FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Scaling Out: Distributed Memory Parallelism for Einsum Trees

BACHELOR'S THESIS

to attain the academic degree
Bachelor of Science (B.Sc.)
in Computer Science (Informatik)

FRIEDRICH SCHILLER UNIVERSITY JENA
Faculty of Mathematics and Computer Science

submitted by Justus Dreßler
born on 04.08.2002 in Jena
advisor: Prof. Dr. A. Breuer
Jena, 23.1.2025

Abstract

Tensor contractions are critical to the efficient execution of machine learning workloads. Einsum expressions offer a succinct and declarative way to express tensor contractions. `einsum_ir` is a library to compute a series of einsum expressions in tree form efficiently on a single compute unit, but its scaling across NUMA domains is lacking. This thesis proposes four algorithms to accelerate `einsum_ir` with distributed memory parallelism to improve the performance on multi socket systems. These algorithms outperform the existing shared memory parallelization for large enough tensors on dual socket systems such as an Nvidia Grace CPU Superchip.

Kurzfassung

Tensor Kontraktionen sind für die effiziente Ausführung von maschinellem Lernen entscheidend. Einsum Ausdrücke bieten dabei eine prägnante und deklarative Möglichkeit, um Tensor Kontraktionen auszudrücken. `einsum_ir` ist eine Bibliothek um eine Serie an Einsum Ausdrücken in der Form eines Baumes effizient auf einer einzelnen Recheneinheit auszuwerten, aber es skaliert unzureichend gut über NUMA-Domänen hinweg. In dieser Arbeit werden vier Algorithmen zur Beschleunigung von `einsum_ir` mit verteiltem Speicherparallelität vorgeschlagen, um die Leistung auf Mehr-Sockel-Systemen zu verbessern. Die Algorithmen übertreffen die bestehende verteilter Speicher Parallelisierung für ausreichend große Tensoren auf Zwei-Sockel-Systemen wie dem Nvidia Grace CPU Superchip.

Contents

1 Introduction

Tensor contractions are at the core of many applications like fluid dynamics and machine learning models. Their fast execution is paramount for these applications. Einsum expressions are a very succinct way to describe tensor contractions. Their evaluation is supported in many popular libraries like Torch[3], Numpy[4] or TensorFlow [6]. A slightly different approach is presented by `einsum_ir`[1]: Instead of inputting individual einsum expressions a user inputs a whole series of contractions as an einsum tree. This enables additional performance gains by changing the execution order of contractions and the dimension order of intermediate tensors, which lets `einsum_ir` reduce the amount of transpose operations significantly. The current version of `einsum_ir` exhibits bad scaling behavior if more than one NUMA domain is used. It also currently cannot use multiple nodes at all to speed up the contraction.

The contribution of this thesis is the creation of a distributed memory parallelization layer with MPI to improve `einsum_ir`'s scaling across multiple NUMA domains and enable the use of multiple nodes to speed up the contraction. I achieved that by developing four algorithms, one master-worker algorithm specifically aimed at dual socket systems and three algorithms that can scale to any number of NUMA domains and/or nodes, where data gets kept distributed throughout the runtime. A fundamental design aspect in all these algorithms is overlapping computation and communication with a dedicated communication thread to reduce idle times of all threads. The algorithms are implemented in C++ and their performance are tested on dual socket systems. The tests showcase the promising performance of all algorithms for large enough tensor contractions.

2 Background

2.1 Tensor Contractions

Tensor contractions are a generalization of batched matrix multiplications on n-dimensional arrays called order-n tensors. In a binary contraction of two tensors A and B to a third tensor C we refer to the first input tensor A as left tensor and the second input tensor B as the right one. We use the following classification for the dimensions in the tensors A,B and C:

dimensiontype	present in A	present in B	present in C
c	✓	✓	✓
m	✓	×	✓
n	×	✓	✓
k	✓	✓	×

We also define two operations for our distribution. The first operation is *cutting* a tensor along a dimension o. It shall refer to dividing a tensor into $n \in \mathbb{N}$ chunks. The first such chunk composes the all values of the tensor with $o \in [1, \frac{|o|}{T}]$, the second all values with $o \in [\frac{|o|}{T} + 1, 2 \cdot \frac{|o|}{T}]$ and so on. The last chunk shall consist of all remaining

values with $o \in [(n-1) \cdot \frac{|o|}{T}, |o|]$. The second operation is *concatenating* a tensor as the inverse operation of *cutting*.

2.2 Einsum Expressions

Einsum expressions allow for a more succinct expression than the typical Tensor notation as used in 2.1. Instead of writing $C_{pstuv} = \sum_q \sum_r A_{pqrs} B_{tuqvr}$ the same contraction is expressed as $A_{pqrs} B_{tuqvr} \to C_{pstuv}$. The summation signs are now implicit. Einsum expressions can also describe the contractions of more than 2 tensors[8]. Instead of writing $D_{ij} = \sum_k \sum_l A_{ik} B_{jkl} C_{il}$ it is expressed as $A_{ik} B_{jkl} C_{il} \to D_{ij}$. All expression in the rest of this thesis are written as simplified einsum expression, either written as $A, B \to C$, leaving out the indices, or as $pqrs, tuqvr \to pstuv$, leaving out the tensor variables.

We make a few observations about the different dimension types introduced in Section 2.1 for the binary tensor contraction $A, B \to C$:

Assuming that A,B and C share a c-dimension c_0 the expression is equivalent to:

1. cut A,B and C along c_0 into $T \in \mathbb{N}$ chunks $A_1 \dots A_T$,$B_1 \dots B_T$ and $C_1 \dots C_T$
2. $\forall i \in [1, T] : A_i, B_i \to C_i$
3. concatenate all chunks C_i along c_0 to C

Assuming that A and C share an m-dimension m_0 and B and C share an n-dimension n_0 the expression is equivalent to:

1. cut A and C along m_0 into $T \in \mathbb{N}$ chunks $A_1 \dots A_T$ and $C_1 \dots C_T$
2. cut B and all chunks C_i along n_0 into T chunks $B_1 \dots B_T$ and $C_{i,1} \dots C_{i,T}$
3. $\forall i, j \in [1, T] : A_i, B_j \to C_{i,j}$
4. $\forall i \in [1, T]$: concatenate all chunks $C_{i,j}$ along n_0 to C_i
5. concatenate all chunks C_i along m_0 to C

Assuming that A and B share a k-dimension k_0 the expression is equivalent to:

1. cut A and B along k_0 int $T \in \mathbb{N}$ chunks $A_1 \dots A_T$ and $B_1 \dots B_T$
2. create T tensors of the same shape as C that we call $C_1 \dots C_T$
3. $\forall i \in [1, T] : A_i, B_i \to C_i$
4. add all tensors C_i elementwise to C

2.3 einsum_ir

`einsum_ir`[1] is a software to evaluate a series of einsum expressions expressed in tree form. An example for an einsum tree is $[A, B \to C], D \to E$. This thesis builds on top of this software, using their implementation of a binary tensor contraction

$A, B \to C$ as local primitive for my algorithms. It is important to note that the current binary tensor contraction interface expects each tensor to reside in contiguous memory.

2.4 MPI

As a tool for distributed memory parallelization we use MPI. It is a standard for distributed memory parallelization libraries like OpenMPI and MPICH. We only use point-to-point communication, where one process sends data directly to another.

3 Motivation

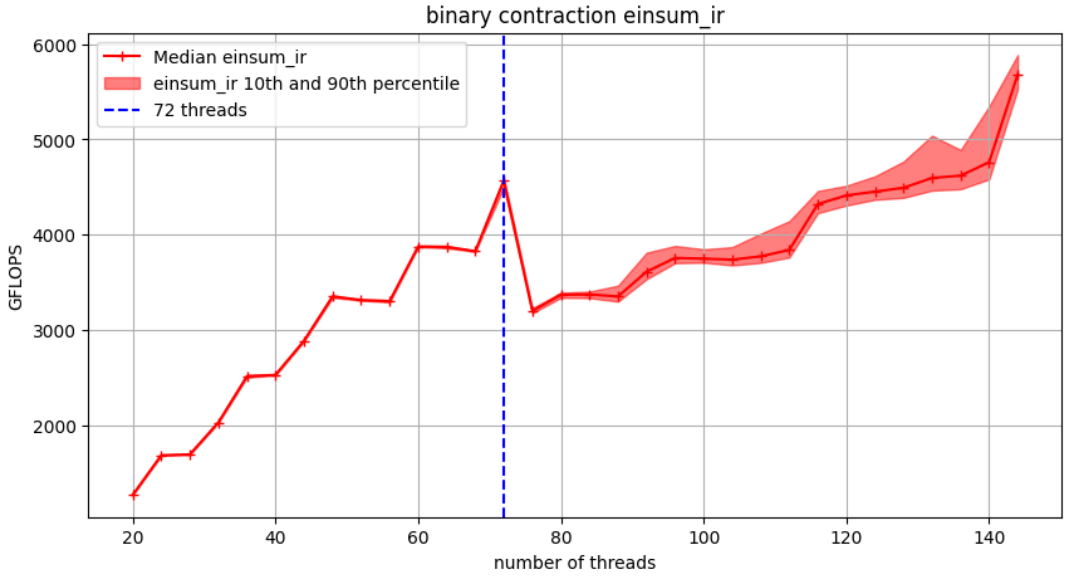


Figure 1: Performance of einsum_ir on an Nvidia Grace CPU Superchip. Grace consists of two 72-core CPUs connected via NVLink. The contraction used is $m_0 c_0 k_0 k_1 m_1, n_0 c_0 k_0 n_0 n_1 \to m_0 n_0 c_0 n_1 m_1$ with $|c_0| = 2$, $|m_0| = |n_0| = |k_0|$ and $|m_1| = |n_1| = |k_1| = 70$. JIT compilation times of `einsum_ir` are included in the measurement.

`einsum_ir` already implements shared memory parallelization with OpenMP[2], so it can already exploit all CPU cores on a single NUMA domain. But as shown in Figure 1 the performance breaks down as soon as more than one NUMA domain is used. Even using all threads on Grace improves the performance by merely 24% over only using one socket. The goal of this thesis is to improve this scaling behavior across NUMA domains.

To achieve this goal we look at four distributed memory algorithms. Before going over these algorithms in detail we should look at how other applications parallelize similar workloads. Parallelizing tensor operations is essential to machine learning, so we can take lessons from that field. The most popular practices to parallelize machine learning workloads are data, tensor and pipeline parallelism [7]. Data parallelism is specific to the training phase of machine learning models, where the weights of a model get updated in parallel. We do not want to restrict ourselves to

only AI training workloads, so this type is not applicable. That leaves tensor and pipeline parallelism. Tensor parallelism is generally favored when scaling across a single node [7], while pipeline parallelism performs better when scaling across nodes. Since our motivation stems from accelerating `einsum_ir` on a single execution node primarily, I decided to pursue tensor parallelism.

4 Master-Worker Algorithm

As our first algorithm we think of a master-worker architecture to improve performance across NUMA domains. Since the main hardware target were dual socket servers, like the Nvidia Grace CPU Superchip, we restrict our algorithm to two processes. This also trivially ensures that the communication is balanced between all processes. The basic premise of this algorithm is treating the worker as a matrix accelerator where you send parts of the input matrix to and receive the corresponding parts of the output matrix from. Since both NUMA domains should ideally be equally fast we cut all tensor across one c-dimension c_0 into $|c_0|$ chunks, with the master working on the first $\left\lceil \frac{|c_0|}{2} \right\rceil$ chunks together as one big contraction and send the remaining $\left\lfloor \frac{|c_0|}{2} \right\rfloor$ chunks to the worker to process them in parallel to the master process. To enable the overlap of communicating with the other process and computing the chunks, each process uses one of its thread exclusively as communication thread. This thread is solely responsible for communicating the input chunks from the master to the worker and the output chunks from the worker back to the master.

From the perspective of the master thread the communication and compute threads only have to synchronize at the end of the contraction, waiting for the other to finish. The worker's compute threads however have to wait until their input data is fully assembled before they can start working. To minimize the waiting times the computation and communication goes as follows, with A_i, B_i and C_i each being the i-th chunk contracted on the worker:

compute threads	communication thread
	`receive` A_1 and B_1
$A_1, B_1 \rightarrow C_1$	`receive` A_2 and B_2
$A_2, B_2 \rightarrow C_2$	`receive` A_3 and B_3 `send` C_1
...	
$A_k, B_k \rightarrow C_k$	`receive` A_{k+1} and B_{k+1} `send` C_{k-1}
...	
$A_n, B_n \rightarrow C_n$	`send` C_{n-1}
	`send` C_n

The communication thread of the master matches each `receive` of the worker with a `send` and each `send` of the worker with a `receive`. The workers compute threads have to synchronize with its communication thread after each row. To minimize the initial memory allocation on the worker it only has 2 buffers for A, B and C chunks

respectively. This is possible since all A and B chunks are used in the next step after they are received and all C chunks are sent in the next chunk after they are contracted, leaving each chunk only for two steps in the worker. The time in the contraction where the worker does not use its compute threads takes only as long as the time to communicate a chunk of each A, B and C (the first and last row in the table), so this time gets reduced the larger $|c_0|$ is. This is somewhat restricted by a more granular split also increasing synchronization overheads though. It should be noted that the $|c_0|$ dimension has to be the outermost dimension of A, B and C in the implementation I developed. This is caused by the restriction in the binary contraction interface to only work on contiguous memory as noted in Section 2.3. To fix this limitation the contraction interface would need to support strided tensors, for example by supplying a pointer to the first entry and offsets for each dimension. The MPI calls would also need to be adjusted to a custom datatype describing the strided tensors instead of sending contiguous chunks.

5 Distributed Algorithms

The previously discussed master-worker algorithm expects all input tensors before a contraction and the output tensor after a contraction to reside within a single master process. This introduces unnecessary data transfers for tensor contractions within the tree, as the output tensor must be further contracted and for that has to be communicated again to the worker. Instead, let us consider algorithms that work on distributed tensors as input and output. We formulate the algorithms in a way that they can scale up the number of processes, so we could use them for scaling across nodes, though our tests are be restricted to dual socket systems.

As basis for such algorithms, we make the following assumptions:

1. input tensors are predistributed across all processes
2. output tensors may stay distributed
3. tensors are sufficiently compute intensive that communication is not a bottleneck
4. the distributed dimension is a multiple of the total number of processes p
5. each process has multiple threads.

Given sufficiently large einsum trees with sufficiently large tensors the initial distribution of the input tensors and the final gather of the output tensor consume a negligible amount of time compared to the evaluation of the tree. Larger tensors also generally have a higher compute intensity enabling a high throughput despite the lower interconnect speeds compared to local memory speeds. These assumptions indicate that the proposed algorithms need a minimum tensor size to show improvements over the base `einsum_ir` implementation.

Assumption 4 is merely necessary to simplify the algorithms for this thesis. Should assumption 4 not hold true, for example if a dimension o with $|o| = 14$ gets distributed across $p = 4$ processes, the data could be distributed across processes with

the processes p_1, p_2, p_3, p_4 receiving $4, 4, 3, 3$ or $4, 4, 4, 2$ parts respectively. In either case each communication and contraction would merely need at most four variants depending on both input tensor states, either having a "full" chunk with $|o'| = 4$ parts or a "partial" chunk with either $|o''| = 3$ or $|o''| = 2$ parts.

The last assumption is needed as some of the following algorithms use an extra communication thread. This thread is solely responsible for pushing all communication and feeding the computation threads continuously with data to process. This design was chosen to facilitate overlapping communication with computation and should minimize waiting times for the computing threads.

With those assumptions we conceive three algorithms:

5.1 Distributed c-Dimension Algorithm

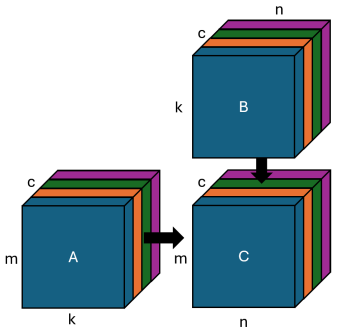


Figure 2: Visualization of the Algorithm 1 for the contraction $cmk, ckn \rightarrow cmn$; each color represents a process and the data they hold; since the tensors are distributed along the same c-dimension, each node can locally contract its chunks to generate its chunk of the output tensor.

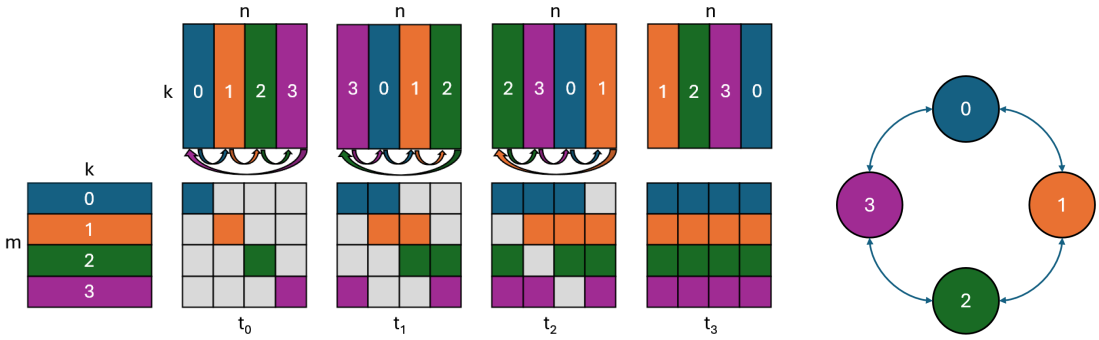
Algorithm 1 Distributed c-dimension contraction

Input: $i = \texttt{mpi\_rank}, A_i, B_i$
Output: C_i
$A_i, B_i \rightarrow C_i$

Imagine A,B and C are distributed over a c-dimension c_0. In such a case each process has to only contract its local chunks as already described in Section 2.2. This algorithm represents the best case as no communication has to occur. Cutting a tensor among its c-dimension keeps its compute intensity in contrast to cutting any other dimension, where the compute intensity falls.

5.2 Distributed m- and n-Dimensions Algorithm

Imagine A and C are distributed over an m-dimension m_0 and B is distributed over an n-dimension n_0. As described in Section 2.2, we can calculate C by contracting chunks $C_{i,j}$ out of A_i and B_j and then concatenating them over n_0 to end with C distributed over m_0. This is the first algorithm where we employ a ringlike communication scheme. As with the master-worker algorithm each process employs a



(a) distributed m- and n-dimension algorithm **(b)** communication ring

Figure 3: Visualization of Algorithm 2 for the contraction $mk, kn \to mn$; each color represents a process and the data they hold;
3a shows how in each step a chunk $C_{i,j}$ of the output gets calculated while the arrows show which data each process will acquire in the next step, being coded by the arrows tip and the processes colour.
3b shows the ring like communication setup where each process only communicates to its immediate higher and lower rank.

Algorithm 2 Distributed m and n contraction

Input: $i = \texttt{mpi\_rank}, A_i, B_i$
Output: C_i

```
next = p_(i+1) mod p
prev = p_(i+p-1) mod p
comp_buffer ← B_i
for j in 0 ... p − 1 do:
    k ← (i + j) mod p
    do in parallel:
        A_i, comp_buffer → C_{i,j}
        mpi_recv recv_buffer from next
        mpi_send comp_buffer to prev
    comp_buffer ← recv_buffer
C_i ← concat(C_{i,0} ... C_{i,p})
```

dedicated communication thread. This means that each process p_i can only communicate with process p_{i-1}, called previous process, and process p_{i+1}, called next process. Each process p_i starts by contracting its local chunks A_i and B_i to calculate $C_{i,i}$. Simultaneously each process sends its chunk B_i to the previous process and receives the chunk B_{i+1} from the next process. We use additional memory in size of one chunk $|B_{i+1}|$ to implement the communication. This repeats p times after which all chunks $C_{i,j}$ are calculated on each process p_i and get concatenated to C_i. An example for $p = 4$ and the binary contraction $mk, kn \to mn$ can be seen in Figure 3a.

In the implementation I omitted the final communication of B's chunks, since I assumed that the input tensors would not need to be reused. In the case of no reuse, this should slightly speed up the algorithm. This assumption no longer holds true if the input tensors need to be used multiple times, such as in inference workloads.

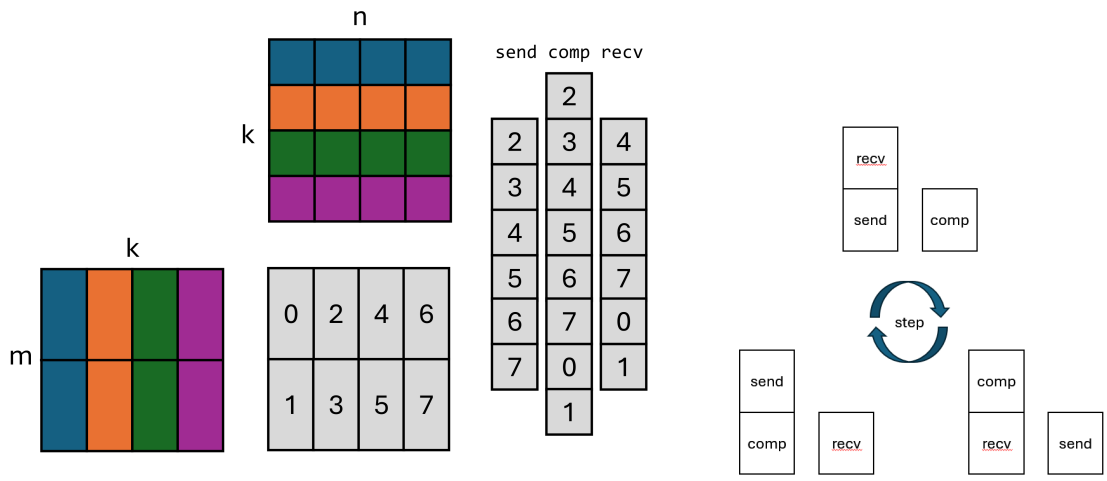
If that was the case it would also be necessary to copy the final chunk to the input tensor from the additional memory, provided the number of processes p be uneven.

We can leave out the final concatenation in the implementation as long as we contract each chunk $C_{i,j}$ already with the correct strides into the output tensor C_i. Due to limitations in the binary contraction described in Section 2.3 this is only possibly should each chunk $C_{i,j}$ be contiguous in C_i. To fulfill that the implementation has to restrict the m_0-dimension to be the outermost dimension of C.

We could also consider another very similar algorithm where chunks of A get sent and C is distributed over n_0 instead. Such an algorithm is omitted here, since the same effect can be achieved by swapping the input tensors as $A' \coloneqq B$ and $B' \coloneqq A$, which results in $A', B' \to C$.

The algorithm employs a ring-like communication pattern to reduce memory usage as shown in Figure 3b. Instead of moving all the tensors in each time step, one could imagine an algorithm where each process keeps its chunk B_i and sends them directly to each other process. Each process still needs one tensor to contract on and another to handle the communication at the same time. Since the initial tensor B_i may no longer be moved now, this takes an additional buffer of size $|B_i|$.

5.3 Distributed k-Dimension Algorithm



(a) distributed k algorithm **(b)** rotation of buffer roles

Figure 4: Visualization of Algorithm 4 for the contraction $mk, kn \to mn$;
4a shows the order that process 0 (blue) adds its partial updates on the output chunks and which are sent to process 3 (purple) and received from process 1 (orange).
4b show the output buffer as two connected buffers and the additional memory and how after each step their roles change. After three steps each buffer has the same role again.

Imagine A and B are distributed over a k-dimension k_0, C is distributed over an m-dimension m_0 and B has an n-dimension n_0. As described in Section 2.2, each process p_i could just compute its partial update $A_i, B_i \to C_i$ and then add them together with the other processes. This means that each process has to hold a partial update for all values in the output tensor. To reduce the memory footprint

Algorithm 3 rotate

Input: `send_buffer`, `comp_buffer`, `recv_buffer`
Output: `send_buffer`, `comp_buffer`, `recv_buffer`

```
temp ← send_buffer
send_buffer ← comp_buffer
comp_buffer ← recv_buffer
recv_buffer ← temp
```

Algorithm 4 Distributed k contraction

Input: $i = $ `mpi_rank`, A_i, B_i, j
Output: C_i

```
next = p_{(i+1) mod p}
prev = p_{(i+p-1) mod p}
send_buffer ← firstHalf(C_i)
comp_buffer ← secondHalf(C_i)
recv_buffer ← new memory
repeat (p + 1) mod 3 times //last contraction on secondHalf(C_i)
    rotate(send_buffer, comp_buffer, recv_buffer)
A_{i,0}, B_{i,(i+1) mod p} → comp_buffer
for j in 0 ... 2 * p − 1 do:
    k = j mod 2
    m = (i + 1 + j/2) mod p
    do in parallel:
        A_{i,k}, B_{i,m} → comp_buffer
        mpi_recv recv_buffer from next
        mpi_send send_buffer to prev
    rotate(send_buffer, comp_buffer, recv_buffer)
A_{i,1}, B_{i,i} → comp_buffer
```

we augment the algorithm by calculating only chunks of the partial updates of C by cutting it along n_0. The basic idea is that each process gets a partial chunk from the next process, adds its own update on top and then sends the update to the previous process, employing a ring like communication scheme as shown in Figure 3b. If each process starts with the chunk of its next neighbour, each process ends with its own chunks of the output tensor. To overlap computation and communication we further cut each chunk in half over m_0, we call these subchunks. This enables each process to calculate a subchunk in each step while sending the last subchunk it calculated to the previous process and get the next chunk it needs from the next process.

An example of how this would look from the perspective of process p_0 is seen in Figure 4a. We use the output chunk as memory during the algorithm to save on memory needs, which are visualized in Figure 4b as the two connected subchunks. We need additional memory for a third subchunk so we can simultaneously send the last update, calculate the current one and receive the next one. This is visualized in 4b as the single subchunk next to the connected subchunks. To ensure that the updates end in the output tensor and not in the additional memory, we use

the observation that the roles of the subchunk changes in each step in the order `recv` → `comp` → `send` → `recv` and that after three steps each subchunk has the same role again. With that in mind we just need to know the end state we want, which is the bottom left state where the first subchunk of the output chunk gets calculated in the second to last step and the second subchunk gets calculated in the last step, and the number of steps, which equal $2 \cdot p$. With that information the role each subchunks starts with is just determinable by a modulo calculation, as seen in Algorithm 4.

In the implementation this algorithm has additional restrictions. Since the contraction interface only works on contiguous memory, m_0 has to be the outermost dimension of A and C and n_0 has to be the outermost dimension of B. If strided tensors are supported in the future the restriction on n_0 would cease to exist and as long as the additional memory is allowed to be a whole chunk instead of just a subchunk the one on m_0 would also be irrelevant.

It should also be noted that this algorithm only works under the assumption that we contract a new C tensor and do not add onto an existing one, as we could not use its extra memory in that case. Instead, a slightly different algorithm would likely be preferable, where each process only calculates its own partial updates to a subchunk and then sends them as a reduction to the process which holds the output chunk in the next step. This needs $|C|$ additional addition operations and needs a full chunk of additional memory so it can hold two subchunks at a time.

As a variation on this algorithm we could imagine an algorithm with the roles of m_0 and n_0 swapped. This was not explored because the two input tensors can be swapped for the same effect. We could also cut C and only one input tensor over just one m- or one n-dimension $2 \cdot p$ times. This would achieve the same basic algorithm and might be preferable if one dimension is much larger than the other.

6 Evaluation

6.1 Experimental Setup

Implementation of algorithms. The algorithms were implemented in C++ code. The algorithms are built on top of `einsum_ir` and use `torch` for the test setups. The contractions are hard-coded in the current implementation.

Platform. The algorithms were tested on a server with an Nvidia Grace CPU Superchip in the Uni Jena and on an AWS g8c.metal-48xl Graviton instance. All programs were compiled and executed with `g++` and `MPICH`. `einsum_ir` used `LIBXSMM`, `Torch` and `OpenBLAS` on Graviton. On Grace `einsum_ir` used Nvidia's `NVPL BLAS` instead of `OpenBLAS`.

Measurements. All measurements show the average result of 10 contractions for each size. I used Python Matplotlib[5] to visualize the results. The JIT Compilation time of `einsum_ir` is included in all measurements.

6.2 Distributed einsum_ir Performance

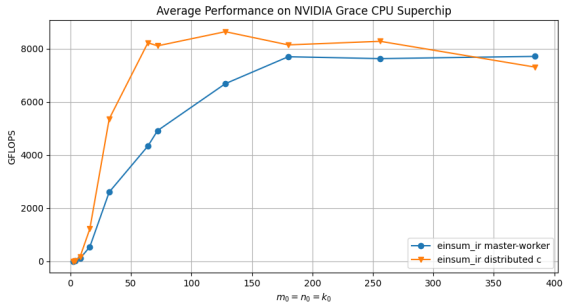
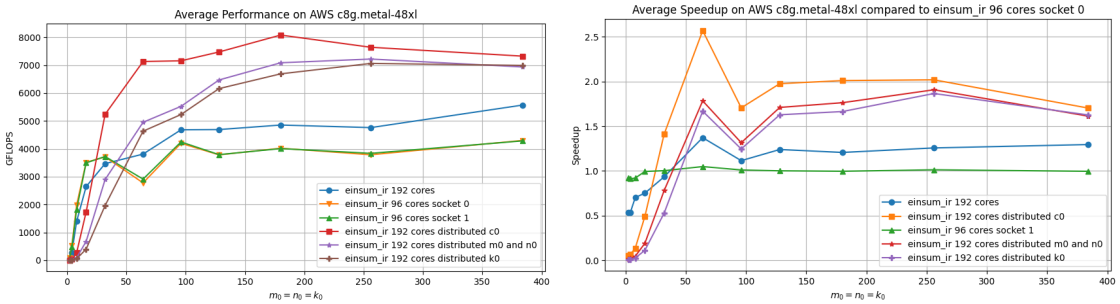


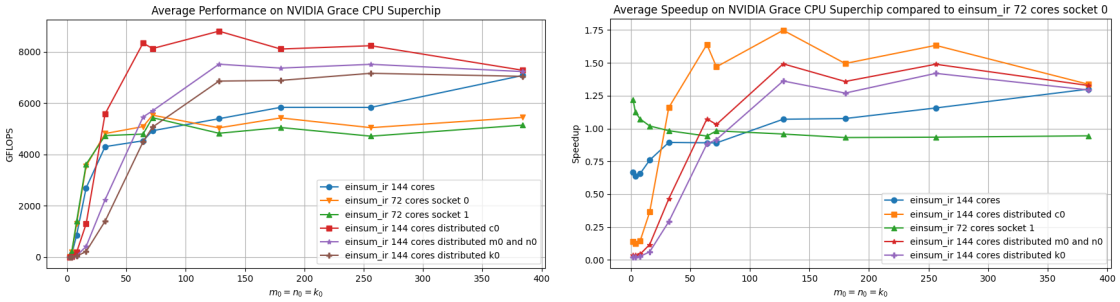
Figure 5: Performance of the distributed c-dimension algorithm and the master-worker algorithm on a dual socket NVIDIA Grace CPU Superchip. The contraction is $c_0m_0k_0k_1m_1, c_0n_0k_0n_1k_1 \rightarrow c_0m_0n_0n_1m_1$ with $|c_0| = 2$, $|m_0| = |n_0| = |k_0|$ and $|m_1| = |n_1| = |k_1| = 70$. Grace has 72 cores per NUMA domain.

First we compare the performance of the master-worker algorithm with the distributed c-dimension algorithm. We evaluate these algorithms on the einsum expression $c_0m_0k_0k_1m_1, c_0n_0k_0n_1k_1 \rightarrow c_0m_0n_0n_1m_1$. Figure 5 shows, that the distributed c-dimension algorithm performs significantly better than the master-worker algorithm especially for small tensors. This is expected since the distributed c-dimension algorithm performs no communication and thus is not slowed down on small tensors by a low compute intensity which suffers more from slower interconnect speeds. Its advantages diminish the higher the compute intensity is, as seen from $|m_0| = |n_0| = |k_0| = 180$ onwards. At $|m_0| = |n_0| = |k_0| = 384$ the master-worker algorithm even performs better than the distributed c-dimension algorithm. I could not test larger sizes as the messages got to big for MPI, which uses a 32-bit signed integer for the element count and due to time constraints I could not rewrite the methods to send multiple smaller messages instead. The overall result still shows the greater potential in a distributed algorithm, as it can scale easier to more nodes and performs better on most sizes, at least those I could test. It also places fewer restrictions on the contraction itself, as the distributed c-dimension does not have to be the outermost dimension of all tensors.

Next we compare the more general distributed algorithms against each other and the base implementation. For that we look at the results in Figure 6. For tensors larger than $|m_0| = |n_0| = |k_0| = 128$ we see significant performance improvements of the new distributed memory algorithms over the base shared memory implementation on both Grace and Graviton, except for $|m_0| = |n_0| = |k_0| = 384$ on Grace, where shared memory implementation matches the distributed memory's performance. We also note that for small tensors the distribution harms the performance, which suggests that it could be beneficial contracted small tensors on only one process. We also observe that in general the distributed c-dimension algorithm performs significantly better than both the distributed m- and n-dimension and the distributed k-dimension algorithms, which are quite close in performance to each other with the distributed k-dimension one being a bit worse. This is expected since the distributed c-dimension algorithm is the only one of the three algorithms to not have any communication happening and thus never having to run idle waiting for



(a) AWS g8c.metal-48xl



(b) Nvidia Grace CPU Superchip

Figure 6: Performance and speedup compared to running `einsum_ir` on a single socket of various algorithms. Tested on a dual socket NVIDIA Grace CPU Superchip (a) and on a dual socket g8c.metal-48xl Graviton AWS instance (b). The contraction is $m_0c_0k_0k_1m_1, n_0c_0k_0n_1k_1 \rightarrow m_0n_0c_0n_1m_1$ with $|c_0| = 2$, $|m_0| = |n_0| = |k_0|$ and $|m_1| = |n_1| = |k_1| = 70$ on Grace and 94 on Graviton. Grace has 72 cores per NUMA domain; Graviton has 96.

new data and has no synchronization overheads. The distributed k-dimension algorithm is likely slightly slower than the distributed m- and n-dimension algorithm, since it has double the amount of steps and with that also double the amount of synchronization happening.

It also shows that the second socket on Grace performs about 5% worse than the first socket. I am not sure what causes this, but in an earlier measurement all algorithms performed significantly worse on Grace as well, even showing negative speedups in the distributed m- and n-dimension and the distributed k-dimension algorithms. The difference between those first measurements and the final ones were their runtime. The first measurements comprised 100 runs for each contraction for all sizes $|m_0| = |n_0| = |k_0| = 2, 4, ..., 128$, which made the tests run significantly longer, taking multiple hours per algorithm while the new tests took under an hour for all algorithms combined. One hypothesis to explain the performance drop would be bad cooling on specifically Grace's second socket, which gets exacerbated the longer the Grace node gets stressed for. The inclusion of the AWS Graviton instance should show that the algorithms perform well and that the bad initial performance on the Grace node was an outlier.

7 Future Work

To make this thesis' result impactful enough to include into `einsum_ir` there is still work to do. First would be rewriting `einsum_ir`'s binary contraction implementation to accept strided tensors, so many of the limitation on the distributed dimensions vanish. The algorithms should also be rewritten to decouple the JIT compilation from the execution, so workloads like inference can work more efficiently. While doing this support for dimensions that are not a multiple of the process number, as described in Section 5, should be added.

When this is done there would need to be development work into finding an algorithm decide when to distribute computations and when to keep them local, likely depending on the tensor sizes. There also needs to be an algorithm to decide which dimensions are best to distribute in any given einsum expression. An easy version would be to always distribute the outermost dimension of the output and pull those dimensions to the outermost position in the input tensors. This would also omit the distributed k-dimension algorithm which showed the worst performance in the examples, since the k-dimension would never be the outermost dimension in the output tensor. Finally, this algorithm and the algorithms to decide when and how to distribute the tensors would need to be implemented into the regular einsum evaluation of `einsum_ir`, most likely as an optional component.

Additionally while scaling across nodes is possible with the algorithms described in this thesis, we have not tested how performant they are at that task. It is likely that an additional pipeline parallelism layer would be ideal to scale across nodes [7]. Such an algorithm could take the tensor parallel scaling algorithms from this thesis as primitive for each node, like we took the shared memory parallelization of `einsum_ir` as primitive for each socket.

8 Conclusion

This work showcased four new algorithms to speed up `einsum_ir` with distributed memory parallelization in MPI. The algorithms showcased promising performance, outperforming the existing shared memory parallelism of `einsum_ir` on two dual socket systems. I also outlined which next steps need to be taken in order to include these algorithms into `einsum_ir`.

References

[1] Breuer et al. *Einsum Trees: An Abstraction for Optimizing the Execution of Tensor Expressions*. preprint (personal communication).

[2] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[3] R. Collobert, K. Kavukcuoglu and C. Farabet. 'Torch7: A Matlab-like Environment for Machine Learning'. In: *BigLearn, NIPS Workshop*. 2011.

[4] Charles R. Harris et al. 'Array programming with NumPy'. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[5] J. D. Hunter. 'Matplotlib: A 2D graphics environment'. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[6] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[7] Deepak Narayanan et al. *Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM*. 2021. arXiv: 2104.04473 [cs.CL]. URL: https://arxiv.org/abs/2104.04473.

[8] Tim Rocktäschel. *Einsum is All you Need - Einstein Summation in Deep Learning*. URL: https://rockt.github.io/2018/04/30/einsum (visited on 16/01/2025).

A Appendix

A.1 Algorithm Implementations

The algorithms, testing scripts and use of the algorithms is demonstrated in my fork of einsum_ir on github. A description for how the data got generated is in `Bachelor_Arbeit/data/results.readme`. The master-worker algorithm is implemented in `bench_binary_mpi.cpp` while the other three algorithms are implemented in `bench_tree_mpi_dist.cpp`. The distributed m- and n-dimension algorithm described in this work is equivalent to the `contract_distributed_m_n_out_n` algorithm in `bench_tree_mpi_dist.cpp`. The distributed c-dimension and the distributed k-dimesnion algorithm are described in
`contract_distributed_c` and `contract_distributed_k` respectively.

A.2 Hardware used

The AWS g8c.metal-48xl was used with the ami-03b37c21960cc5a6e image and 100 GiB of storage.

Last edited: January 2024

Declaration of Academic Integrity

1. I hereby confirm that this work — or in case of group work, the contribution for which I am responsible and which I have clearly identified as such — is my own work and that I have not used any sources or resources other than those referenced.

   I take responsibility for the quality of this text and its content and have ensured that all information and arguments provided are substantiated with or supported by appropriate academic sources. I have clearly identified and fully referenced any material such as text passages, thoughts, concepts or graphics that I have directly or indirectly copied from the work of others or my own previous work. Except where stated otherwise by reference or acknowledgement, the work presented is my own in terms of copyright.

2. I understand that this declaration also applies to generative AI tools which cannot be cited (hereinafter referred to as 'generative AI').

   I understand that the use of generative AI is not permitted unless the examiner has explicitly authorized its use (Declaration of Permitted Resources). Where the use of generative AI was permitted, I confirm that I have only used it as a resource and that this work is largely my own original work. I take full responsibility for any AI-generated content I included in my work.

   Where the use of generative AI was permitted to compose this work, I have acknowledged its use in a separate appendix. This appendix includes information about which AI tool was used or a detailed description of how it was used in accordance with the requirements specified in the examiner's Declaration of Permitted Resources.

   I have read and understood the requirements contained therein and any use of generative AI in this work has been acknowledged accordingly (e.g. type, purpose and scope as well as specific instructions on how to acknowledge its use).

3. I also confirm that this work has not been previously submitted in an identical or similar form to any other examination authority in Germany or abroad, and that it has not been previously published in German or any other language.

4. I am aware that any failure to observe the aforementioned points may lead to the imposition of penalties in accordance with the relevant examination regulations. In particular, this may include that my work will be classified as deception and marked as failed. Repeated or severe attempts to deceive may also lead to a temporary or permanent exclusion from further assessments in my degree programme.

Place and date

Signature