the sub matrices $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$ and $C \in \mathbb{R}^{m \times n}$ that should be assembled. For the full matrix multiplication lda is equal to m, ldb is equal to k and ldc is equal to m. A schematic multiplication is shown by figure 3.3.

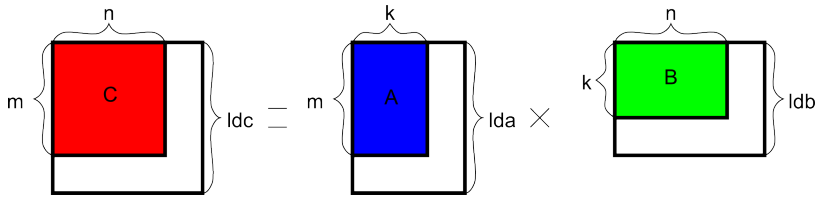


Figure 3.3.: A matrix multiplication of two sub matrices. Only the red part of C is computed with the blue and green data of A and B.

Part II.

Algorithms

4. Non Cyclic Tensor Trains

The first of the two problems we are going to analyze is the contraction of non-cyclic Tensor Trains. Given two Tensor Trains $v \in \mathbb{T}_\rho(V)$ and $u \in \mathbb{T}_\rho(V)$ we want to compute $s = \sum_{i \in I}(v[i] \cdot u[i])$. For most tensors it is not feasible to iterate over the index set, as $\#I = n^d$ can be very large for high dimensions d. By using the advantage of the Tensor Train representation we can reduce the needed operations to an order of $\mathcal{O}(d \cdot n \cdot \rho^3)$. This can be reached by first contracting over the dimension index and afterwards over the index to the next subtensors of u and v[10]. The basic principle is shown by figure 4.1.

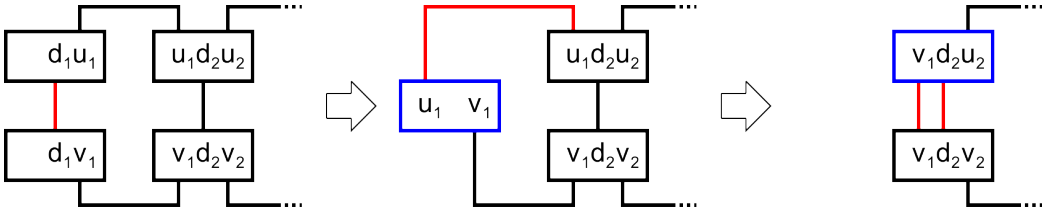


Figure 4.1.: An efficient contraction of two Tensor Trains. The red line indicates over which index set is contracted. The blue Tensors are temporary results over the contraction of the previous step.

The computational cost of a tensor contraction is similar to the matrix-matrix multiplication. For each index of the resulting tensor we have n multiplications and n additions, where n is the size of the Index Set for the contraction. Considering the second contraction, we have the size of $n\rho^2$ for the resulting tensor and ρ for the size of the index set we are sum over. This results in a total amount of $2n\rho^3$ operations. For the third contraction, we have a resulting tensor of size ρ^2 and a size of $n\rho$ for the Index set for contraction. The very first contraction and the last two are cheaper since we have smaller tensors as input or output. Therefore we need at most $4n\rho^3$ operations per dimension. So we have an upper bound of $4dn\rho^3$ operations that are required in order to compute the result. The exact number of operations we have to compute using this way is $2n\rho(1 + 2\rho + 2(d-2)\rho)$.

The minimum additional memory, which is needed for the computation is in the order of $(n+1)\rho^2$. We need two tensors for storage of the temporary values. The first one of size of $\rho \times \rho$ for the contraction over the dimension and the second one tensor of size $\rho \times n \times \rho$ for the contraction over ρ. The memory can be reused, since we don't need those results for later computations.

4.1. CPU Implementation

For checking the results of the GPU implementation, we also wrote an algorithm for the CPU. Please note that this code is just for verifying the result of the GPU code. It is not opti-

mized and therefore should not be used as comparison of efficiency. The basic contraction over one index set is defined as

$$t[i_u, i_v] = \sum_{i \in I} u[i, i_u] \cdot v[i, i_v]. \tag{4.1}$$

Using this, we can easily implement a loop based tensor contraction. Since the code is semantic equivalent to a matrix-matrix multiplication with implicit transposition of the first matrix, we could also use an existing implementation like BLAS.

The second part of the application is to iterate over all subtensors of the Tensor Train. We assume all tensors to have the same size $n \times 2 \times n$, except the tensors at the border $2 \times n$ on the left and $n \times 2$ on the right. Listing 4.1 shows an implementation for the Tensor Train contraction. This outer loop is the same for all non-cyclic Tensor Train contractions. Using the contraction over the first index and the correct ordering of the result allows us to use the same procedure for all calls. Therefore we have only one function that has to be optimized in order to achieve good performance. In line 4 and 7 we sum over two indices. This is basically the same as using I as $I = [2] \times [n]$.

```
1 temp = contractTensor(TT1[0], TT2[0], n);
2 for(int i = 1; i < d-1; i++){
3   temp = contractTensor(temp, TT1[i], rho);
4   temp = contractTensor(temp, TT2[i] , rho*n);
5 }
6 temp = contractTensor(temp, TT1[d-1], rho);
7 temp = contractTensor(temp, TT2[d-1], rho*n);
```

Listing 4.1: Loop over all subtensors of the Tensor Trains `TT1` and `TT2`. The first contraction and the contractions in the loop are illustrated in figure 4.1

4.2. GPU Implementation

In the following we present three GPU implementations for the `contractTensorTrain` algorithm. The first one is a simple handwritten code. It is a beginner level implementation providing very high occupancy. However it suffers from high overhead of integer operations and does not consider advantages of cache hierarchy. The second one is based on the `sgemm` implementation from [21]. It speeds up the execution by evaluating sub matrices using shared memory. It gives a hint, how efficient BLAS routines can look like. The third code is based on CuBLAS, the BLAS implementation from Nvida. It only uses the standard interface and provides good results. It can benefit from the latest improvements of drivers or hardware by changing little or none of the own code.

We restrict ourselves to Tensor Trains that are small enough to fit into the global memory. For example we are able to compute Tensor Trains of size $d = 100$ with $\rho = 1024, n = 2$ on a 2GB RAM.

The basic algorithm for contracting the Tensor Trains on the GPU is the same for all three implementation (see listing 4.1). The only difference is the algorithm for one tensor contraction. In this section we will further concentrate on the algorithms for the matrix-matrix multiplication. Only for the CuBLAS implementation we need support for an additional

handle `cublasHandle_t`. The main difference to the CPU version is that we have to allocate the temporary tensors on the device. Also the data of the Tensor Trains have to be on the global memory of the GPU. Therefore `TT1` and `TT2` are arrays of device pointers. For the final result we also have to copy back a single value from the device to the host. We call the method for contracting one pair of tensors `dgemmTN` since the implementations are equivalent to a matrix-matrix multiplication with transposing the first argument in a column major matrix representation with double precision floating point values.

4.2.1. Implementation 1: Simple Loop

The first implementation is based on the straight forward implementation of a matrix-matrix multiplication as described in [17]. Every thread is computing the result for one entry of the resulting tensor using one simple loop. The thread block size should be chosen as large as possible to obtain high occupancy. For supporting all sizes of matrices one could use an array bound check within the kernel.

Those simple kernels produce a lot of overhead computing index offsets compared to the workload. Also every thread loads the needed values from the input matrices on their own. The code suffers from a lot of slow memory calls and it only use the advantage of cache by chance.

The code does not use shared memory and less than 10 registers per thread, so the only limitation for occupancy is the block size.

4.2.2. Implementation 2: Using Shared Memory

The code shown in listing 4.2 shows some improvements compared to the algorithm explained in section 4.2.1. It shows some techniques, how the performance of a kernel can be enhanced. The code is based on the `sgemm` implementation from [21] and uses ideas described in [20]. The achieved throughput for large matrices of single precision floating point numbers is still worse compared to the CuBLAS implementation, which is shown in section 6.3. This is caused by the fact that a good BLAS library has many kernels optimized for different matrix sizes and hardware capabilities.

The code computes a sub matrix of the size of 32×32, but the block size is 64. So every thread have to compute 16 values. Also the matrices `A` and `B` are loaded in blocks of size 32×32 into shared memory. The array `sum` is used for computing the values of `C` and it is stored on local memory, but profiling yield that the array is kept in L1 cache. We also adjust the pointers of all matrices to the first used offset to reduce overhead caused by integer arithmetic.

The outer loop is divided into two parts. The first part loads a matrix block from `A` into shared memory. Shared memory is visible to all threads of one block. Every thread have to copy 16 values. Reminding, that the thread block is executed in warps, we can profit from the cache hierarchy ordering the load commands to be coalescent. The `__syncthreads()` call is important, because we have to ensure, that the matrix `As` has been entirely copied from global memory. Afterwards we multiply the sub matrices and add them to the resulting sub matrix. The values of `B` are loaded on the fly when needed. The same value of `B` is used by every thread within a warp in each step of the for loop. After evaluating the matrix product, we have again to ensure that all warps have finished their work. The

thread synchronization prevents the data of `As` to be overwritten from a faster warp. The last part consists of writing the computed results back to the global memory.

```
1  __global__ void dgemmNTKernel(...){
2          ...
3          __shared__ double As[32][32];
4          for (int i = 0; i < sizeA1; i += 32){
5                  for (int k = 0; k < 16; k++){
6                          As[idx][idy + k] = A[(k + idy)* sizeA1];
7                  }
8                  __syncthreads();
9
10                 for (int k = 0; k < 32; k++){
11                         double a = As[k][idx];
12                         for (int j = 0; j < 16; j++){
13                                 sum[j] += a * B[sizeA1 * j];
14                         }
15                         B++;
16                 }
17                 A += 32;
18                 __syncthreads();
19         }
20
21         for (int j = 0; j < 16; j++)
22                 C[j * sizeA2] = sum[j];
23 }
```

Listing 4.2: The main loop of the tensor contraction. The loop loads the first tensor block wise into shared memory and afterwards computes the result.

4.2.3. Implementation 3: Using CuBLAS

The last GPU implementation for non-cyclic Tensor Trains is based on the CuBLAS library provided by Nvidias CUDA Toolkit. CuBLAS does not restrict the size of the matrices that can be computed, but matrices with sizes of the power of two have normally the best performance. Since we prepared the data-structure such that we only have to sum over the first or the first two index sets, we can use the same BLAS routine for every contraction. CuBLAS uses the column major representation of matrices, so we have to transpose the matrix representing the tensor of the first argument. After the multiplication, the result will have the right structure for further computations. This is the only implementation of `DgemmTN` where we have to provide the `cublasHandle_t`. Nvidia added this handle in Version 2 of CuBLAS for a better support of multiple threads on the host and multiple GPUs.

5. Cyclic Tensor Trains

The second kind of problem we are going to analyze is the cyclic Tensor Trains. The difference to non-cyclic Tensor Trains is, that we have also three dimensional tensors at the border and the additional index sets are connected to each other. The schematic structure of the inner product of two cyclic Tensor Trains is shown in figure 5.1.

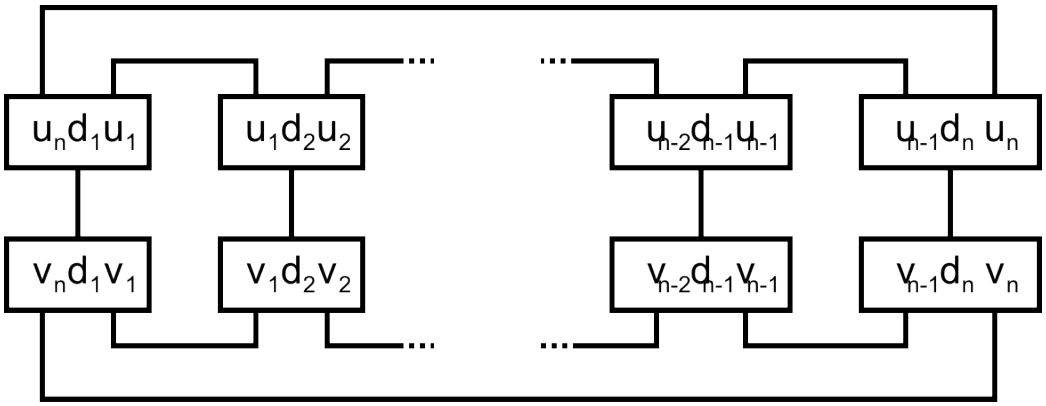


Figure 5.1.: The connections of two cyclic Tensor Trains.

The contraction of cyclic Tensor Trains needs operations in an order of $\mathcal{O}(d \cdot n \cdot \rho^5)$. By contracting two sub-tensors using the same algorithm as for non-cyclic Tensor Trains, we get intermediate tensors in the size up to $\mathcal{O}(n \cdot \rho^4)$. For this kind of algorithm the major problem is the huge size of the temporary tensors. By restricting the problem size to fit into the memory we are only able to compute for example a Tensor Train with sub tensors of the size $n \times 2 \times n$ with 2 GB global memory.

Because of the two additional index sets of the intermediate tensors we cannot implicit profit from BLAS routines. Using the matrix product for tensors requires, that the index sets we want to sum over are at the first or last position. So we need an additional permutation step in between two tensor contractions.

For cyclic Tensor Trains, we provide two different implementations for the GPU. There is also a CPU implementation for cyclic Tensor Trains that can be used to verify the GPU results.

The first GPU algorithm solves the problem, using implicit permutation. It contains four different kernels, where only one is used within the outer loop doing the major work. So the performance of this implementation strongly depends on this single kernel. Since it is handwritten and does not provide optimized code for different tensor sizes, the size of each ρ index set is restricted to multiples of 32. The second algorithm follows the principle to evaluate smaller tensor blocks rather than the whole tensor. It is based on the same technique for large matrices divided into smaller matrix-blocks. This allows us to reduce

the required memory overhead from $n \cdot \rho^4$ to $n \cdot \rho^2$.

5.1. Implementation 1: Implicit Permutation

We want to preserve the basic structure of algorithm 4.1. The main difference is that we now need four separate kernels. By using special permutations in the first and last two kernels we can provide a single kernel for both contractions within the loop. The starting and finalizing kernels have a workload of at most $\mathcal{O}(n \cdot \rho^4)$ floating point operations and the kernel within the loop has an order of $\mathcal{O}(n \cdot \rho^5)$ operations. So we have only one kernel, which is doing the major work. Every optimization of this kernel has a huge impact to the overall performance of the algorithm. We can also see in figure 5.3 that we are working with tensors with up to five dimensions. The memory requirement is $\mathcal{O}(n \cdot \rho^4)$. The requirement, that every tensor have to fit into the global memory reduces the size of the Tensor Trains on a 2GB memory to $\rho \leq 64$ for $n = 2$.

The first contraction is illustrated in figure 5.2. Special to the non-cyclic version is, that the index set we are going to sum over is in the middle. So we cannot implicit use the matrix-matrix multiplication. Also the resulting tensor provides a new ordering of the indices.

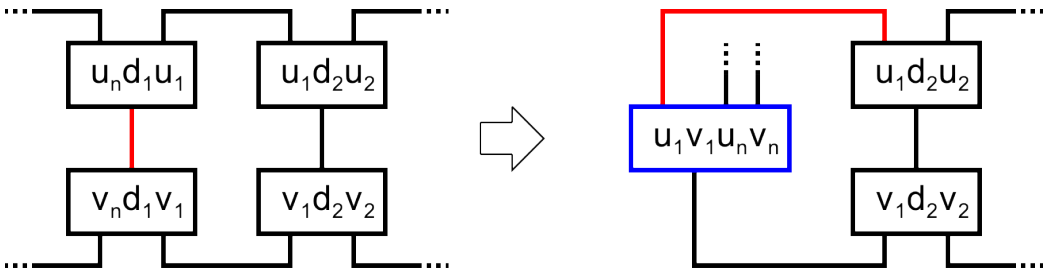


Figure 5.2.: The first contraction of a cyclic Tensor Train. The Index set where we contract over is highlighted in red. The resulting intermediate tensor is blue.

Shifting the first indices from the input tensors to the back allows us to benefit from coalesce memory. Also, those indices are only needed in the last two kernels.

Figure 5.3 shows the principle tensor contraction within the loop. The permutation used in the algorithm can be used by both calls. However they use different combinations of the indices. Those are highlighted in green. The code for the kernel `contractTensorPerm` is very similar to the implementation of the `DgemmNT`. We only need an additional `z` index of the thread block for the permutation that is used for the index shifts. This produces less overhead than an additional permutation step.

The last two steps are illustrated in figure 5.4. We can see that we have to sum over two not neighboring index sets, so it cannot implicit be done by a BLAS routine. The last kernel `contractTensorFin2` is equivalent to a standard scalar product, therefore every dot product implementation can be used.

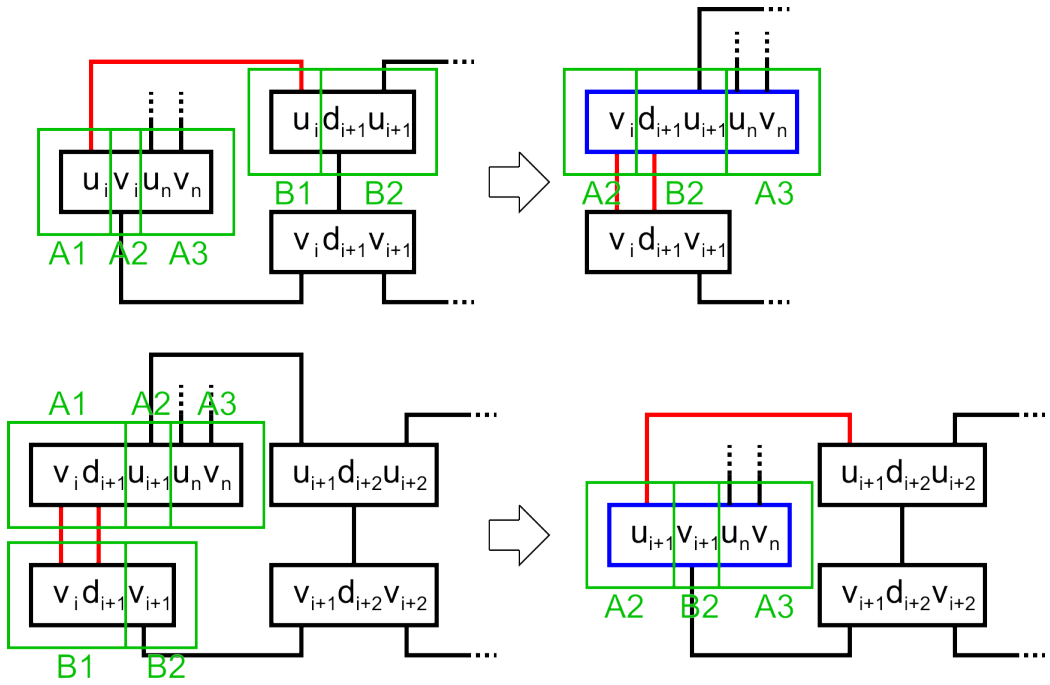


Figure 5.3.: The main contraction of a cyclic Tensor Train. The Index set where we contract over is highlighted in red. The resulting intermediate tensor is blue. The similarities of the permutation are highlighted in green.

5.2. Implementation 2: Reduced Memory Requirement

In order to avoid the huge size of the temporary tensors and therefore allow bigger cyclic Tensor Trains, we modify our algorithm such that we can use non cyclic contraction routines. This can easily be achieved by using basic properties of $\mathbb{R}$. Given two cyclic Tensor Trains $u \in \mathbb{T}_\rho(V)$ and $v \in \mathbb{T}_\sigma(V)$, $\rho, \sigma \in \mathbb{N}^d$ our goal is to compute the scalar product of both tensors.

$$\sum_{j_1=1}^{\rho_1} \dots \sum_{j_d}^{\rho_d} \sum_{k_1=1}^{\sigma_1} \dots \sum_{k_d}^{\sigma_d} \sum_{i_1 \in I_1} \dots \sum_{i_d \in I_d} u^{(1)}_{\rho_d, \imath_1, \rho_1} \cdot u^{(2)}_{\rho_1, \imath_2, \rho_2} \cdot \dots \cdot u^{(d)}_{\rho_{d-1}, i_d, \rho_d} v^{(1)}_{\sigma_d, \imath_1, \sigma_1} \cdot v^{(2)}_{\sigma_1, \imath_2, \sigma_2} \cdot \dots \cdot v^{(d)}_{\sigma_{d-1}, i_d, \sigma_d} \tag{5.1}$$

Since we only have finite sums of finite values we can switch sums. Moving the sums over ρ_d and σ_d outwards, we get an equivalent of the sum over $\rho_d \cdot \sigma_d$ non-cyclic Tensor Trains.

$$\sum_{j_d}^{\rho_d} \sum_{j_d}^{\sigma_d} \left(\sum_{j_1=1}^{\rho_1} \dots \sum_{j_{d-1}}^{\rho_{d-1}} \sum_{k_1=1}^{\sigma_1} \dots \sum_{k_{d-1}}^{\sigma_{d-1}} \sum_{i_1 \in I_1} \dots \sum_{i_d \in I_d} u^{(1)}_{\rho_d, \imath_1, \rho_1} \cdot \dots \cdot u^{(d)}_{\rho_{d-1}, i_d, \rho_d} v^{(1)}_{\sigma_d, \imath_1, \sigma_1} \cdot \dots \cdot v^{(d)}_{\sigma_{d-1}, i_d, \sigma_d} \right) \tag{5.2}$$

For this version we can use an efficient implementation of the non-cyclic Tensor Train implementation with an additional reduction afterwards. Also we do not need much more

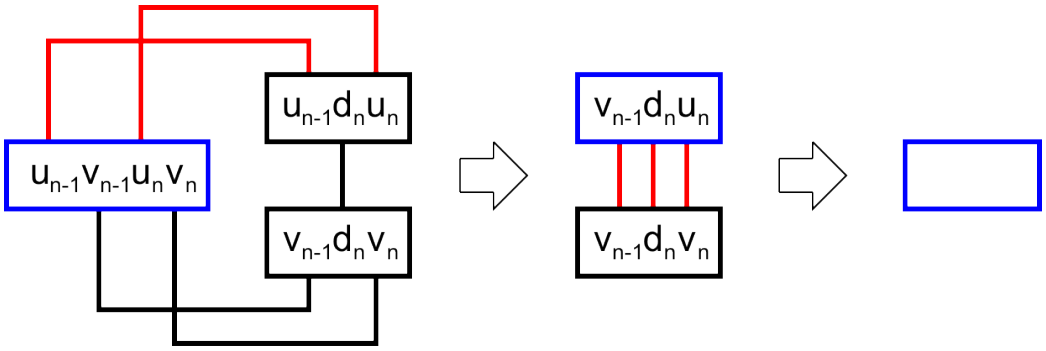


Figure 5.4.: The last two contraction steps done by the algorithm. The last tensor, without indices is a scalar value.

temporary memory as for the non-cyclic Tensor Train implementation. The algorithm is also well suited for the use of multiple GPUs.

In order to use the non-cyclic Tensor Train contraction we have to provide two dimensional tensors for $u^{(1)}$ and $u^{(d)}$ instead of the given three dimensional tensors. For both, the first and the last tensor, we need different methods for creating the two dimensional tensor, because of the structure of their indices.

Part III.

Results and Conclusion

6. Results

The following chapter shows some experimental results of the described algorithms.

6.1. Restrictions and Limitations

As in many cases our code cannot handle every contraction of Tensor Trains. So our implementation provides some restrictions and limitations. First, we restrict ourselves to tensors that fit into the GPUs RAM. Tensors that does not fit into the memory provide some disadvantages. The code gets more complex and worse understandable. It produces a lot of memory transfer overhead. Communication hiding gets more important than an efficient implementation. However it is possible to compute Tensor Trains with $\rho = 2048$ and $d = 50$ or $\rho = 1024$ and $d = 250$ on a 2GB RAM, which should be sufficient for most applications.

Second, we look only at problems that are big enough where we can expect to see an advantage compared to the CPU. Very small tensors can also be computed efficient on CPUs, and we do not have the overhead of starting the Kernels and transfer data to the GPU.

Third, for some implementations we restrict ρ to be a multiple to 32. Tensor sizes multiple to 32 are fitting very well into CUDA architectures. If there are other values needed, one could use the next higher multiple of 32 and fill the additional space with zeros. Also, there is for every algorithm at least one implementation that allows any tensor sizes.

6.2. Used Hardware

As test environment the following hardware was used:

- 2 Tesla M2090 with Intel SandyBridge-EP Xeon E5-2670 Host System
- GeForce GTX 645 witch Intel Core i7-4770 Host System
- GeForce GTX 750 Ti with Intel Core i7-870 Host System

Table 6.2 shows some technical data about the GPUs. For our application the main difference between Tesla and GeForce GPUs is, that the Tesla cards have more double precision floating point cores. This is an important performance factor since the result can easily exceed the range of single precision floating point numbers ($> 3.4 \cdot 10^{38}$) which will result in a non-meaningful `inf`. Tesla GPUs are considered to be used in High Performance Computing where GeForce GPUs are targeted for gaming purposes.

Name	Tesla M2090	GeForce GTX 645	GeForce GTX 750 Ti
Architecture	Fermi	Kepler	Maxwell
Compute Capability	2.0	3.0	5.0
Streaming Multiprocessor	16	3	5
CUDA Cores	512	576	640
GPU Clock Rate	1301 MHz	824 MHz	1280 MHz
Theoretical Peak Performance	1332MFLOP/s	949 MFLOP/s	1638 MFLOP/s
L2 Cache Size	768KB	256KB	2MB
Global Memory	5375MB	1023MB	2047MB

Table 6.1.: Technical data of the used Hardware. The hardware dependent data was read with the sample program *devicequerry* provided in the CUDA Toolkit [5]. The theoretical peak performance was determined by the formula CUDA Cores * Clock Rate * 2. The factor 2 came from the fused multiply-add operation.

6.3. Experimental Results

For testing the algorithms we assume that we already have the Tensor Trains for contraction. They are generated on the CPU using random numbers in the range of $[-1; 1]$. The runtime or performance of creating the Tensor Trains will not be part of the analysis. Also we assume that ρ is constant within a particular Tensor Train. This simplifies the analysis and is practical for many applications. We also assume $n = 2$ for most settings. Such configuration is often used in Quantum Tensor Networks. Higher values for n for other applications will usually result in a better performance due to the higher operations memory ratio.

All results represent the median of 3 from individual runs.

6.3.1. Comparison of a Single Contraction

First we want to compare the Kernels for a single tensor contraction used by all algorithms. For this we compute the matrix multiplication $C = A^T \cdot B$ with $A, B \in \mathbb{R}^{n\rho\times\rho}, C \in \mathbb{R}^{\rho\times\rho}$ for different ρ and $n = 2$. It reflects the data sizes of the tensors in one of the contractions in the algorithms. There are more different sizes to consider, but it gives insights about the expected performance. Figure 6.2 shows the performance of all three Algorithms explained in section 4.2 for both single and double precision floating point operations. We can clearly see, that the simple loop reach the lowest operations per second rate. For the 32Bit floating point values we can see that the highest performance is only 1.4% of the theoretical peak on Fermi and for bigger matrices the values are decreasing. The code on the Kepler card reaches only 0.25% of the theoretical peak performance. This might be caused by the small L2 Cache since the simple loop algorithm highly depend on direct global memory accesses. In comparison the 64Bit floating point version reaches almost the same absolute values. Therefore we can clearly see, that the algorithm is limited by the memory access and we haven't reached the maximum performance.

By assembling blocks of sizes of 32×32 and using shared memory for fast load instructions we can get an improvement of up to 20 times faster on the Kepler card and about 10

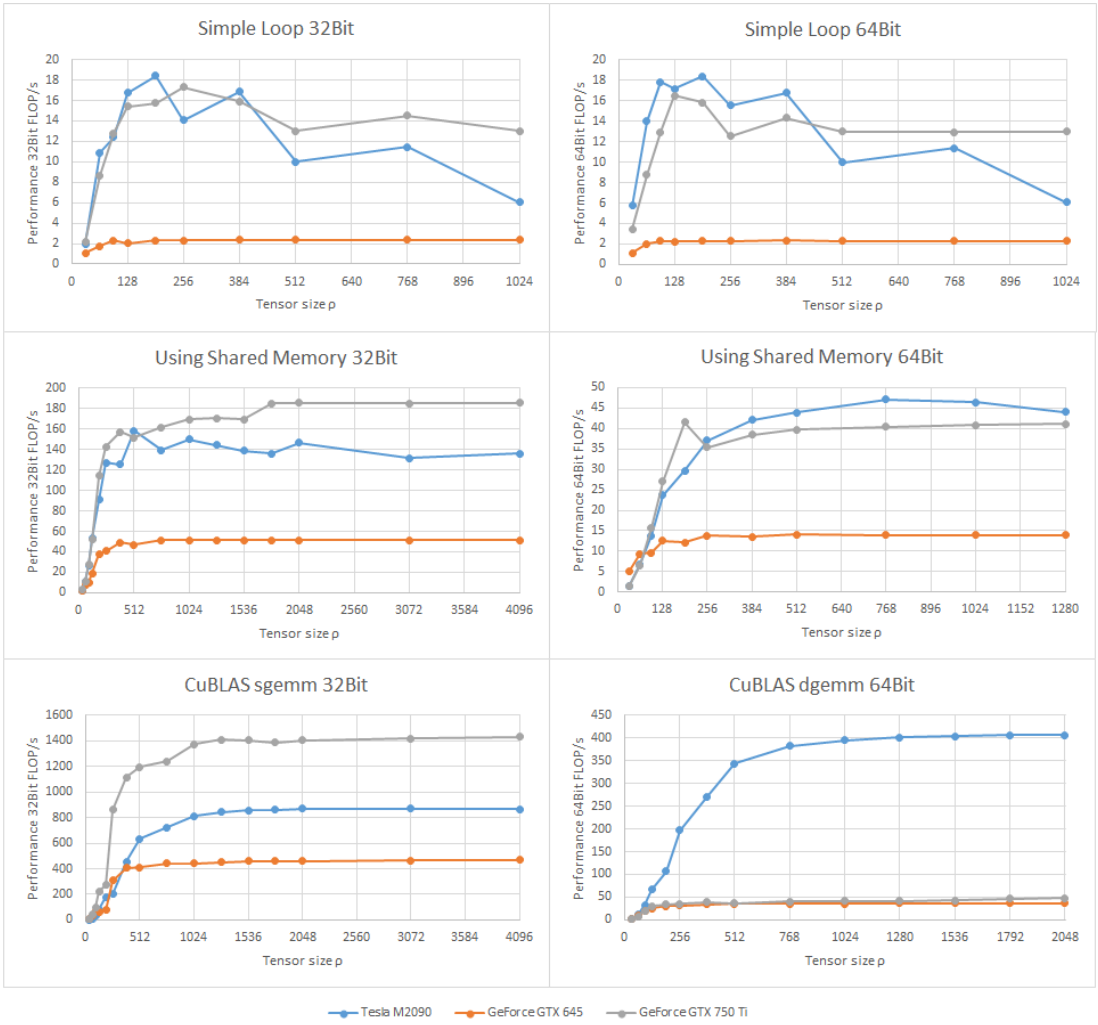


Figure 6.1.: Floating point operations per second performance of one single kernel.

times faster on the Tesla and Maxwell card. Profiling reveals that the memory access has been drastically improved. The L2 Cache is better utilized by a very high 73% hit rate. By using the shared memory the loaded data is better reused. The load instructions decreases by a factor of almost 10. And also the overhead for computing positions of the matrix values and handling loops has been reduced. Table 6.3.1 shows some of the profiling data on the Maxwell card.

The best results comes with the CuBLAS implementation. We get a performance of 50% of the peak performance and higher for sufficient large problems. Values greater than 50% can only be achieved by an excessive use of the *Fused Multiply Add Operation* (FMA). It is the only operation on a Nvidia GPU that can perform two operations in one clock cycle. In linear algebra a lot of functions uses FMA. Using the Profiler, we can see that all floating point operations are FMAs except for the CuBLAS implementations where a very small additional portion are multiplications ($< 0.01\%$).

We can see, that the GeForce GTX 750 Ti benefits most from the CuBLAS implementa-

Algorithm	Simple Loop	Shared Memory	CuBLAS
Occupancy	96.3%	49.3%	24.6%
Floating Point Instructions	$2.147 \cdot 10^9$	$2.147 \cdot 10^9$	$2.149 \cdot 10^9$
Hit Rate for Global Loads	8.33%	73.53%	50.01%
Global Load Instructions	$2.416 \cdot 10^9$	$2.852 \cdot 10^8$	$8.388 \cdot 10^6$
Global Load to Floating Point Instructions Ratio	1.125	0.133	0.004
Integer Instructions	$1.188 \cdot 10^{10}$	$3.017 \cdot 10^9$	$7.805 \cdot 10^7$
Integer to Floating Point Instructions Ratio	5.53	1.47	0.036

Table 6.2.: Profiling data of the kernels of the different algorithms. The data was collected from the 32Bit floating point version using the parameter $n = 1024$ on a GeForce GTX 750 Ti.

tion with a performance of up to 88% of the theoretical peak performance. Compared to the self-written sgemmTN using shared memory the CuBLAS implementation further reduces global memory load instructions and drastically decreases the overhead of integer operations.

Another interesting point is, that the occupancy is decreasing with the algorithms with the better performance. It is known that an occupancy of 100% is not needed to achieve peak performance [20], but for memory bound problems it might be easier. On the other side, using lower occupancy enables the programmer to use more of the very limited resources like registers and shared memory, which are clearly needed for the tensor contraction.

Looking at the 64Bit algorithms we can see, that the Tesla M2090 creates the best results in absolute values. This is caused by the fact, that Tesla GPU has a lot more 64Bit floating point units compared to GeForce cards. In the Fermi architecture the ratio of double/single precision units is 1:2 for Tesla and 1:8 for GeForce. Like for the single precision, the double precision performance on Tesla M2090 is about 61% of the theoretical peak performance. For later architectures the ratio is further decreasing in favor of the single precision units. Kepler has a ratio of 1:24 for GeForce and 1:3 for Tesla GPU. Maxwell does not have any dedicated double precision units so far, since for games only single precision units are needed [15]. However double precision can be emulated by the single precision units in a ratio of 1:128 [17]. Our results claim a better ratio of at least 1:32. In relative values the Kepler GPU gets about 91% of the theoretical maximum. For Maxwell we only provide absolute data for the double precision performance since no official data about the performance of 64Bit doubles exists.

6.3.2. Non Cyclic Tensor Train Contraction

Looking at the complete contraction of non-cyclic Tensor Trains we can see, that we also get quite good results. Figure 6.2 shows the performance of the full Tensor Train contraction using different parameters. Comparing those with the single kernel execution we can see, that the performance of the algorithm heavily depends on the performance of the kernel.

The synchronization and the handling of the tensor data have only small impact to the runtime. Also the starting and finishing kernels do not affect the overall performance. Since they have a much smaller workload they usually working with lower floating point operations per second. However their workload is so small that their worse performance is negligible.

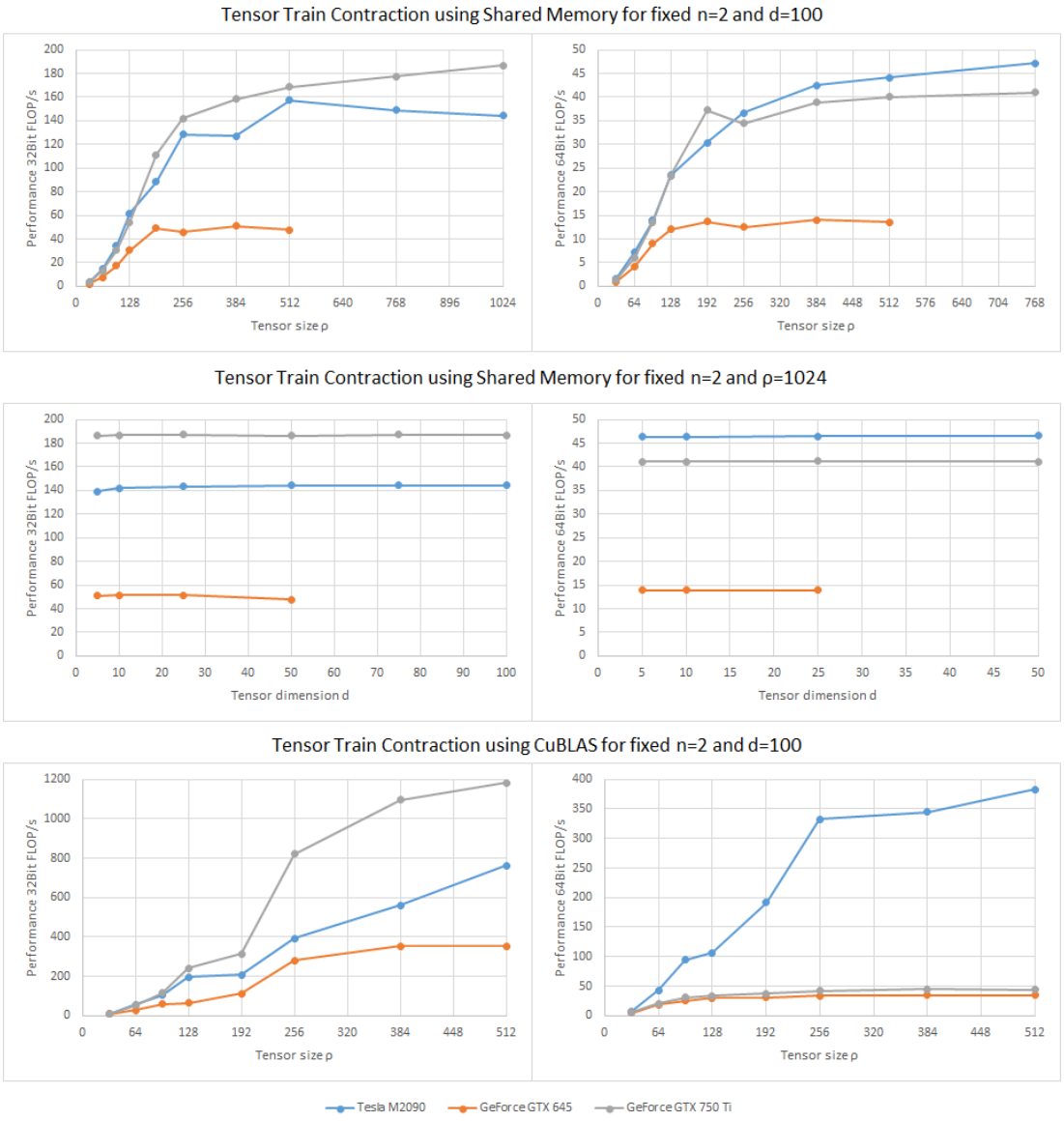


Figure 6.2.: Floating point operations per second performance of the contraction of non-cyclic Tensor Trains using various parameters and algorithms.

Testing different tensor dimensions we can see that there is only little change in performance. For some lower dimensional tensors the impact of the slower starting and finishing kernels do not influence the performance of the algorithm. For the higher dimensional tensors the overhead caused by the CPU for synchronizing and kernel invocation is also very

small.

6.3.3. Cyclic Tensor Train Contraction

First we are looking at the algorithm with the implicit permutation. The main problem of the algorithm is the requirement for the temporary tensors to fit into the memory. By only using multiples of 32 for ρ the only values we can test are 32 and 64 for the Tesla M2090 and GeForce GTX 750 Ti. The GeForce GTX 645 only has 1GB RAM so the four dimensional temporary tensors with $\rho = 64$ do not fit into memory. The main kernel is like the shared memory algorithm of section 4.2.2 based on the sgemm implementation of [21]. Therefore the overall performance of the algorithms is comparable to the performance of the single kernel. Figure 6.3 shows some results using this algorithm. Since we handle the tensors as one big entity we reach for the whole algorithm equivalent values as the maximum performance of the single kernel. Again the starting and finishing kernels influence the performance only little. As we can see only for lower dimensions they have a moderate impact.

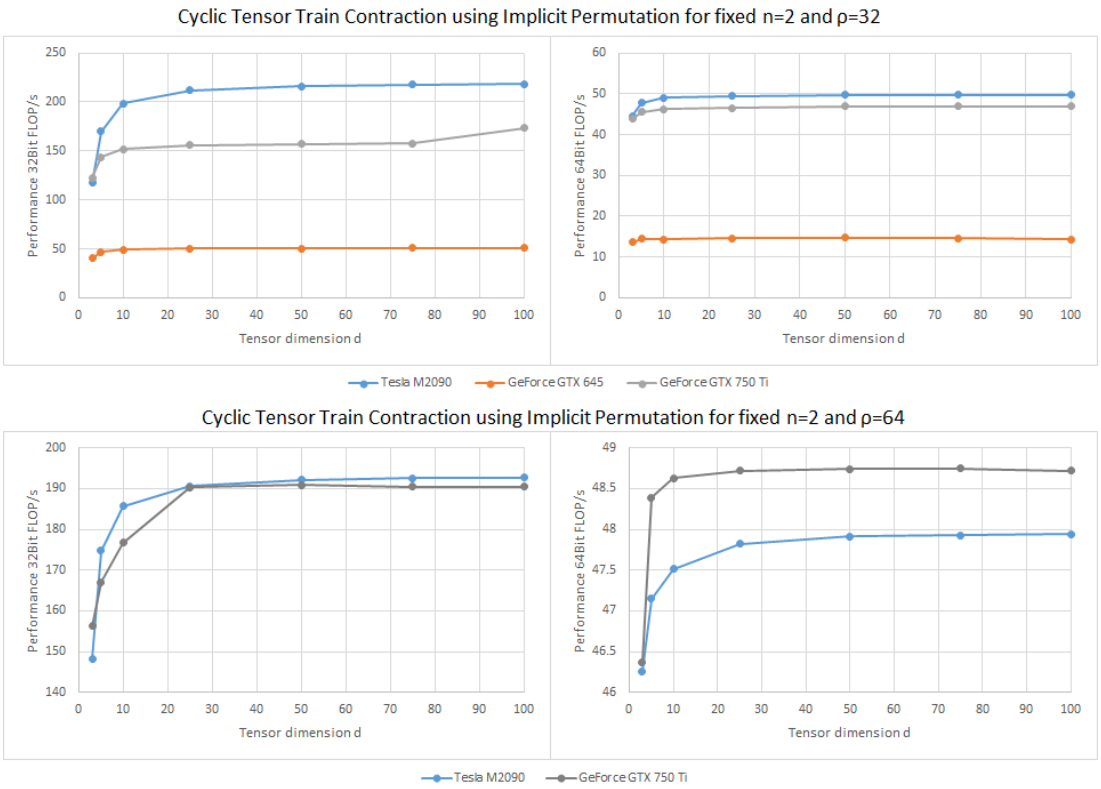


Figure 6.3.: Performance data of the cyclic Tensor Train algorithm with implicit permutation.

For the use of tensors with larger ρ we have to split up the tensors into smaller pieces. Using the efficient implementation for the non-cyclic Tensor Train contraction and sufficient large tensors, we also receive results like the single use of a CuBLAS kernel. Figure

6.4 shows the results for the reduced memory requirement. The algorithm uses many small tensor contractions, so for smaller values of ρ the performance is low. In case of $\rho = 32$ or $\rho = 64$ one could use other algorithm for better performance.

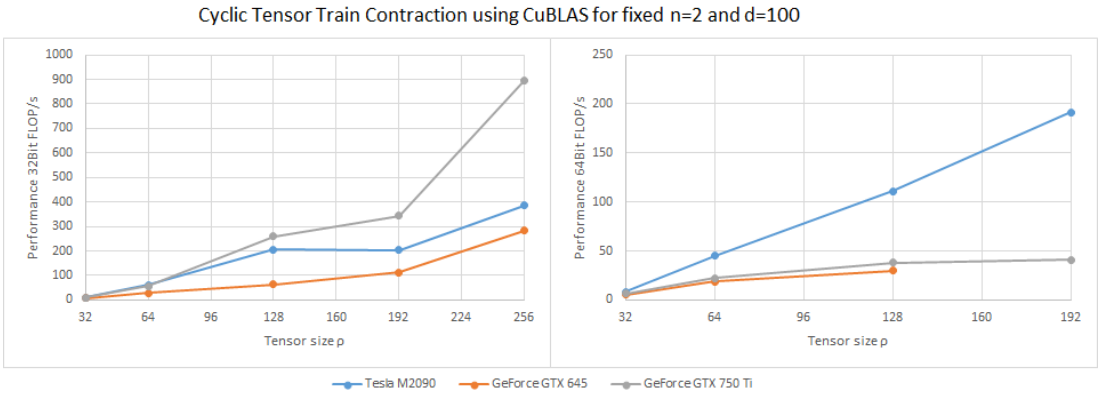


Figure 6.4.: Performance data of the cyclic Tensor Train contraction using reduced memory requirement and CuBLAS.

For the massive uses of BLAS functions we tried the dynamic parallelism feature introduced for compute capability 3.5. The dynamic parallelism allows to start kernels directly from the device and therefore reduces the need of communication between host and device. Figure 6.5 shows the speedup using kernel calls from the device. Directly comparing the algorithm with and without dynamic parallelism shows us, that dynamic parallelism runs only for the larger Tensors faster. The GPU also have to handle the CuBLAS calls in single thread execution. In such cases the CPU perform faster and therefore can reduce the impact of the latency of the communication. The benefit of using dynamic parallelism for large Tensor Trains is not very high. On the other side, we already reach values over 70% of the theoretical peak performance in our test and expect even higher values for larger problem sizes like those from the non-cyclic Tensor Trains. For those performance values a speedup of up to 2% is not easy to reach.

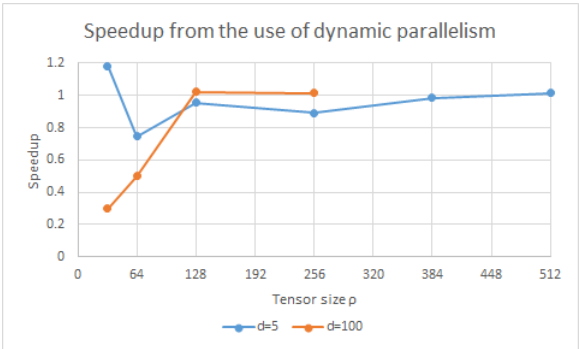


Figure 6.5.: Speedup from the use of dynamic parallelism on the GeForce 750 Ti

Since we have a lot of independent non-cyclic Tensor Trains that have to be computed, we also tried the use of multiple GPUs. We used two Tesla M2090 cards directly connected

to one Intel Xeon E5-2670 processor as host. The data of the tensor trains were copied to both Tesla cards. Each GPU was handled by one thread on the host using OpenMP. For work distribution we split the outer sum of equation 5.2. The reduction of the results from the GPUs took place on the host. For larger problems we got a speedup of about 2. The results for the speedup are illustrated in Figure 6.6.

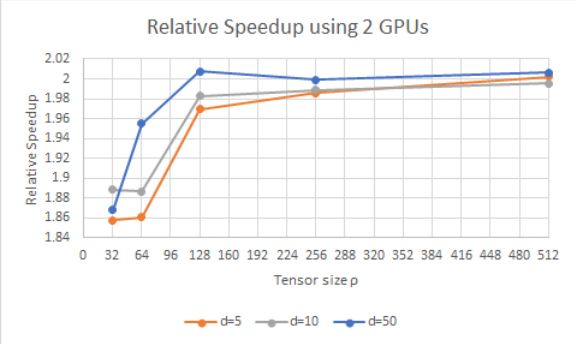


Figure 6.6.: Relative Speedup by the use of two GPUs

7. Conclusion

In this thesis we presented algorithms for the contraction of Tensor Trains on GPUs. We have seen that the contraction of tensors is well suited to be executed on GPUs. The massive use of the Fused Multiply Add Operator gives us a throughput that only few applications in practice can achieve. On all tested GPUs we reached over 50% of the maximum FLOP/s for single precision values. Especially for Maxwell, the latest Nvidia GPU architecture, we can get about 88% of the theoretical peak performance. We only tested the algorithms on Nvidia GPUs, but we can expect that we get similar results on GPUs of other manufacturers using OpenCL. Also on parallel CPU systems and accelerators like the Xenon Phi it is most likely that we can extremely benefit from parallelism.

Since the code for the cyclic Tensor Trains has very little data dependencies we have seen that we can speed up the algorithm using two GPUs using OpenMP on a single host. Even for smaller Tensor Trains we got a speedup of 2. The computation of the inner product of two cyclic Tensor Trains is highly compute bound ($d \cdot n \cdot \rho^2$ Memory compared to $d \cdot n \cdot \rho^5$ Operations). Because of the high operations per memory ratio, we can expect that a MPI implementation would also benefit from large clusters of GPUs.

We have also seen, that we can reduce the additional memory needed for the computation of the inner product of two cyclic Tensor Trains from $n \cdot \rho^4$ to $n \cdot \rho^2$. This was done by splitting some tensors into smaller sub-tensors, a similar technique known from large matrix multiplications. This allows us to handle tensor trains with bigger values of ρ and therefore more accurate tensor approximations.

Bibliography

[1] http://icl.cs.utk.edu/magma/, 2014.

[2] Arndt Bode. Multicore-architekturen. *Informatik-Spektrum*, 29(5):349–352, 2006.

[3] Christopher M Dawson, Jens Eisert, and Tobias J Osborne. Unifying variational methods for simulating quantum many-body systems. *Physical review letters*, 100(13):130501, 2008.

[4] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. On the best rank-1 and rank-(r 1, r 2,..., rn) approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1324–1342, 2000.

[5] NVIDIA GmbH. http://docs.nvidia.com/cuda/cuda-samples/#device-query, year= 2014.

[6] NVIDIA GmbH. https://developer.nvidia.com/cuda-toolkit-archive, 2014.

[7] NVIDIA GmbH. http://www.nvidia.de/, 2014.

[8] Lars Grasedyck, Daniel Kressner, and Christine Tobler. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen*, 36(1):53–78, 2013.

[9] Wolfgang Hackbusch. *Tensor spaces and numerical tensor calculus*, volume 42. Springer, 2012.

[10] Thomas Huckle and Konrad Waldherr. Numerical linear and multilinear algebra in quantum control and quantum tensor networks, September 2014.

[11] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[12] Aaftab Munshi et al. The opencl specification. *Khronos OpenCL Working Group*, 1.2, 2014.

[13] Nvidia. Whitepaper: Nvidia's next generation cuda compute architecture: Fermi. *NVIDIA Corporation*, 2009.

[14] Nvidia. Whitepaper: Nvidia's next generation cuda compute architecture: Kepler gk110. *NVIDIA Corporation*, 2012.

[15] Nvidia. Whitepaper: Nvidia geforce gtx 750 ti. *NVIDIA Corporation*, 2014.

[16] CUDA Nvidia. Cublas library user guide. *NVIDIA Corporation*, v6.5, 2014.

[17] CUDA Nvidia. Cuda c programming guide. *NVIDIA Corporation*, v6.5, 2014.

[18] CUDA Nvidia. Profiler user's guide. *NVIDIA Corporation*, v6.5, 2014.

[19] IV Oseledets and EE Tyrtyshnikov. Recursive decomposition of multidimensional tensors. In *Doklady Mathematics*, volume 80, pages 460–462. Springer, 2009.

[20] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.

[21] Vasily Volkov. http://www.cs.berkeley.edu/~volkov/, 2014.