GPU Parallelization, Validation, and Characterization of the Tensor Template Library

Alexander C. Winter

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Engineering

University of Washington,
2019

Committee:

Chair: Andrew Lumsdaine

Duane Storti

Jeff Lipton

Program Authorized to Offer Degree:

Mechanical Engineering

University of Washington

Abstract

GPU PARALLELIZATION, VALIDATION, AND CHARACTERIZATION OF THE TENSOR TEMPLATE LIBRARY

Alexander C. Winter

Chair of the Supervisory Committee:

Dr. Andrew Lumsdaine

Department of Computer Science and Engineering

Previous work has developed a tool, the Tensor Template Library (TTL), which uses variadic expression template metaprogramming to capture tensor behaviors clearly and in a manner resembling the mathematical abstraction engineers are familiar with while concealing the cumbersome looping structures, in an optimized manner. This has utility in simulating physical systems in material science via finite element modelling, but with applications in systems with large numbers of small, dense tensors.

The initial work of this author was to update the TTL to operate within a graphics processing unit (GPU), build a test suite to verify those updates compiled and generated correct output in a GPU environment, and then analyze performance within a submodule of a finite element solver, the Parallel Generalized Finite Element Solver (PGFEM). Initial characterization work in a GPU environment utilizing the TTL inside a submodule of the PGFEM, the Generalized Constitutive Model (GCM), was not as performant as the raw loop implementation, nor even an MPI distributed memory solution. To determine where the problem lay within the TTL (if at all), microbenchmark tests were developed to examine distinct TTL tensor operations over varying expression categories and complexities. The microbenchmark results were contrary to those observed in the GCM and indicated the TTL was considerably faster than compiler-optimized raw loops. It did however isolate a particular class of tensor operation, tensor inner products, as a point of interest to examine the dichotomous TTL behavior. Additional microbenchmarks were developed to examine the assembly code generated by the nVidia C Compiler (NVCC). Those microbenchmarks, stripped of any potentially compounding factors that may have cast doubt on the first set of microbenchmarks, validated the previous microbenchmarking results. Analysis of the assembly indicated that, in low order tensors, near-identical assembly could be generated through manual intervention over the compiler's optimizations, however, it revealed that the compilation pipeline of the NVCC was likely to modify template source code in non-optimal ways. Template specialization of these loop structures should resolve the problem and is currently implemented in the TTL.

Table of Contents

Contents

Table of Figures

Foreword & Acknowledgements

The conclusion of this thesis marks a liminal point in my career, and I would be remiss if I did not acknowledge the many people who made this possible.

I wish to give a special thanks to Andrew Lumsdaine for his patient guidance throughout the course of graduate career, from coursework to researching and writing this master's thesis, and securing funding for my research. Your devotion to mentorship, attention, and patient guidance through endless questions and impromptu lectures have been greatly appreciated, more than I can convey here.

I would like to thank Duane Storti for his consistent perspective on alternative CUDA solutions for those of a Pythonic perspective, Jeff Lipton for his friendly and helpful demeanor as I brought all of this to a conclusion, and Kate Gayle for her indefatigable shepherding through the ME graduate program.

I would also like to thank Luke D'Allesandro for his continual input and mentorship from the very beginning of my time with NIAC/C-SWARM and Karel Matous of the Center for Shock Wave-processing of Advanced Reactive Materials (C-SWARM) for being brought into the research group and use of the center's compute resources. Kevin DeWeese was an invaluable sounding board and rubber-duck companion. Aaron Howell and Sangmin Lee were invaluable for their work on the GCM and in collecting data for performance of the TTL within PGFEM.

I also must thank the PSAAP grant from the DoE and Nathan Barton for the opportunity to work at the Lawrence Livermore National Laboratory during Spring and Summer of 2019 to further explore topics adjacent to this while there. Returning to this work with the perspective gained at Livermore was invaluable.

Lastly, I would like to thank my mother for her endless faith in my ability and restraint for the final duration of this thesis, and Jamie Waldock for her friendship, support, and belief in me from the beginning of my time in graduate school until the end.

I couldn't have done it without all of you.

1. Introduction

A great deal of work has been done in high performance computing (HPC) around optimization in solving very large sparse matrices through myriad techniques (parallelization of matrix stencils, unique design strategies to minimize memory accesses, compiler improvements to more aggressively unroll loops, inline, reorder instructions, etc. etc.) – countless tomes have been written on the topic. However, many problem spaces, particularly in materials and mechanical engineering around crystal matrices and crystal plasticity, are driven by problems requiring immense sets of small, dense matrices, or, more specifically, tensor abstractions of such small dense matrices, each of which can be individually and uniquely solved and are themselves highly parallelizable ("embarrassingly parallel"). While these can be executed natively with simple loop structures, such implementations are unwieldy, illegible, and their scale in real applications makes them cumbersome to the point of arcane with code maintenance.

Previous work by Luke D'Allessandro with the C-SWARM research group at Notre Dame developed a tool, the Tensor Template Library (TTL), which uses variadic expression template metaprogramming to capture tensor behaviors clearly and in a manner resembling the mathematical abstraction engineers are familiar with while concealing the cumbersome looping structures behind an API, in an optimized manner. This has utility in simulating physical systems in material science via finite element modelling, but with applications in any systems with large numbers of small, dense tensors.

The work of this author was to update the TTL to operate within a graphics processing unit (GPU), build a test suite to verify those updates compiled and generated correct output in a GPU environment, and then analyze performance within a submodule of a finite element solver, the Parallel Generalized Finite Element Solver (PGFEM). Considerable work was needed after an inconsistency in the performance of the TTL in comparison to a raw loop implementation was discovered that ultimately indicated either problem in the NVCC's compiler heuristics, or the memory management of the submodule of the PGFEM, the Generalized Constitutive Model (GCM) that was used to test the TTL's performance.

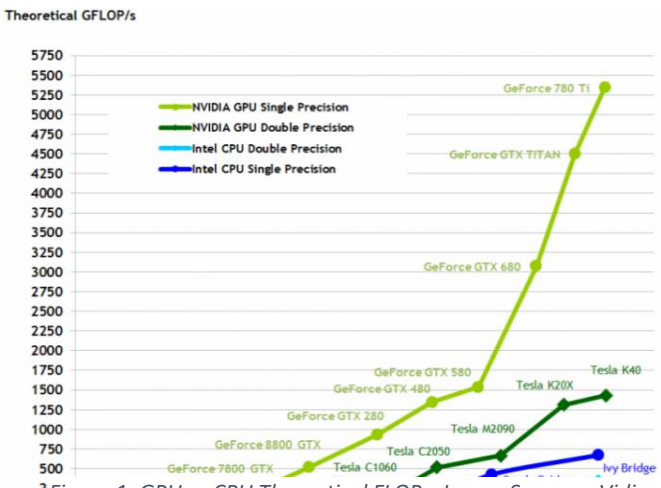
2. Literature Review and Background

2.1 A Brief History of Parallel Computation and GPUs

"If you were ploughing a field, which would you prefer: two strong oxen, or 1024 chickens?" – Seymour Cray

One of the titans of high-performance computing and founder of Cray Computing, Seymour Cray, once famously quipped this in the early days of digital computation when asked what role parallelization would play in the evolution of the field. Today, a more apropos analogy than Cray's would be along the lines of "which would you prefer: four very strong oxen, or 1024 sled dogs?" Yes, there remain many tasks that the sled dogs cannot do as quickly as the oxen - plowing smaller fields, or ones with lots of rocky terrain - but there are many things one can do with *so many* sled dogs which otherwise would be impossible on a reasonable timescale with just a handful of oxen. The scale of disparity from low-specialization parallel systems like GPUs vis-à-vis highly control-flow specialized CPUs is captured in Figure 1.

The first consumer oriented graphics processing unit (GPU), which was not integrated directly into a device (as was the case for gaming consoles through the 90s), was developed by NVIDIA in 1999, 28 years after the first digital transistor central processing unit (CPU), and has since evolved to be a ubiquitous parallel processor in modern computing, indispensable to its original development sphere (hobbyist video gaming) and increasingly invaluable to scientific computation [1], machine learning [2], and cryptoanalysis. To give a sense of scale of how much GPUs have grown in importance in computation, currently GPUs accelerating bitcoin mining, the most common parallelization paradigm within the cryptocurrency sphere, globally accounts for more power consumption than the Republic of Ireland [3]. Originally designed to provide hyper-realistic graphics and digital image rendering in real time from user inputs and interruption events, today's GPU's have evolved into a general processor with unprecedented floating-point performance and programmability. Today's GPUs greatly outpace CPUs in arithmetic throughput and memory bandwidth [4], making them the ideal accelerator for a variety of data parallel applications, particularly scientific computation, where the hardware is sometimes referred to as General Purpose Graphics Processing Units (GPGPUs). [5]



[2] *Figure 1. GPU vs CPU Theoretical FLOPs. Image Source: nVidia Corporation, CUDA Toolkit Documentation, 2019*

2.2 Mathematical Foundations

It will be instructive to the reader to examine the mathematical formalism of tensors before examining their use within the TTL, as well as a brief overview of some of the domains in which tensors find use, such as the PGFEM that is the intended client for the TTL.

2.2.1 Tensors

Whole fields of mathematics are predicated on extrapolating from singular concepts to broader abstractions: set theory explores the relationships of groups of objects without regard to the constituents; category theory explores related patterns of the various mathematical fields and abstracts them to provide structure for mathematics. In a similar vein, the most rudimentary explanation of "what is a tensor" that is not too misleading but intuitive would be an extrapolation: in the same sense that a vector is an ordered collection or array of related scalars, and a matrix an ordered collection of related vectors (an array of arrays), a tensor is an object which can describe ordered collections of related matrices, an array of arrays of arrays, and in the same manner that matrices encompass vectors and vectors scalars, tensors necessarily encompass the simpler constituent abstractions.

That said, that is not a rigorous definition, it omits the most valuable aspect of tensors (the definitions for how tensors transform), and there are numerous definitions of tensors, tensor notations, and tensor jargon which reflect the academic field that is utilizing them and the utility different tensor properties provide to those different disciplines, which leads to a dizzying (some might argue inconsistent) jargon for the tensor lexicon in the literature. A definition of tensors that is simultaneously sufficiently formal and rigorous to satisfy mathematicians vis-à-vis an intuitive definition of the structure and operations of tensors for those in applied fields, where clarity is more desirable than formalism, is a difficult and somewhat contradictory proposition, and so this author provides several to reflect the interdisciplinary utility of tensors.

A formal definition from Bertschinger's *Introduction to Tensor Calculus for General Relativity* [6] is:

"A tensor of rank (m, n), also called a (m, n) tensor, is defined to be a scalar function of m one-forms and n vectors that is linear in all of its arguments,"

While precise, this is not illuminating of itself. Here (m,n) is used to indicate the number of co- and contravariant vectors needed to describe the tensor (cf. §2.2.1.2 for a discussion of variance). A simpler definition of this author's formulation, one which admittedly eschews the mathematical richness of a tensor's geometric origins but provides an intuitive understanding would be:

"A tensor is a data structure spanning multidimensional arrays (but also singular variables), and which defines operations between and within the member arrays, as well as member arrays of other tensor objects, by way of the indices (or unique regular expressions) which span the component arrays."

A definition bridging the two is provided by Taha Sochi's *Introduction to Tensor Calculus* [7]:

"A tensor is an array of mathematical objects (usually numbers or functions) which transforms according to certain rules under coordinate changes".

As alluded to in Bertschinger's definition, the number of arrays is formally referred to as the rank or order of the tensor, each rank spanned by an index. It is important to note the rank or number of indices does not dictate the dimensionality of those indices: a vector in the Euclidean plane spanned by two of the Euclidean basis vectors is an array with two dimensions, while a vector in the Euclidean space spanned by the three Euclidean basis vectors is a three dimensional array, yet both are still rank 1 tensors (specifically (0,1) tensors). The distinction of dimensionality between the two is not indicated by the rank, as is the case in matrices (rank 2 tensors, specifically (1,1) tensors), but that having different indices reflects inoperability between the arrays (such distinct indices *can* reflect different dimensionality, but not necessarily). More plainly stated, a (2,2) matrix and a (10,10) matrix are both still rank 2 tensors, the inability to operate between the two in say a matrix multiplication would be reflected in the two tensors having different indices.

The number of data points or components needed to describe a tensor of rank N in an M dimensional space is M^N, e.g. no rank 0 tensor (i.e. a scalar) can be more than 1 number ($M^0 = 1 \forall M$), a 5 dimensional vector is a rank 1 tensor and requires 5 elements or points of data ($5^1 = 5$), a rank 2 tensor with 5 dimensions would require $5^2 = 25$, etc. From this it should be apparent that high ranking tensors quickly devolve into computationally large systems - a rank 5 tensor just existing in Cartesian 3D space would require 243 discrete components to describe, and most problems that require tensors are interested in how tensor fields evolve with time via floating point operations, from which the reader can quickly surmise how quickly such a problem grows in computational complexity. As such, a great deal of research is done in reduced order modeling for tensors to reduce the computation load of such problems [8].

An important caveat: the association of rank 2 tensors to matrices is strong but not exact – all rank 2 tensors can be represented as matrices, but not all matrices are rank 2 tensors, such as non-square matrices. One of the important distinctions between rank 2 tensors and a matrix is tensor contraction – all tensors (except rank 0 tensors) can be contracted, but not all matrices are able to do so. Cf. §2.2.1.4.3 tensor operations for a discussion of contraction and ways matrices differ from tensors.

2.2.1.1 Tensor Applications

The question of "where, if anywhere, do we find applications for higher order tensors?" is informative, as large swathes of classical physics can be encompassed with a specific but familiar rank 2 tensor, the matrix (specifically the (1,1) tensor). In general relativity, the Riemann curvature tensor is a 4th-order tensor (a (1,3) tensor) that describes the local curvature of spacetime (or, more accurately with regards to the previous paragraph, it is a tensor field); in image processing for machine learning, a 3rd-order tensor can describe the intensity values of the multiple hue channels used to create a 2-D image i.e. red, green and blue of the pixels.

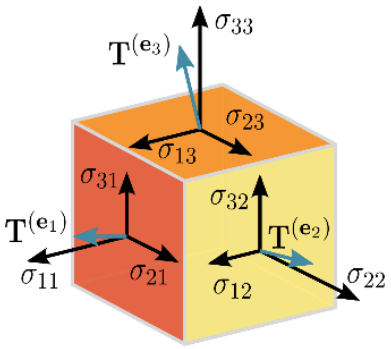


Figure 2. Cauchy Stress Tensor. Image source: Wikimedia, 2009

The tensor that is probably most familiar to the reader however is also the progenitor tensor dating to the early 19th century, the Cauchy stress tensor, a second order tensor which examines how a material crystal stretches and deforms under stress and sheer (in fact, 'tensor' etymologically derives from the latin 'tensus', meaning 'to stretch', which also gives us the word 'tension') (depicted in Figure 2). The stress tensor consists of nine component stresses that define the total stresses/shears at a point in a material, coordinated across the three planes the stresses and shears deform a material This kind of material tensor is utilized in the GCM, a subunit of the PGFem package, through the stiffness tensor.

2.2.1.2 Transformation, Variance, and Contra-variance

Vectors, as the term is used vernacularly to mean "a tuple/array of values," often treats row and column vectors as though they were identical entities, which is not universally accurate (although the ease of their mutability indicates their relatedness). All vectors, whether they are row or column, alternately scale with or against changes in the basis of the space they exist in, and the distinction of how they scale, variance, allows a more formal delineation beyond the purely notational arrangement of "data in a row" or "data in a column" to reveal their underlying behavior.

The distinction between co- and contra-variance is what distinguishes vectors and co-vectors (the latter also referred to as dual vectors or one-forms in different literature, the myriad nomenclature reflecting historical origins from different authors/disciplines). True vectors are geometrically derived and are composed of a magnitude and direction. They are invariant objects that exist independent of their coordinate system, i.e. the means we choose to describe the vector does not alter the vector's characteristics. However, ordinary non-basis vectors are described componentially from the basis vectors which span the space the vector inhabits, and because the basis itself is mutable through scaling or transformation, the description of the vector must adapt if the basis changes. How a "vector" reacts to a change in the basis reflects its variance:

a vector is contra-variant, or a column vector, if a change to the basis causes a contrary scaling change to the vector, e.g. if by scaling some basis vector e_i by 2, the vector component in the new basis would be one half. In contrast, if scaling occurs in the same manner as the change to the basis, the vector is a row vector and more formally co-variant, sometimes called a dual vector, co-vector, or one-form. To reflect the myriad sources and synonymous interchangeability of them, this author will generally refer to co-vectors as such when discussing variance, one-forms when discussing transformation and mapping, and dual vectors when discussing bases, spaces, or contrasting co- and contra-variant vectors (but in the spirit of the wanton inter-changeability of the terms in the literature, this may not be 100% consistent). Notationally, in instances where there may be ambiguity, → accents are used to indicate vectors, and ~ accents to indicate duals.

This distinction of variance fundamentally derives from what a "vector" is and describes. Column/contra-variant vectors are built componentially from basis vectors which span a vector space to encapsulate a direction and magnitude, whereas row/co-variant vectors are functional maps. A co-vector takes a vector and outputs a scalar, and multiple co-vectors in a dual vector space can, by outputting multiple scalars, build a higher dimensional vector; a (column) vector is a true vector, and describing a higher dimensional vector requires more independent vectors in the basis that describes the vector.

Tensor notation incorporates super- and sub-scripting to distinguish covariance and contravariance, respectively, and how those two notions relate to variance and basis transformation. This said, the reader can generally infer the appropriate variance or vector orientations implicitly from ordinary matrix subscript notation, and in some of the literature this suffices, such that often it is not belabored to clarify if a space ever transforms coordinates simply by implementing or omitting the use of superscripting. The TTL does not implement script orientation to indicate variance, it entrusts that the client can manage variance and uses index binding responsibly.

2.2.1.3 Tensor notation

Tensors can be described under various notations. The "standard" or most formal is Ricci notation, however, Einstein notation is more common in practice (and physics) as it reduces much of the overhead of the more formal Ricci notation by removing e.g. summation notation (and Leibniz notation in the case of gradients) as the intended operations are typically contextually evident.

In tensor notation, the variance of the vector on a given index is indicated via the index's position as either super or subscript. Subscripting or lower indicial notation indicates covariance (i.e. row orientation), and upper or superscripted notation reflects contravariance (i.e. column orientation), e.g., for a given rank 1 tensor v, i.e. vector v, if it is a (1,0) tensor, it would be

$$v^i = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

And a (0,1) vector

$$v_i = [v_1 \quad v_2 \quad v_3]$$

The variance of the index is used to indicate different operations between tensor objects, however, transposing tensors is quite simple: rank 1 tensors exchange subscript for superscript and vice versa via the metric tensor (cf §2.2.1.4.5)

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \left(v^i\right) \rightarrow \left(v^i\right)^T = v_i = [v_1 \quad v_2 \quad v_3]$$

If a tensor is rank 2 or greater, indicial order is exchanged, e.g. for a rank 2 tensor, $\left(A_i^j\right)^T = A_j^i$, or also commonly as ${A_{ij}}^T = A_{ji}$.

However, the principal utility of tensor objects is not as a data structure per se but as a means of dictating operations between member arrays and other tensor objects through index regular expressions in a way often impossible in linear algebra.

2.2.1.4 Tensor operations

In the same manner that linear algebra defines matrices and vectors through superposition, i.e. additivity (i.e. $f(x+y) = f(x) + f(y)$) and first degree homogeneity (i.e. $f(\alpha x) = \alpha\big(f(x)\big)$), and guarantees certain behaviors thereby, so do tensors guarantee superposition. As such, tensors are capable of all the same operations the reader is familiar with in scalars, vectors, and matrices (unsurprisingly, as tensors encompass all those objects), and some additional ones, and typically much more directly and simply.

Vector and matrix addition via Einstein notation (sometimes called ‘tensor’ or ‘indicial’ notation) is, in the case of a vector (here a (1,0) tensor but the principle is identical for a (0,1) tensor), simply

$$c_i = a_i + b_i$$

and

$$c_{ij} = a_{ij} + b_{ij}$$

for a matrix (any variety of rank 2 tensor – (2,0), (1,1), or (0,2)). Similarly for scaling,

$$c_i(\alpha) = \alpha c^i$$

Of note, superposition of dual vectors is marginally but discernibly different from superposition of vectors in a vector space: additivity is no longer $f(\vec{x}) + f(\vec{y}) = f(\vec{x} + \vec{y})$ but rather $f(\vec{x}) + g(\vec{x}) = (f + g)(\vec{x})$, and homogeneity becomes $(\alpha f)(\vec{x}) = \alpha\big(f(\vec{x})\big)$ instead of $f(\alpha\vec{x}) = \alpha\big(f(\vec{x})\big)$.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} [d \quad e \quad f] = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}$$

The manner in which tensors manipulate indices indicates the operation intended. Operations within or between tensors which share indices will produce scalars via summation of the products and reduce the rank of the tensor in tensor contraction or the inner product, whereas operations between tensors which do not share indices generate higher rank tensors in a tensor or outer product. Indices which occur only once in a tensor or in a tensor expression are referred to as "free" indices, whereas an index which occurs twice either within a tensor or a tensor expression is a "bound" index. The binding or freeness of the indicated indices dictates the operations occurring in tensor expressions.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} [d \quad e \quad f] = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}$$

Figure 3. Tensor Product of 2 rank 1 tensors in 3D space, $S^i T_k = U^i_k$

2.2.1.4.1 Tensor Product

The tensor product, sometimes referred to as the outer product and occasionally denoted with $\otimes$, develops a single larger tensor object from two smaller tensors across free indices. Its elements are the product of each of the components of the first tensor with each of the components of the second tensor (cf. **Error! Reference source not found.**). Given two tensors, S^i_j and T^k_l, of rank (m_1, n_1) and (m_2, n_2), the outer product, $S^i_j T^k_l$, would form a $(m_1 + m_2, n_1 + n_2)$ tensor, U^{ik}_{jl}. The tensor product is not commutative, i.e. $S^i_j T^k_l \neq T^k_l S^i_j$. An example of a tensor outer product in TTL is given in Figure 4.

```
template<int D, typename T>
__global__
void OPKernelTTL(T *dC, T *dA, T *dB){
    ttl::Tensor<1,D,T*> A { dA };
    ttl::Tensor<1,D,T*> B { dB };
    ttl::Tensor<2,D,T*> Z { dC ]};
    C(i,j) = A(i) * B(j);
}
```

Figure 4. Templatized TTL Kernel of an outer product of two rank 1 tensors.

The outer product is computationally considerably more expensive than other rank-altering tensor operations, as it requires considerably more write operations to increase the rank of the tensor object, and in the same manner that looping non-linearly increases computational complexity, so do tensor products.

2.2.1.4.2 Inner Product

The inner product produces a scalar via summation across bound indices. In tensor notation, summation occurs whenever a co-variant vector is paired with a contra-variant vector on a bound index, which also reduces the tensor rank by two for each shared index, e.g. if there were some second rank tensors S_j^i and T_k^j, their product $S_j^i T_k^j$ would be U_k^i, while the jth index would map to a scalar c,

```
template<int D, typename T>
__global__
void IPKernelTTL(T *C, T* dA, T *dB){
    ttl::Tensor<1,D,T*> A { dA };
    ttl::Tensor<1,D,T*> B { dB };
    auto C = A(i) * B(i);
}
```

Figure 5. Templatized TTL kernel of an inner product of two rank 1 tensors.

$$\sum_{j=0}^{n} S_j T^j = S_j T^j = s_1 t_1 + s_2 t_2 + s_3 t_3 = c$$

The lefthand expression with the sigma notation is the more formal Ricci formation, the expression to the right of that is Einstein notation. This example assumes the Euclidean space but as the index does not necessitate dimensionality as explained in §2.2.1, if index i were four dimensional,

$$\sum_{i=0}^{n} S_j T^j = s_1 t_1 + s_2 t_2 + s_3 t_3 + s_4 t_4$$

Of note, the choice of bound index is in some sense arbitrary, as any index can be equated to another (with identical dimensionality) via a dummy index to bind them.

Inner products are cheaper to compute as the resultant tensors are always reduced order, and it has excellent locality (cf. §1.4.1.0), which is not guaranteed for tensor products. An example of the TTL implementing an inner product is given in Figure 5.

2.2.1.4.3 Contraction

Tensor contraction differs from the other tensor operations as it doesn't define an operation between tensors but within one, although it is related to the inner product. It's a one-tensor operation that produces a scalar tensor of reduced rank, (p-1,q-1). If there were some second rank tensor A_j^i, we could equate indices i and j through a dummy index u, and perform the contraction as

$$\sum_{i=0}^{n} A_i^i = A_i^i = a_1 a_1 + a_2 a_2 + a_3 a_3 = c$$

2.2.1.4.4 Levi-Civita Symbol and the Permutation Tensor

We construct an ordinary vector space through basis vectors, e_n, however, basis vectors can only build a vector space and only ever output vectors from that basis; to construct a dual vector space which dual vectors inhabit and which outputs scalars, we need an equivalent dual vector object to basis vectors (and interestingly, every vector space has a corresponding dual vector space). That object is the Levi-Civita symbol, sometimes called the permutation or antisymmetric tensor, indicated with ϵ and some number of alphanumeric indices spanning the space (typically ϵ_{ijk} for the cartesian space but potentially infinite).

In the same manner that for some vector $\vec{v}$ in the canonical *ijk* cartesian space, defined by the cartesian basis vectors e_i, e_j, and e_k, the vector product $\vec{v}e_i$ will produce the components of $\vec{v}$ in the e_i direction. Similarly, the Levi-Civitia symbol maps a vector to a scalar based on the basis dual vector performing the mapping. So for $\vec{v}$, mapping via the dual basis ϵ^i would produce $\vec{v}(\epsilon^i) = v^i$, which is no longer a vector and instead a scalar. $\epsilon^{ij}(\vec{v})$ would produce a dual vector with two scalar elements, $\epsilon^{ij}(\vec{v}) = v^{ij}$, etc.

In this regard Levi-Civita is related to the Kronecker delta piecewise function $\delta_{ij} = \{0\ if\ i \neq j, 1\ if\ i = j\}$) explicitly, as $\epsilon^i(e_j) = \delta_{ij}$, but also by analogy. Contrast the Kronecker delta with the Levi-Civita symbol, for which in the latter, mapping corresponds to a piecewise function dependent on the order of indices. Levi-Civita defined in 3 dimensions is

$$\epsilon_{ijk} = \begin{Bmatrix} +1 & if\ (i,j,k) = (1,2,3),(2,3,1),(3,1,2) \\ -1 & if\ (i,j,k) = (3,2,1),(2,1,3),(1,3,2) \\ 0 & if\ i = j, j = k, or\ k = i \end{Bmatrix}$$

This ordering pattern extends to any dimensionality, even infinitely, and the pattern of index ordering, referred to as cyclic permutation (as the ordering can be seen as a modulo arithmetic which can process clockwise or anti-clockwise), similarly extends. As long as every index is unique and the procession to the modulo is in ascendant order (even), the permutation tensor is one; if every index is unique and the procession in descendent order to the origin from the modulo (odd), it is negative one; all other permutations require repetition of at least one index and map to zero.

2.2.1.4.5 The Metric Tensor and Raising and Lowering Indices

The metric tensor is a (0,2) tensor whose principal utility is as a map for vectors into dual vectors and by its inverse, duals into vectors. Altering the variance of a tensor index is unimaginatively referred to as raising or lowering the index, and doing so requires mapping via the metric tensor.

The metric tensor, often denoted with g (not to be confused with the gravitational constant) can produce scalars from two one-forms or two vectors. In contrast to a one-form, it is a symmetric bilinear map, i.e. it

accepts two vectors or vector spaces and maps them to a scalar field (symmetry indicates the scalar output is order agnostic, i.e. $g(\vec{v},\vec{w}) = g(\vec{w},\vec{v})$). However, it also has the special property that, if only one vector is mapped to the dual, $g(\vec{v}, \cdot)$, the absence of the second vector simply maps the vector to a one form, $\tilde{v}$, and conversely, there is an inverse to the metric tensor, g^{-1}, which maps a one-form to a vector, $g^{-1}(\cdot, \tilde{v}) = \vec{v}$.

2.2.2 Linear Solvers

With enough effort and the right perspective, many computational questions can be reduced to a system of equations of the form $Ax = b$. Linear solvers suss out x from $x = A^{-1}b$, and writing a generalizable, stable algorithm to determine x without directly solving A^{-1} is the principle task of writing a linear solver (it's why these techniques are called linear solvers and not matrix inverters).

While $A^{-1}b = x$ is mathematically true, this is virtually never what is implemented in a linear solver. Linear solvers eschew explicitly calculating the inverse, instead factorizing the matrix such that some vector x can be calculated indirectly (typically via backward or forward substitution of the matrix from factorized triangular matrices). This is generally much more computationally tractable while minimizing memory requirements, more numerically stable, and ultimately, more germane, as we generally care for the result x and not the inverse itself.

2.2.2.1 QR Factorization

QR factorization decomposes an NxN non-singular matrix into two matrices, an orthonormal basis and a moiety that holds the system dynamics in an upper triangular matrix (triangular matrices are discussed in detail in §2.2.2.4 LU Factorization), such that $A = QR$. Stated as directly as possible, applying Gram-Schmidt orthogonalization to a square non-singular matrix produces one half the QR factorization (the Q) and the upper triangular matrix is produced by matrix multiplying the transpose of Q with the original matrix A.

Developing an orthonormal basis from a matrix A via modified Gram-Schmidt simply requires subtracting the previously orthonormalized contravariant vectors, q_{n-k}, from the contravariant vector of current examination, q_n, and taking the L_2 norm of the result (or, for the initial contravariant vector, simply taking the L_2 norm). E.g.

$$q_1 = \frac{a_1}{\|a_1\|}$$

from which then q_2 can be calculated as

$$q_2 = \frac{a_2 - q_1^T a_2 q_1}{\|a_2 - q_1^T a_2 q_1\|}$$

and q_3 can be calculated as

$$q_3 = \frac{(a_3 - q_1^T a_3 q_1 - q_2^T a_3 q_2)}{\|a_3 - q_1^T a_3 q_1 - q_2^T a_3 q_2\|}$$

all the way to the nth q.

As all orthonormal matrices hold the property that $Q^T = Q^{-1}$ (and we required that the original A matrix be non-singular i.e. that all contravariant vectors are linearly independent), we can compute R from

$$A = QR \rightarrow Q^{-1}A = R \rightarrow Q^T A = R$$

Given that R is upper triangular (cf. §2.2.2.4), there will be a single expression in the nth equation

$$R_{nn}x_n = b_n \rightarrow x_n = \frac{b_n}{R_n}$$

Back substitution up the upper triangular matrix allows complete resolution of the system. In implementation, this can be done recursively as there is a distinct base case, however, in practice, many popularly utilized linear solver libraries such as LAPACK eschew a recursive solution, despite implementations that provide performance improvements through it [9], as large N matrices require expanding the stack considerably and can blow out the main memory [10].

In numerical implementations, QR factorization is rarely implemented with Gram-Schmidt but rather via the Householder reflection algorithm (sometimes called the Householder triangularization algorithm) due to its superior numerical stability. The general premise derives from the fact that a vector can be reflected such that all coordinates but one are removed. In implementation this is reflected by transforming the matrix A with a series of lefthand permutations to successively develop an upper R triangular matrix

$$A = \begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix} \rightarrow Q_1 A = \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & x & x \end{bmatrix} \rightarrow Q_1 Q_2 A = \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & 0 & x \end{bmatrix}$$

where each successive Q_k leaves the first $k-1$ rows and columns unaltered and ends up of the form

$$Q_k = \begin{bmatrix} I_{k-1} & 0 \\ 0 & D \end{bmatrix}$$

where D captures the dynamics to perform the transformation in non-zeroed elements, such that

$$Dx = \begin{bmatrix} \|x\| \\ 0 \\ 0 \end{bmatrix} = \|x\| e_1$$

Where D is generally calculated from

$$D = I - \frac{2vv^T}{v^T v}$$

and

$$v = \|x\| e_1 - x$$

It is noteworthy that QR factorization can be generalized to MxN non-square matrices, however, this results in redundant sparsification of the lower (m-n) rows of the R matrix where all entries will be zero, a reflection of the overdetermination of the system due to the linearly dependent dimensions.

QR factorization operates in $O(4/3\, n^3)$ time, which, as discussed in §2.2.2.4, is about half as fast as LU factorization, however, it is considered stable under all conditions, whereas additional work may be needed in LU factorization to guarantee stability (cf. §2.2.2.5 on pivoting for stability).

2.2.2.2 Cholesky Factorization

Cholesky factorization is a specialization of gaussian elimination which decomposes a positive semi-definite matrix into two moieties, a lower triangular matrix, and the conjugate transpose of that same matrix, which is both exceptionally cheap in memory, as only one matrix need be stored and indicial artifice can be used to mimic the matrix transpose, and cheap to compute as the symmetry (expressed below) requires only one matrix be calculated.

By a positive semi-definite matrix it is meant that for some matrix A, the scalar produced from $x^T Ax \geq 0 \forall x$. Determining that a matrix is positive semi-definite is possible by a few means – one is to examine the determinants, however, this steps back into the problem Cholesky factorization (and virtually every factorization method) attempts to avoid. It's generally faster to examine the eigenvalues, which, if all positive, guarantees that the system cannot transform any vector except positively.

Computing the Cholesky factorization is comparatively cheap. It fundamentally derives from the first pivot – for some 3x3 matrix A (i.e. a (1,1) tensor of dimension 3)

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = LL^T = \begin{bmatrix} l_{00} & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} l_{00} & l_{10} & l_{20} \\ 0 & l_{11} & l_{21} \\ 0 & 0 & l_{22} \end{bmatrix}$$

$$= \begin{bmatrix} l_{00}^2 & l_{10}l_{00} & l_{20}l_{00} \\ l_{10}l_{00} & l_{10}^2 + l_{11}^2 & l_{20}l_{10} + l_{21}l_{11} \\ l_{20}l_{00} & l_{20}l_{10} + l_{21}l_{11} & l_{20}^2 + l_{21}^2 + l_{22}^2 \end{bmatrix}$$

The first pivot is simply the root of the unfactored matrix, and that root is a scalar for all first equation terms/the first covariant vector; these terms can then be substituted to solve all other terms, and the symmetry of the matrix's required positive-definiteness means approximately only a bit more than half the terms need to be directly calculated (due to the diagonal). Moreover, backward calculation isn't strictly necessary, the symmetry of Cholesky compatible systems lends itself to two patterns for calculating element values that can be used to calculate any pivot,

$$l_{kk} = \sqrt{a_{kk} - \sum_{j=0}^{k-1} (l_{kj})^2}$$

and then from the pivot elements, non-pivot elements can be calculated from

$$l_{ik} = \frac{1}{l_{kk}} \sqrt{a_{ik} - \sum_{j=0}^{k-1} l_{ij} l_{kj}}$$

When applicable, Cholesky factorization is a fast routine, operating in $O(n^3/3)$ compute time, which is approximately twice as fast as LU factorization, and it also has exceptional stability. However, the speedups derive from the constraints placed on which tensors can be Cholesky factorized, and the requirement to be positive semi-definite limits the number of instances where it can be invoked. The requirement for symmetry is generally easier to satisfy in systems that are governed by some kind of conservation law as that generates such a symmetric matrix, and so many physical systems can implement Cholesky factorization, however, the additional work of examining whether a matrix is Cholesky factorizable reduces the performance considerably and so the technique is not generally implemented as a solver per se but generally must be explicitly invoked, typically when the possibility of Cholesky factorization is known a priori from knowledge of the kind of system being examined.

2.2.2.3 LDL Factorization

LDL factorization derives from Cholesky factorization, but is even faster, as it isn't avoids the unnecessary calculation of square roots for any of the elements [11]. Rather than factorizing as $A = LL^T$, A is factorized

as $A = LDL^T$, where D is a diagonal matrix, and L is unitary along the primary diagonal, so the equation becomes

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = LDL^T = \begin{bmatrix} 1 & 0 & 0 \\ l_{10} & 1 & 0 \\ l_{20} & l_{21} & 1 \end{bmatrix} \begin{bmatrix} d_{00} & 0 & 0 \\ 0 & d_{11} & 0 \\ 0 & 0 & d_{22} \end{bmatrix} \begin{bmatrix} 1 & l_{10} & l_{20} \\ 0 & 1 & l_{21} \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} d_{00} & d_{00}l_{10} & d_{00}l_{20} \\ d_{00}l_{10} & d_{00}(l_{10})^2 + d_{11} & d_{00}l_{10}l_{20} + d_{11}l_{21} \\ d_{00}l_{20} & d_{00}l_{10}l_{20} + d_{11}l_{21} & d_{00}(l_{20})^2 + d_{11}(l_{21})^2 + d_{22} \end{bmatrix}$$

from which the diagonal components can be calculated from

$$d_i = a_{ii} - \sum_{k=0}^{i-1} (l_{ik})^2 d_k$$

and the other elements can be calculated from

$$l_{ij} = \frac{1}{d_i}\left(a_{ij} - \sum_{k=0}^{i-1} d_k l_{ik} l_{jk} \right)$$

2.2.2.4 LU Factorization

LU factorization, oftentimes referred to as LU decomposition, is a way of factoring or decomposing any given matrix into constituent triangular matrices, an upper and lower moiety, where the product of those two moieties is the original un-factorized matrix. LU factorization is one of if not the most common strategies for linear solvers to compute the quintessential numerical analysis problem, $Ax = b$, as the LU factorization is guaranteed to exist, can be functionally guaranteed to be stable if the matrix is pre-conditioned with a permutation matrix, and it can be solved in $O(^2/_3\, n^3)$ FLOPs for an N sized square matrix, which is 2 times faster than alternative methods like QR factorization, which requires $O(^4/_3\, N^3)$ FLOPs, at the expense of stability (although LU stability is not often problematic in practice) [12]. This is expressed in its most generalized form as

$$A = L * U$$

and interestingly, the LU and Cholesky factorization are related – if $U = L^T$, the two are identical.

A triangular matrix is a special class of a square matrix. A square matrix is lower triangular if all the entries above the main diagonal are zero, while an upper triangular matrix is the reverse: a square matrix where all the entries below the main diagonal are zero. A diagonal matrix is a special subclass of triangular matrices which is both upper and lower: the identity matrix is an excellent example of this, as one might

expect from its various and important properties. It is simultaneously square and upper and lower triangular. Matrix equations with triangular matrices are writ large in numerical analysis because they are much easier to solve, as at some point in the matrix there is a simple expression, $X = A$.

The LU decomposition algorithm writes an invertible NxN matrix as the product of a lower triangular matrix L and an upper triangular matrix U if and only if all its leading principal minors are non-zero. This could be extrapolated to an MxN matrix, however, for exemplary purposes the 3x3 matrix is the simplest non-elementary example of LU factorization. From a given matrix A, the LU factorization is

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{bmatrix}$$

The lower moiety is unit valued (i.e. the major diagonal is exclusively ones), resulting in

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{10} & 1 & 0 \\ L_{20} & L_{21} & 1 \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{bmatrix}$$

Which, if multiplied out resolves explicitly to

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} U_{00} & U_{01} & U_{02} \\ L_{10}U_{00} & L_{10}U_{01} + U_{11} & L_{10}U_{02} + U_{12} \\ L_{20}U_{00} & L_{20}U_{01} + L_{21}U_{11} & L_{20}U_{02} + L_{21}U_{12} + U_{22} \end{bmatrix}$$

Of particular importance to the reader in this last representation is that it clearly shows how, structurally, the L matrix preserves the scaling needed between rows to achieve the triangular structure of the U matrix, while the U matrix preserves the matrix dynamics of the original A matrix. Notice that in row 1, all elements are defined exclusively in terms of the U matrix, whereas row 2 is the U matrix scaled by L, etc. etc. For example,

$$\begin{bmatrix} 1 & 1 & 1 \\ 4 & 6 & 8 \\ 2 & 2 & 5 \end{bmatrix} => R_2 - 4R_1, R_3 - 2R_1 => \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 4 \\ 0 & 0 & 3 \end{bmatrix}$$

In most factorizations, an additional step would be needed to zero out the third row, however, this matrix was specially selected to add to the discussion in §2.2.2.5.

Examining more closely, LU factorization is gaussian elimination in which it isn't required that the solution matrix be in row reduced echelon form and where the steps of the elimination are preserved in the lower triangular matrix. As this example shows, the non-diagonal values of L are simply the values needed to do the gaussian elimination via a lefthand permutation matrix multiplication: in this example, nothing needed to be done to row 0 for gaussian elimination. Row 2 of column 1 was zeroed by subtracting row 1 scaled by 4 from row 2, and to preserve the matrix behavior, that scaling factor had to be distributed to the other

columns, providing the upper matrix values for columns 2 and 3 of row 2. Row 3 is zeroed by subtracting 2 of row 1. Noting that zeroing row 3 in the first column also zeroed column 2 of row 3, no input from row 2 is necessary, and the lower matrix instead has a zero in that position. Those are the coefficients of the L matrix. The U matrix simply stores the data needed to reconstitute the original matrix from those scaling coefficients. [13]

2.2.2.5 Pivoting

There is an intrinsic instability to LU factorization if the matrix is not initially well-conditioned. By well-conditioned it is meant that if a pivot is ever exactly zero, some value that is close to zero, or near machine precision in floating point systems, an unsophisticated LU factorization will fail. Examine the following matrix, identical to the previous example except rows 2 and 3 were exchanged, i.e., examine a matrix with a known LU factorization:

$$\begin{bmatrix}1 & 1 & 1\\ 2 & 2 & 5\\ 4 & 6 & 8\end{bmatrix} \rightarrow R_2 - 2R_1, R_3 - 4R_1 \rightarrow \begin{bmatrix}1 & 1 & 1\\ 0 & 0 & 3\\ 0 & 2 & 4\end{bmatrix}$$

There is no means by which the second row in this instance could be subtracted from the third to zero out the last row, and yet having done this in the previous equation, we know the factorization exists. This problem is general for all matrices if ever a pivot is zero or, in numerical scenarios, if the pivot is nearly zero. A human would recognize that this factorization is completed by simply exchanging rows 2 and 3, however, a naïve LU algorithm will simply fail. For this reason, most LU implementations left multiply by a permutation matrix, P, such that PA = LU, so that each row is guaranteed or nearly always guaranteed to prevent prematurely zeroing out, and this pivoting strategy is sometimes referred to as LUP.

In this instance, the permutation matrix would be

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 0 & 1\\ 0 & 1 & 0\end{bmatrix}\begin{bmatrix}1 & 1 & 1\\ 2 & 2 & 5\\ 4 & 6 & 8\end{bmatrix} = \begin{bmatrix}1 & 1 & 1\\ 4 & 6 & 8\\ 2 & 2 & 5\end{bmatrix} = \begin{bmatrix}1 & 0 & 0\\ 4 & 1 & 0\\ 2 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 1 & 1\\ 0 & 2 & 4\\ 0 & 0 & 3\end{bmatrix}$$

Modification of the matrix by exchanging rows leads to stable solutions for most square non-singular matrices and complete modification of rows and columns ensures success for all non-singular matrices [12]. An algorithm which exchanges only the rows is referred to as partial pivoting, one which permutes all rows and columns is complete or full pivoting, and lastly there is a hybrid permutative strategy called rook pivoting. More formally, this whole schema of pivoting can be reduced such that partial pivoting is synonymous with left permutation multiplication, complete pivoting is synonymous with left and right permutation multiplication, and rook pivoting employs a search algorithm to minimize whether left or right permutation is optimal at each pivot. An interesting note, from the perspective of memory implementation of an LU

algorithm, the permutation matrix only ever needs to be a one dimensional vector (in the same manner that any matrix [or tensor] can be vectorized into one dimension).

Partial pivoting rearranges rows such that the largest entry of the first column of the matrix is row swapped to be the first row, the largest entry of the second column from the second row and below is swapped to become the second row, etc. etc., in this fashion, for each column, until the final column, which is trivial. Alternatively stated, for a given n × n matrix, the target is the largest value across all rows of the first column; at step k of the elimination, the largest of the n − (k + 1) subdiagonal entries of column k is selected, e.g. if examining column 2 of a 4x4 matrix, rows 2-4 would be scanned for their largest value. This operation costs O(nk) examinations for each step of the examination, meaning an n×n matrix requires $O(n^2)$ examinations. This entry is then moved into the pivot position A_{kk} on the diagonal of the matrix by multiplying B on the left with the permutation matrix P. From there the LU factorization algorithm can be implemented.

In the following NxN matrix, bolded elements indicate the array of N values that are compared, with alpha variables indicating that element is the largest value within that column, and the subscript indicating its value relative to other column-max values, e.g. $\boldsymbol{\alpha_2 > \alpha_3}$. Stepping through the reduction

$$\begin{bmatrix} \boldsymbol{x} & x & x & x \\ \boldsymbol{x} & x & \alpha_1 & x \\ \boldsymbol{\alpha_2} & x & x & \alpha_3 \\ \boldsymbol{x} & \alpha_4 & x & x \end{bmatrix} = \begin{bmatrix} \alpha_2 & x & x & \alpha_3 \\ 0 & \boldsymbol{x} & \alpha_1 & x \\ 0 & \boldsymbol{x} & x & x \\ 0 & \boldsymbol{\alpha_4} & x & x \end{bmatrix} = \begin{bmatrix} \alpha_2 & x & x & \alpha_3 \\ 0 & \alpha_4 & x & x \\ 0 & 0 & \boldsymbol{x} & x \\ 0 & 0 & \boldsymbol{\alpha_1} & x \end{bmatrix} = \begin{bmatrix} \alpha_2 & x & x & \alpha_3 \\ 0 & \alpha_4 & x & x \\ 0 & 0 & \alpha_1 & x \\ 0 & 0 & 0 & x \end{bmatrix}$$

From the implementation perspective, the permutation matrix is virtually never implemented as indicated above because that could require moving significant amounts of data for high rank tensors; rather, a single vector records how rows should have been switched and the indexing to the data wherever it lays in memory is adjusted, e.g. in the example above there would be a vector based on the original matrix, [3,4,2,1], indicating to first manipulate the 3rd row, then the 4th, etc. etc.

Complete pivoting requires searching for the largest value in the entire submatrix instead of just the next subcolumn. It is "full" in the sense that the pivot for each row is the largest possible pivot that matrix could use of all relevant entries. This requires $O(n - k)^2$ comparisons just to locate the element for a single pivot, which in a square nxn matrix requires $O(n^3)$ comparisons in total. For an n × n matrix A, the first step is to scan n rows and n columns for the largest value. Once located, this entry is then permuted into the next diagonal pivot position of the matrix. So in the first step the entry is permuted into the (1, 1) position of matrix A. Rows are exchanged exactly as in partial pivoting, by multiplying A on the left with a permutation matrix P. To interchange columns, A is multiplied on the right by another permutation matrix Q. The matrix product PAQ interchanges rows and columns accordingly so that the largest entry in the matrix is in the

(1,1) position of A. With complete pivoting, the general equation for L is the same as for partial pivoting, but the equation for U changes slightly. As in the previous example, bold indicates an element that could potentially be a pivot, but the subscripting now indicates the four largest values within the matrix

$$\begin{bmatrix} \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{\alpha_1} & \boldsymbol{x} \\ \boldsymbol{\alpha_2} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{\alpha_3} \\ \boldsymbol{x} & \boldsymbol{\alpha_4} & \boldsymbol{x} & \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \alpha_1 & x & x & x \\ 0 & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ 0 & \boldsymbol{x} & \boldsymbol{\alpha_2} & \boldsymbol{\alpha_3} \\ 0 & \boldsymbol{\alpha_4} & \boldsymbol{x} & \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \alpha_1 & x & x & x \\ 0 & \alpha_2 & x & \alpha_3 \\ 0 & 0 & \boldsymbol{x} & \boldsymbol{x} \\ 0 & 0 & \boldsymbol{\alpha_4} & \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \alpha_1 & x & x & x \\ 0 & \alpha_2 & x & \alpha_3 \\ 0 & 0 & \alpha_4 & x \\ 0 & 0 & 0 & x \end{bmatrix}$$

Rook Pivoting is a hybrid implementation, offering the same big O performance as partial pivoting but the stability of complete pivoting. In this pivoting strategy, the selection of the optimal pivot is eponymously derived from how a rook in chess moves – pivots can be selected from elements within the same row or column as the current pivot position, but not by examining the entire submatrix, saving considerable time in large matrices. [14]

$$\begin{bmatrix} \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \boldsymbol{x} & x & \alpha_1 & x \\ \boldsymbol{\alpha_2} & x & x & \alpha_3 \\ \boldsymbol{x} & \alpha_4 & x & x \end{bmatrix} = \begin{bmatrix} \alpha_2 & x & x & \alpha_3 \\ 0 & \boldsymbol{x} & \boldsymbol{\alpha_1} & \boldsymbol{x} \\ 0 & \boldsymbol{x} & x & x \\ 0 & \boldsymbol{\alpha_4} & x & x \end{bmatrix} = \begin{bmatrix} \alpha_2 & x & x & \alpha_3 \\ 0 & \alpha_1 & x & x \\ 0 & 0 & \boldsymbol{x} & \boldsymbol{x} \\ 0 & 0 & \boldsymbol{\alpha_4} & x \end{bmatrix} = \begin{bmatrix} \alpha_2 & x & x & \alpha_3 \\ 0 & \alpha_1 & x & x \\ 0 & 0 & \alpha_4 & x \\ 0 & 0 & 0 & x \end{bmatrix}$$

Note that from the same initial matrix layout, three unique pivoting solutions emerged.

2.2.2.6 Solving from LU factorization

The triangular nature of all other factorization schemes is fundamental to solving them in the same manner as LU factorization . Solving a matrix from the LU factorization is trivial and avoids unnecessarily calculating determinants to produce an inverse. From the original equations $Ax = b$ and $A = LU$

$$Ax = b \rightarrow (LU)x = b \rightarrow L(Ux) = b$$

Noting that the product of Ux is just some vector y, we are left with a system of equations, $Ux = y$, and $Ly = b$. Solving for y via backward substitution of Ux enables the solution for b from Ly via forward substitution.

2.3 Computational Hardware

An examination of the hardware which enables all of this software is helpful, as in some sense, performance optimization operates on two planes: the first is the optimization of the algorithm itself as explored in comparing big O runtimes across the different solvers in §1.2.1. The other has to do with the physical limitations of the hardware. For example, there is a mathematical limit to LU factorization, at absolute best it performs at $O(\frac{2}{3}n^3)$, however, which of data structures or families of implementation are selected can significantly alter the performance based on the structure of the hardware. Considerable optimization is possible from

awareness of alternative implementation strategies of almost identical algorithms and even customization to the hardware. Today, some of this kind of optimization is implemented via the compiler or OS, however, implicit in this is the question of how well the compiler can optimize a particular code, a topic of discussion throughout 6.4.

2.3.1 CPUs and Host Hardware

As the reader might surmise from the names, GPUs are derived from CPUs – both integrate control units (CUs), arithmetic logic units (ALU), memory management units (MMU), etc., and so an understanding of that hardware in the CPU offers insight and disparity to their homologous GPU hardware implementation.

HPC today is fundamentally driven by different strategies of parallelization, and these strategies are taxonomically classified under Flynn's taxonomy. There are four families in Flynn's taxonomy (although contemporary architectures have evolved to the point there are many derivatives, such as the GPU): single instruction stream single data stream (SISD), which is all but extinct in modern computing environments (certainly within HPC); single instruction stream multiple data stream (SIMD), where a single instruction set can operate independently across separate data sets or streams. The related single instruction stream multiple threading (SIMT) is utilized in GPUs and discussed in §2.3.4; multiple instruction streams multiple data streams (MIMD), which is effectively the configuration of every modern multicore processor.

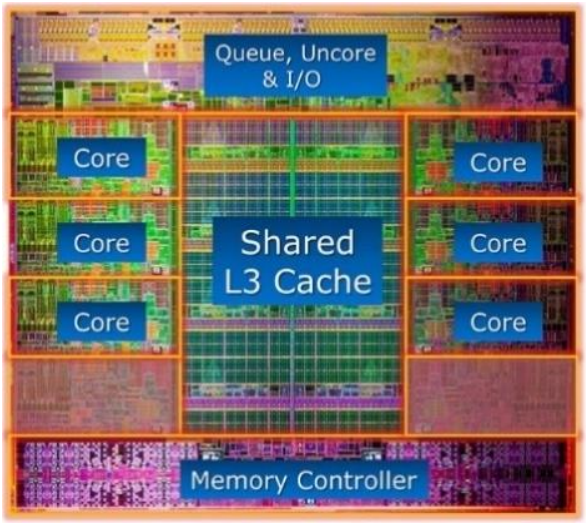


Figure 6. Intel i7 Die. Image Source: Intel

A CPU core is a single processing unit composed of the necessary constituent circuits to perform computation – the ALUs, CUs, registers, cache, etc. Modern CPUs are multicore (Figure 6), composed of conglomerates of cores, each separate core parallelizing potentially the same program or a separate program, concurrently. As the cores running different programs will hold different contexts, multicore CPUs are MIMD.

Layered atop the parallelization of the core is an additional parallelization via multi-threading (or in Intel, hyper-threading), in which multiple processes can be executed concurrently or in parallel via pre-emption and memory management within the core, where multiple tasks or processes swap in and out of context and hold compute resources. The benefit of multi-threading is that multiple tasks can be completed across multiple processing units within the same core, likely working on the same program, with context switching

during long tasks to reduce stalling - typically a cache miss (cf. §2.3.2.3) as data is fetched from somewhere slower in the memory hierarchy, as memory accessing is orders of magnitudes slower than e.g. computation (Figure 14). Allowing a thread to proceed while another loads effectively masks the latency of loading from distant memory banks. As will be discussed in §2.3.4.2.1, the scale of threading and ability to context swap quickly is one of the principal distinguishers in CPU and GPU processing.

2.3.1.1 Clock Generator

The clock is the circuit responsible for synchronizing the other circuits of the CPU. It generates the pulse waves that propagate through the CPU to synchronize the various circuits, and historically was the best indicator of processing speed – increasing pulse frequency allowed more and more computation to be done in a smaller and smaller window. It's desirable to have the clock synchronize such that each subunit completes its task across a single cycle, but not all tasks are performed in uniform time lengths, and so the clock is limited by the slowest circuit.

Today some performance improvements are directed at increasing clock frequency to leverage optimization strategies like ILP (Cf. §2.3.1.3.1) or extending instruction pipelines, but increasingly improvements rely on improved parallelization within the chip through concurrency techniques like hyperthreading and multi-core division, or alternative processing techniques like out-of-order execution (OoOE, cf. §2.3.1.3.3), etc. [15]. In a sense, GPUs represent the pinnacle of performance maximization through parallelization vis-à-vis flattening clock speeds [1].

2.3.1.2 Arithmetic Logic Unit

The arithmetic logic unit (ALU), sometimes referred to as the execution unit (EU), is in some sense the fundamental workhorse of the CPU and GPU, the only component that truly does 'work' in altering data – everything else is data management, prediction, storage, etc. A single modern CPU contains multiple ALUs within each core, but one of if not the defining feature of GPUs is that there are hundreds or thousands of ALUs, depending on the GPU, referred to as GPU cores (cf. §2.3.4.1, Figure 17).

The ALU is the circuit that performs the mathematical, logic, and bitwise operations that most imagine as the "core" of a computer, although it isn't the largest or most complex part of a CPU – that is reserved for the controller, sometimes referred to as the control unit (cf. §2.3.1.3). ALUs come in several "flavors" e.g. integer or floating-point ALUs, however, for simplicity, most refer to these myriad components as a conglomeration under the original title provided by John von Neumann, ALU, regardless of function.

There are two principal inputs to an ALU: the operand, which is the data being operated on, and a code indicating the operation to be performed (Figure 7). These codes are exceedingly simple and eponymously arithmetic: using add, subtract, increment, decrement, two's complement to generate the negative value, etc. It also includes logic gates for AND, OR, XOR, etc. functionality. All of these codes reflect the various transistor junction schemes which the ALU has embedded in the structure, although this paper will not digress further into an examination of those gates, how ALUs implement voltage arithmetic, the solid state physics of PN junctions, etc., as they are well beyond the scope of this work. The curious may find additional information in Nilsson & Riedel's *Electric Circuits, 10th Edition* or Hummel's *Electronic Properties of Materials* for additional exploration on transistor logic and semiconductor solid-state physics, respectively, as both were particularly useful to this author prior to this work.

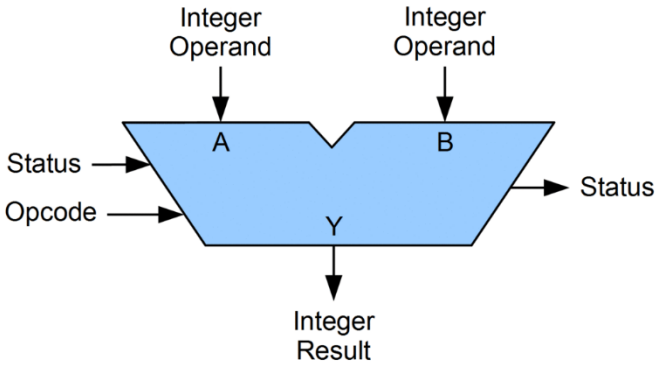


Figure 7. Symbolic ALU representation. Image Source: Wikimedia.

The ALU's output is the result of the operation on the operand. Most designs also have status inputs or outputs, often both, which communicates to other hardware information about a previous operation or the current one in order to enable out-of-order operation (cf. §2.3.1.3.3).

2.3.1.3 Control Unit

While the ALU provides the iconic "math" functionality of the CPU, in a certain sense it is very mechanical, accepting input, generating output. The control unit (CU) coordinates the flow of data and instructions into, out of, and between a processor's various sub-units, such as the ALU. If a program were an orchestra, the ALUs would be the musicians and the CU would be the conductor. The absence of complicated CUs in GPUs alters their performance optimization significantly.

In the same sense that the ALU refers to an aggregation of components/circuits (e.g. logic circuits and integer arithmetic) the controller also is an aggregation of sub-components, such as the instruction unit, which handles scheduling, or the retirement unit, which handles output from elsewhere in the instruction pipeline. CPUs are optimized around control flow and so the controller is a much larger and significantly more complicated circuit (Figure 6, Figure 17) as a portion of the CPU and dictates CPU behavior much more than the operations of the ALUs, including managing inputs to and outputs from the ALU because the CU owns the instruction decoder.

2.3.1.3.1 Instruction Pipeline

Instruction pipelines are a processing technique to significantly improve program run time by subdividing the steps of processing instructions via circuit specialization. Early, rudimentary processors were single-cycle, i.e. all the tasks of performing an instruction occurred across a single clock cycle (Figure 8), however, clock speed can be significantly improved with additional, specialized, independent circuits in the CPU to improve how data is processed, effectively parallelizing the instruction for each subdivision.

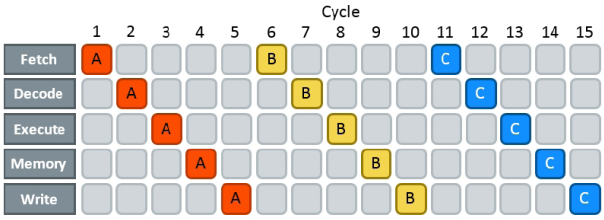


Figure 8. Single cycle pipeline. Image Source: Alex Shinsel, Intel

The simplest model for processing data in a pipeline occurs over five steps: instruction fetch (IF), which gathers the next instruction from the program pointer and assigns it to a register; instruction decoding (ID), i.e. determining what the instruction is and setting it up, which could involve sending data to the ALU or preparing a branch or jump point; execution (EX) of the instruction, often but not exclusively on the ALU; access or memory (MEM), where the appropriate operand in memory is grabbed; write back (WB), where the accessed operand is updated, completing the cycle and allowing the current instruction to be retired and permitting another instruction fetch.

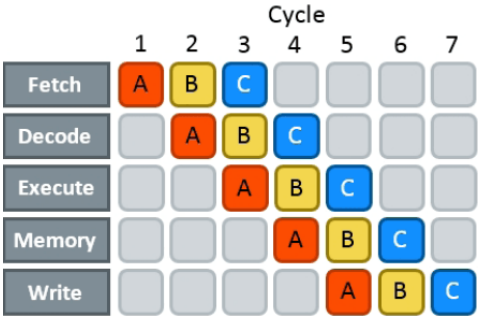


Figure 9. Instruction pipeline. Image Source: Alex Shinsel, Intel

Speeding up a pipeline requires either deepening or widening it. Deepening the pipeline requires additional processing circuit subunits to increase the number of steps needed to process a given instruction, subdividing the stages outlined in the previous paragraph further so each subdivided unit is simpler, faster, but also an additional parallelization of the instruction being processed. This allows each subunit to (ideally) perform its task independent of the other subunits, and therefore simultaneously, e.g., while Subunit A completes its task, Subunit B completes its task, which is to modify the previous product of Subunit A; simultaneously, Subunit C completes its task, a modification on Subunit B's output, etc. etc. (Figure 9). This is referred to as instruction level parallelism (ILP). Alternatively, the pipeline can be widened, i.e. multiple instructions can be fetched simultaneously (and then subdivided and executed as above), creating what are referred to as superscalar processors (Figure 10Figure 9).

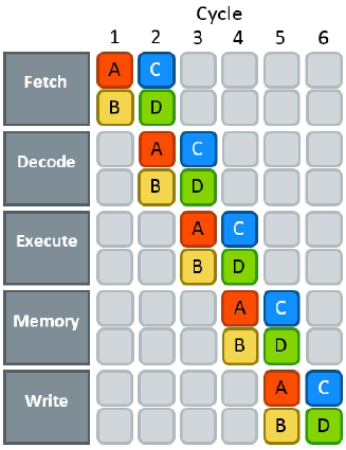


Figure 10. Superscalar processor. Image Source: Alex Shinsel, Intel

The model from the previous two paragraphs is simple and can be significantly discretized into smaller and smaller parcels – modern processors have instruction pipelines that can stretch to be dozens of sub-processes. Whereas the best a single cycle processor can perform is a single instruction per cycle,

Subdividing instructions makes the pipeline an assembly line in the execution of program instructions. The way in which tasks are subdivided depends on the specific instruction set architecture (ISA), and broadly varies depending on whether the CPU is a Reduced Instruction Set Computer (RISC) or Complex Instruction Set Computer (CISC). CISC ISAs allow assemblers to invoke suites of instructions through single invocations, i.e. there may be a single "add" instruction which encompasses loading data, performing the arithmetic, and then writing to memory, whereas a RISC ISA would require separate invocations to each.

While historically RISC architectures outperformed CISC [16] by improving ILP with simpler, flatter instructions (albeit in a somewhat apples-and-oranges comparison due to compiler optimization), most general purpose CPUs that remain CISC today (predominantly Intel chips), while outwardly appearing CISC driven at the level of the assembler, are RISC under the hood in what Intel refers to as "micro-ops", which are effectively just RISC equivalent subtasks, and recent research indicates that the performance differentiation of the two has merged (likely because of this somewhat vestigial distinction when Intel x86 architectures implement RISC-like micro-ops) [17].

As the reader might intuit, this compartmentalization of tasks significantly improves processing time the longer the pipeline, however, as the pipeline stretches, more and more work is done on a task that is not fully completed until the end of the pipeline, and if anything should disrupt this before completion, tasks ahead or behind become staggered, or even pointless work, and the longer the pipeline, the greater the cost for such mistakes, as increasing numbers of instructions become invalid, so there are diminishing returns to longer pipelines. These points of conflict are known as hazards, and their resolutions either require adaptive scheduling to avoid such conditions, stalling work to avoid race conditions (typically to resolve data hazards) or predictive estimation of the correct program pathing (to address control hazards), which, when incorrect, penalizes performance. In this regard architecture design is an optimization problem, maximizing pipeline length while minimizing the constraint function, hazards.

2.3.1.3.1.1 Hazards

Hazards are categorized in three ways: structural, data, and control. Structural hazards are intrinsic to hardware, whereas control hazards are generally intrinsic to logic in the software. Data hazards occur when software instructions conflict with how the hardware processes data and come in three varieties: read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR).

Structural hazards are in a sense hardware deficiencies, a result when some aspect of the integrated circuitry is not sufficient to properly balance a given load to process all instructions efficiently, e.g. in a simplified case, if there were only one register capable of writing back, no amount of parallelization of other tasks would overcome the bottleneck that all output would have to go through that register. Structural hazards sometimes reflect cost optimization, as it is cheaper to manufacture a chip with fewer resources, but other than manufacturing different hardware, there is no performance solution besides stalling.

$$r_1 = r_2 + r_3$$

$$r_3 = r_2 + r_4$$

Figure 11. RAW Hazard

RAW hazards are the second most commonly encountered hazard overall, occurring when an instruction needs a result that either hasn't been calculated or retrieved from memory yet. This can occur in pipelining when the prior instruction has only been partly processed through the pipeline even though the instruction with the hazard is executing after the preceding instruction. Figure 11 depicts a RAW hazard where r_4 depends on the result of r_3. This can be resolved by dynamic execution (§2.3. 1.3.3) or operand forwarding via the reservation station, wherein output from one pipeline stage is redirected from where it would be written to in the register file directly to the dependency where it's needed (the ALU input gate/latch of some dependent instruction) in a feed forward control scheme. To do so, the ALU result from the appropriate outputting EX/MEM register is fed back to the ALU input, and if the hardware determines the previous operation has written to the logical register (cf. §2.3.2.4) corresponding to the register source for the current ALU operation, control logic selects the forwarded result as the ALU input for the dependent instruction, rather than reading data from the register file.

$$r_3 = r_1 + r_2$$

$$r_3 = r_1 + r_4$$

Figure 12. WAW Hazard

Write after write (WAW) hazards occur when two instructions write to the same location, and the latter write precedes its antecedent, which leaves the antecedent in memory rather than the successor. This hazard is only possible in pipelines that write in more than one stage in the pipe or which let an instruction progress even when a previous instruction is stalled. It is not uncommon, but not as common as RAW hazards or control/branching hazards and limited to out-of-order execution pipelines.

A write after read (WAR) hazard is the inverse of a RAW hazard, where a preceding instruction needs data but a pursuing instruction will write to the location before reading occurs. This is the least common of the three (and impossible in in-order pipelines, although overwhelmingly pipelines today are out-of-order execution). This is depicted in Figure 13 where r_3 is needed to generate r_1, but if done out of order by the , a different value for r_3 will be used. Both WAW and WAR hazards can be resolved with register renaming and a reservation unit (cf. §2.3.2.4).

$$r_1 = r_2 + r_3$$

$$r_3 = r_2 + r_4$$

Figure 13. WAR Hazard

The last hazard, the most commonly encountered and in some regards most relevant to GPUs for scientific computation, are branch hazards. A branch occurs any time a program needs to migrate (jump) to a different, non-contiguous memory location, either due to conditionals directing to other instructions, loop statements directing to previous statements, or the invocation of new stack frames.

If there is no other technique for resolving a hazard such as forwarding, any hazard can be resolved with "bubbling", wherein no-op instructions are inserted while the hazard instruction is completed. Any time a section of the pipeline cannot complete its task within a single clock cycle, such a "bubble" is created in the pipeline at the stall, as no work is done, and all instructions behind that stage will similarly be delayed from the bubble moving up the pipeline, so the bubble is not cleared until it exits. This becomes particularly important with branching, as, if the pipeline proceeds and later determines it chose the wrong branch, the pipeline may need to be flushed, i.e. all instructions currently being processed must be stalled until they are out of the pipeline (and also clear the reorder buffer for incorrect instructions that were scheduled to be completed).

2.3.1.3.2 Branch Prediction

One of the reasons the CU in CPUs are so developed is to control the flow of branching points and determine which branch is likely. During the fetch operation, typically the next data from memory that is needed is adjacent in contiguous memory, however, this is not necessarily correct, e.g. when there is a branch. In many instances it cannot be known whether or not a jump is needed until examined at runtime by execution, which means either the pipeline needs to stall until the branch instruction is resolved, which wastes cycles, or it can speculate on which branch will be taken, and if correct, the pipeline continues unabated. However, an incorrect prediction requires any work done speculatively must be undone, and any instructions currently in the pipeline that belong to the incorrect branch must be flushed and those cycles were wasted as well. This is not too many wasted FLOPs in small pipelines, but in modern processors with large, complex, superscalar pipelines, this can mean 10-20 cycles are lost for each poor branch prediction to clear out the pipeline. There are also very ordinary operations that are susceptible to branch prediction which we don't ordinarily think of as "conditional" in the same sense as "if-else" but which structurally are, such as loops. As such, poor branch prediction can significantly impact performance and historically has, although interestingly, this is inversely related to the pipe length/extent of ILP and directly related to the cache's performance [18]. Today branch prediction can push upwards of 99% accuracy in CPUs, however, that relies on the complex circuitry of the CU to do so (and some in the literature argue it is not a solved problem because the remaining 1% are still quite expensive, the potential gains at least on par with those expected from developing improvements by other means [19]).

Branch prediction generally relies on the fact that whatever the branch has been doing is likely to continue – e.g. in a loop with even just three iterations, at any given examination of the branch it would be statistically likely to continue 2/3 of the time, and this only increases with the number of iterations.

CPU branch prediction utilizes either static or dynamic prediction: static prediction is determined at compile time, dynamic prediction at runtime. The simplest branch prediction is to assume a default path and sort out the validity of the branch once it's moved through the pipeline (typically that the branch is not taken for the statistical reason given above). There are considerably more complicated mechanisms for branch prediction: examining preceding states, computing likely branch decisions from statistical analysis of previous branch decisions, etc., but a discussion of this is beyond the scope of this work.

As discussed in §2.3.4.2, the control units of GPUs are not as sophisticated as CPUs (by design - they needn't be with latency hiding), but, as branch prediction falls under the purview of the control unit, any kind of branching can result in much slower processing on a GPU than a CPU. In particular this becomes important at the level of the warp as discussed in §2.3.4.2.1.

2.3.1.3.3 Dynamic Execution, Speculative Execution, and the Reservation Station

Dynamic execution, sometimes referred to as out-of-order execution, allows instructions to be re-ordered from their expected software sequence. One of the limitations of pipelining is that if an instruction stalls in it, no other instructions proceed, and, in something like a RAW hazard (cf. §2.3.1.3.1.1) where the result can't be forwarded, the pipeline stalls if two closely spaced instructions share a dependency, which is quite common. Executing out of order effectively requires ILP of the instruction decoding over at least two stages: the actual decoding, and then examining surrounding code for hazards.

In dynamic execution, instructions are loaded into a queue (the instruction buffer or reservation station) and dispatched such that those with a dependency are not pipelined until their dependency has progressed far enough that its result can either be forwarded or written to memory such that the depending instruction has that result available to it at the appropriate point in the pipeline. While early-on dependent instructions wait for their dependent results to propagate through the pipeline, a non-dependent later instruction can be launched ahead of them in the pipeline to be executed.

This does however mean that the dependency can determine the work done was incorrect, and in which case executed instructions would need to be undone. To avoid this, there is an additional memory buffering queue beyond the output that holds data before it is committed to cache or main memory, the Re-Order Buffer. It not only holds output after exiting the pipeline in case there has been a mistake from a dependency, but to make sure results are re-arranged to be committed to memory in the expected order. Typically

this is done in a first in first out (FIFO) stack, where instructions are added to the buffer when they are dispatched to the pipeline, and removed in the same order they are queued.

2.3.1.4 Vector and Scalar Processors, Vectorization

Processors have historically been developed as vector or scalar processors. Vector processors operate on arrays of data simultaneously using the same instruction in SIMD parallelization, whereas scalar processors operate on single data items. A vector processor is in a sense analogous to data what a pipeline is for instructions: rather than constantly decoding instructions to perform the same operation and fetching individual units of data to process alongside the instruction, a vector processor reads a single instruction and the vector processor executes the operation numerous times across multiple chunks of data, significantly saving time. This is not to be confused with the aforementioned superscaling processors from §2.3.1.3.1 depicted in Figure 10, where scalar units of data are processed across multiple, different execution units (hence, super, i.e. above, scalar), as there are superscalar vector processors, where multiple arrays of data are processed concurrently.

Most modern CPUs however are not strictly vector processors, as most instructions operate on scalars as well as vectors and unique instructions often focus on scalars, yet there are significant benefits to vector operations when possible, so instead they perform ““vectorization”, using a different ISA. Vectorization uses specially reserved registers of the CPU to perform vector processing. In standard x86 chips, this is done through the SSE and AVX ISAs, however, this creates a layer of abstraction between programming and invoking vectorization so that invocation then generally relies on the compiler, and often the heuristics are conservative and prevent vectorization [20]. One of the advantages of GPUs is that the compilers much more readily vectorize. [21]

2.3.2 Memory

While memory is technically a facet of hardware, it exists at the intersection of hardware and software in that nearly everything a programmer does is in some sense memory management, even though in certain high-level languages the explicit work like garbage collection of the heap is automated and obscured from the programmer. Many of the structures of CPU memory are exactly or homologously present in GPUs, and so a review of memory in a CPU environment is valuable context to its implementation in a GPU.

Accessing memory can be many orders of magnitude slower than processing data and one of the principal means of optimization is managing or limiting the number of transfers and data accesses needed in an algorithm. The expense of memory operations vis-a-vis compute operations is such a bottleneck that in some HPC literature, memory is treated synonymously with optimization [22], which is apparent when considering that even the fastest data transfer from off-chip main memory is orders of magnitude slower than operations within the chip. This disparity between modern compute times and the ability to load data from memory to be processed is sometimes referred to as the processor-memory performance gap, or the memory wall. A decade ago there was fear over general compute times failing to improve as disparity

between compute time and memory bandwidth to transfer data to be computed grew (in truth, the memory gap problem was never truly solved, it was just overcome with on-chip caching – cf. §2.3.2.3). As such a great deal of work has been done in optimizing how memory is utilized in CPUs, from how the scheduler uses branch prediction to switch context to do other work as memory is being loaded, to rearranging the order of instructions to improve locality (§2.3.1.3.3).

Modern memory is implemented as a hierarchy of different memory technologies, layered to meet different speed and space requirements. From Memory Systems: Cache, DRAM, and Disk: "a well implemented hierarchy allows a memory system to approach simultaneously the performance of the fastest component, the cost per bit of the cheapest component, and the energy consumption of the most energy-efficient component." The relative gains of effective hierarchical utilization are significant enough that benchmarking in settings where caching, register assignment, and memory allocation are more directly manipulable, as in the CUDA environment, is a topic in the literature unto itself [23].

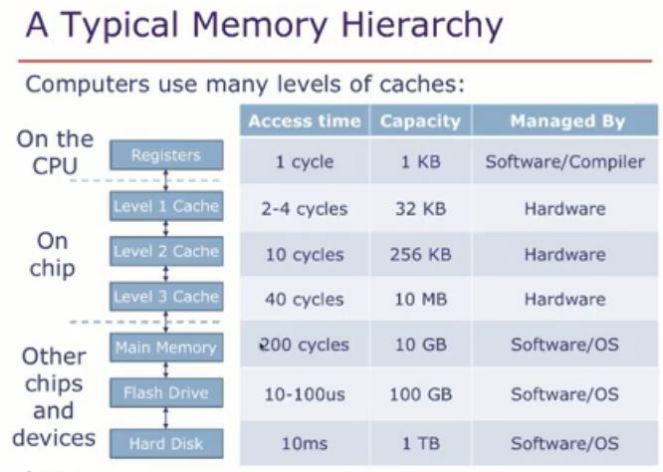


Figure 14. Example clock cycles for memory accesses. Image Source: Chris Terman, MIT Computation Structures

Memory broadly falls into one of three categories: on the chip, primary, and storage, which corresponds to their speed and, inversely, their space. Different memory speeds are often explained as being functions of "locality" as the speed of different memory tiers relates to its presence on or off the chip and potentially whether bussing is needed, but there's a second sense in which locality is used with regards to the data itself rather than a physical characteristic of the medium the data is in.

This second sense derives from the fact that the same or adjacent memory addresses tend to be needed at the same time or place. When the same data needs to be used consecutively across different instructions it has temporal locality, and when the memory needed is located contiguously at adjacent memory addresses it has spatial locality. Memory hierarchies take advantage of temporal locality by holding onto more recently accessed data items and retiring older data to more distant memory in the hierarchy, and leverage spatial locality by moving blocks of contiguous chunks of memory from a less efficient memory tier to a faster tier any time a necessary address isn't discovered at the current tier.

In high performance computing the memory category also dictates the kind of memory, either volatile or non-volatile, however this is not strictly true for all computing fields e.g. embedded systems. Volatile

memory requires constant power to prevent data loss but is often used for memory with faster needs and more frequent exchanges, whereas non-volatile memory retains data regardless of power and is utilized for long term storage.

A digression into the tiers of hierarchical memory will be valuable to the reader and the pursuing sections will examine this closely.

2.3.2.1 Storage

A hard disk drive (HDD) is still standard for modern storage because it utilizes non-volatile memory, as the reader might guess given that the programs and files on (most) computers aren't erased when the machine is power cycled. The data is stored on an aluminum or glass-ceramic annulus called a platter magnetically by a thin coating on both sides of the platter of a ferromagnetic material. The data is recorded by alternating the magnetic direction of the ferromagnetic grains to encode bits, and the data is non-volatile (and semi-permanent) because the magnetization is independent of any power source.

Storage is slow relative to on-chip and primary memory in part because, as discussed in §2.3.3, data to or from the HDD must be encoded and transmitted via a SATA connection and decoded at the destination. However the principal bottleneck is that reading the data requires mechanical operations of the components - the alternating directions in the magnetic field are examined by magnetic heads which can only read the data while the platter is spinning (at rates anywhere from 5000 to 15000 rpm), the spinning necessary to induce current in the head and impart the magnetically encoded data. While the examination of the data once the head is in place is fast, given the reliance on physically actuated components, typically on the scale on 5-20 milliseconds, it is still many orders of magnitude slower than even just the bussing system from the drive buffer to memory via the SATA connection, which is in turn considerably slower than the signal transmission in primary and chip-bound memory, which occurs across double digit or even fractional nanoseconds . Even modern storage formats which eschew mechanical/optical components for non-volatile RAM (NVRAM) such as solid state drives (SSD), while considerably faster than HDDs (Figure 14), are still significantly slowed by encoding and transmission across the SATA connection.

2.3.2.1.1 Virtualization and Paging

The operating system manages memory at a scale far beyond any one program, sharing memory between dozens of simultaneously running programs and allowing for user and system driven interruption events. As such, the computer would run out of memory extremely quickly if not for virtual memory. Virtual memory, or virtualization, abstracts the storage resources that are actually available to create a simulacrum of a much larger main memory.

A combination of hardware via the MMU and operating system software maps memory addresses used by a program (virtual addresses) into physical addresses in computer memory. Main storage, as seen by a process or task, appears as a contiguous address space or collection of contiguous segments, even though the memory is not located contiguously. The operating system manages virtual address spaces and the assignment of real memory to virtual memory. Address translation hardware in the CPU, the MMU, automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities to provide a virtual address space that can then exceed the capacity of real memory and thus reference more memory than is physically present in the computer.

2.3.2.2 Primary Memory

Whereas the hard drive records data magnetically, primary or main memory records data with gated circuits of charged or uncharged cells. As such, it is considerably faster as there is no mechanical actuation needed to read data, only circuitry gating (and virtually negligible loss for electron drift).

Random Access Memory (RAM), so called because the memory can be accessed "at random" rather than sequentially, is used for primary or main memory, sometimes such that RAM and main memory are used synonymously. While many more types of memory currently exist or are being researched [24] such as the NVRAM mentioned in §2.3.2.1, there are two principal formats for RAM: DRAM (dynamic RAM) and SRAM (static RAM), both of which illustrate the gap between memory and caching. In both SRAM and DRAM, bits are recorded by transistor/capacitor circuits called memory cells which store a binary value, charged (1) or uncharged (0). The distinction between S and D RAM (which leads to many other ancillary distinctions) is that DRAM has to be *dynamically* refreshed - DRAM's memory cell has a single transistor gating the capacitor, which results in a tendency for the capacitor to "leak" (slowly discharge), such that DRAM needs to be refreshed semi-consistently (approximately every 10 ms). Moreover, every time RAM is read, it necessarily discharges and needs to be re-written. This results in both a large power draw as the memory refreshes to renew slowly dissipating capacitors, but it is also slower, as in order to refresh, the current contents of the cell need to be read, then rewritten, so a great deal of work is done in just maintaining the current state.

In contrast, SRAM has no need for refreshment because its memory cells, while still using a single capacitor, utilizes (typically) six transistors to construct a flip-flop circuit with binary stability rather than just a single capacitor being charged or not charged behind a transistor gate (although the number of transistors varies from 4 to 8 for specialized hardware). Specifically in the type of SRAM most commonly encountered in HPC and personal computing today, 6T, there are only four transistors in the flip-flop circuit used to form two cross-coupled inverters, and the other two transistors control read-write operations. This both makes the data in SRAM more stable, as it doesn't need refreshment, just constant power, but is also why

SRAM is faster, as clock cycles aren't wasted continuously reading/writing to the same memory cell just to retain the data by refreshment as in DRAM, anywhere from 10-20 times faster than DRAM to be specific.

The cost of SRAM's speed and stability is that it requires many times the transistors and even more space on the chip (approximately 20 times more), as well as more complicated and expensive manufacturing. As such, DRAM is considerably cheaper and its relative size means many more cells can be compacted within the same physical space. As such a hierarchy has emerged where SRAM dominates the hierarchical niches which require speed and stability, i.e. the register and cache memory, whereas DRAM dominates in main memory.

2.3.2.3 Caching

While main memory is several orders of magnitude faster than fetching from storage (Figure 14), it's still considerably slower than the rate at which the processor can operate, and so it still forms a bottleneck in processing. Data caching reduces this latency. When new data and instructions are called by the controller they are first searched for in the various caches' SRAM and if not found, fetched from the main memory and stored in the cache. In this way, the cache is actually an entirely redundant system to the main memory as the majority of its contents are duplicated, although, as the cache is often updated from the processor, it also needs to be synchronized to make sure its contents match main memory to maintain cache coherency.

There are a variety of schemas for promoting and demoting data on the cache which determines if and at what level the data will be found in the cache hierarchy at a future point. The simplest is the least recently used (LRU).

As mentioned above, the cache itself in most modern CPUs is also a hierarchy, referred to as L1, L2, L3, etc. The reason the cache is further subdivided is because, in the same sense that the primary memory is a subset of all the data in storage but much faster, and the cache is a subset of the memory, and in the same manner that memory access becomes much more expensive when it fails to retrieve the needed data from memory and needs to fetch it from storage (a "miss"), the cache can also miss. The time spent searching a cache which results in a miss is in some sense time wasted and there is setback from having examined the cache rather just going to memory; however, having additional tiers of larger and larger caches can still provide significant speedup over direct access from memory, even with numerous misses across the cache because SRAM is so many orders of magnitude faster (Figure 14). The L1 cache is usually reserved for high frequency data, typically recently used memory. L2 is a larger bank of memory that's been demoted from L1. The L3 cache is typically shared across the cores of the CPU (and is often OS related).

2.3.2.4 Registers

In the memory hierarchy, registers are the alpha and omega in both cost and performance, and so most modern CPUs have a few dozen registers on a single chip. As an integrated circuit they are SDRAM, typically configured in 8T circuits. The registers are typically the data source and destination operands for the ALU, the intermediary between data memory put into the ALU and work done there to examine or transform data, although sometimes outputs are written directly to the cache or main memory. The set of registers at a given point in the execution of a program is called the execution context and is effectively the operands, instruction pointer and program pointer – switching contexts can be as small a task as writing one execution context out and loading another in – this is, mechanically, effectively what constitutes a thread.

Registers are generally subdivided in terms of their assigned role within the CPU. Instruction registers (IRs) hold the current instruction while it is decoded and executed. As each instruction to be executed is fetched from memory, it is placed there for the fastest possible reference. The stack pointer, sometimes called the program counter, holds the main memory address of the current position in the program stack to enable proper progression through the program, allowing frames to pop on and off the stack in function calls or to migrate due to conditionals. General purpose registers are subdivided either as address registers, holding the locations of main memory locations with data to draw or be written to, data registers containing actual ints, floats, characters, etc., or are status or control registers indicating where branching or context switching must take place (although some registers are truly general purpose and do not exclusively fill only one niche).

Registers are not addressed directly however - in the same manner that the cache is managed through the virtual lookaside buffer, or the main memory and storage are managed via paging, there is a virtualization of the processor registers referred to as register renaming, which acquires and retires registers through a register file rather than direct invocation, and engages registers as “logical” registers rather than directly as physically coupled registers. Register renaming is managed by the reservation unit. Allowing fluidity of which registers do what eliminates extraneous data dependencies and thereby improves ILP via superscalar and OOO execution.

2.3.3 Busses

In the same manner that distributed computing systems must use message passing systems to share local memory resources from the CPU to other devices (the CPU is generally referred to as “the host” in GPU/CUDA literature), memory allocated and accessible to the program running on the CPU is not directly available to the GPU (GPUs are referred to as “the device” in CUDA literature). Such data must be explicitly communicated between the host and device in the same manner as e.g. the master and slaves in MPI. This occurs locally on the machine via a bus system on the motherboard. Ordinarily GPUs bus using one

of the members of the peripheral component interconnect (PCI) connection family, principally PCI-e in the last two decades although specialized busses do exist (such as NVLink on V100 model hardware, such as the Tesla described in §2.0).

The question of how data is transmitted and how quickly across the GPU is instructive, because one of the most expensive tasks involved in GPU parallelization is data transmission between the host and the device, which must all necessarily travel across the bus. When the GPU is latency hiding (cf. §2.3.4.2.1) the determining factor for the duration of latency is the combination of the bussing speed and size of the read/write actions in the device memory.

First introduced in 1992, just 7 years prior to the first GPU, there are myriad variations in design and capability within the family of PCI slots (PCI, PCI-X, PCI-mini, PCI-e) and the PCI family is fairly standard for various computer peripherals such as network cards, sound cards, extra ports for e.g. USB, etc. Today, most GPUs, AMD or Nvidia, operate via PCI-e, of which there are now 5 generations, each offering different transfer speeds and methods of packetizing the data, each successive generation effectively doubling the speed of transmission over the previous generation (2.5 Gb/s of possible bandwidth in Gen 1, 5 Gb/s in Gen 2, 8 in gen 3, 16 in 4, and finally nearly 32 Gb/s in gen 5, released in May of 2018). This paper will not explore the ramifications of the different voltages, dimensions, etc. of PCI busses on GPU performance as those fall well beyond the scope of this thesis.

PCI-e busses communicate data via a link, each link minimally composed of 1 but generally 16 to 32 lanes in GPUs, although configurations of four and eight also exist, generally for other peripherals such as sound cards with lower data intensity. Scaling for bandwidth is linear with lanes, so e.g. a slot with eight lanes is twice as fast as a slot with two. Each lane itself is composed of four wires, two for transfer to a device and two for transfer from a device, allowing simultaneous transmission along the same lane to and from a device.

The need for two wires is due to the physical scales of the wire diameters, their proximity, and the currents involved: electromagnetic interference from neighboring lanes is measurable and can introduce noise in the signals transmitted. Two wires allows differential signaling, which is noise resistant because it measures the difference between signals rather than a raw signal; the signals sent along both wires are close enough that both will be modified by virtually the same amount, meaning their difference won't be altered and any noise is effectively filtered, e.g. if the intended signal were "2", a "4" and "2" could be sent, and if there were interference, both signals would be equally altered, so the signal difference would remain 2.

Data is transmitted across the PCI-e as a data packet called a transaction layer packet (TLP) through three layers: the transaction layer, the data link layer, and the physical layer, each layer adding a limit or burden

to the rate of data transmission. At the base level is the physical layer which describes the actual circuitry and logic gates limiting transfer rate. Data is packetized in the data link layer, where different encoding strategies impose different overheads in the size of the data that must be transmitted to reduce (20% of all transmitted data in generations 1 and 2 for 8b/10b encoding, 2% for 128b/130b)

Large data transmissions across PCI-e such as in GPU exchanges that must be transmitted on multiple lane links are interleaved, meaning each successive byte uses an adjacent successive lane. The PCI-e specification calls this interleaving "data striping". While this adds significant complexity to synchronize or deskew striped data, striping significantly reduces the latency, which in e.g. real time rendering is extremely valuable. While not strongly synchronized, the extent to which skew is limited in the various PCI-e generations so that the hardware buffers can re-align the striped data. Striping does not necessarily reduce the latency of small packets because of padding requirements, but rarely are small packets sent when working with GPUs.

As with other high data rate serial transmission protocols, a cyclic redundancy clock (CRC) is used to verify the accuracy of transmitted packets. PCI Express 1.0 and 2.0 implement symbol encoding utilizing the 8b/10b encoding scheme [25], the same encoding often used in SATA connections to the disk drive, to ensure that consecutive identical binary digits are limited in length (no more than 5 identical bits consecutively), specifically to prevent the receiver from losing track of where the bit edges are due to clock issues. 8b/10b encoding transmits every eight uncoded bits of data as 10 encoded bits of transmit data, causing a 20% overhead (limiting repeated data patterns reduces noise from electromagnetic interference). To improve the available bandwidth (and meet the goal of doubling bandwidth rates with each new PCI-e generation), version 3.0 uses 128b/130b encoding with XOR scrambling. S128b/130b encoding relies on the scrambling to limit the run length of identical-digit strings in data streams and ensure the receiver stays synchronized to the transmitter. Most modern GPUs use a x16 PCI-e 2.0 or 3.0 interface.

What this hopefully conveys is a sense of how data transmission from the host to the device can quickly become expensive and minimizing exchanges between the two is important in optimizing performance. While this barrier was tolerable in the original problem space GPUs occupied in graphics processing, where, once an image had been processed on the GPU, it could be bussed directly to the display monitor without requiring further processing, scientific computation often needs the data transmitted back to the CPU where it is either examined, utilized in another CPU driven calculation, or re-transmitted to the GPU, or any combination of those, which can slow computation considerably. In this regard it is best to keep data locked to the GPU for as long as possible in contiguous spans of processing time and design kernels/GPU implementations to perform as many related computations contiguously in time as possible.

2.3.4 GPUs and CUDA

A GPU is a massively parallel device built around a Single Instruction Multiple Thread (SIMT) model for parallel processing. This is derived from the Single Instruction Multiple Data (SIMD) paradigm in Flynn's taxonomy of parallel processing. SIMT divides computation not across compute nodes but across threads, where threads move in and out of context to load, process, and unload data efficiently. Generally speaking, this thread oriented mode of data processing means that GPUs scale considerably more with large data sets and conversely, can struggle to find computational parity with small data sets (if at all) [4].

CUDA (Compute Unified Device Architecture) is nVidia's proprietary wrapper for the C/C++ languages that allows code to run on Nvidia's GPU hardware. Programmers define CUDA functions, called kernels, which are then dispatched to the GPU across a bus to perform whatever computation the kernel entails. CUDA is compiled through nVidia's proprietary NVCC (nVidia CUDA Compiler). Kernels are expressed identically to ordinary functions except some additional markup is required to indicate it is a kernel to be run on the device. Invoking a kernel also requires specifying the number of CUDA threads that will execute that kernel and the number of grids that will own those threads. A variety of intrinsic variables exist that are accessible within each kernel such as the threadId variables (for each of the Cartesian coordinates) to identify threads, blocks, etc.

Figure 15. Kepler Architecture. Image Source: nVidia Corp.

It is not strictly necessary that programming for nVidia GPUs be done via CUDA – there are open source programming solutions such as OpenACC (OpenACCelerators) and OpenCL (Open Computing Language), however, there are performance disparities between the two related to how OpenACC translates GPU kernels to object code and volatility in its heuristics for optimization which the NVCC does not have, at least in comparison. In that regard the maximal opportunity for performance gains are possible via CUDA, however, the learning curve for OpenACC is not as steep, and the development cycle to performance improvement tradeoff may make OpenACC development more cost-effective. [26]

Figure 16. Kepler SM. Image Source: nVidia Corp.

By "wrapper" nVidia means the NVCC is not a complete compiler and it requires a separate functional host side compiler (GCC/G++ for Linux environments, CLANG for MacOSX, cl.exe for Windows) to generate the host binaries, itself only producing the cubins (CUDA binaries). These cubins are typically embedded directly within the host side binary, however, it is possible to leave them unlinked at compile time, potentially to provide interoperability for different languages in building a library, and many other languages provide their own CUDA libraries to operate on nVidia GPUs (e.g. NUMBA and PyCUDA for developers of a Pythonic twist).

Despite the different performance and priorities when programming for one environment or the other, it is important to remember that the GPU is a highly specialized CPU and one of the principal changes derives from removing the speculation machinery that make optimization in host-oriented development more streamlined – the integrated circuitry driving it are the same as those on the CPU, the structure of caching and hierarchical memory are similar, generally the largest difference is the proportions and control schema such that in a GPU, the emphasis is on data throughput, so instructions are streamed to many ALUs simultaneously at the expense of branch prediction, out-of-order execution, and memory pre-fetch, whereas the CPU optimizes control and prediction speculation, but the actual processing capacity via ALUs is, comparatively, limited.

Figure 17. Graphic depiction of proportion of CPU and GPU sub-components. Image Source: nVidia Corporation

2.3.4.1 CUDA Cores and Streaming Multiprocessors

As the core is the smallest unit of a gestalt CPU, so too do GPUs have distinct cores (or in the case of AMD GPUs, conflictingly referred to as streaming multiprocessors). However, the deconstruction of the CPU core's transistor resources, or rather, redistribution, does not make the analogy 1:1. While the exact number of cores is strongly hardware dependent for any particular GPU, contemporary units often have thousands of cores, and every GPU is composed of anywhere from tens to hundreds of aggregate core units called Streaming Multiprocessors (SM) (Figure 16). A slight digression into the ambiguity of this nomenclature is worthwhile: historically, nVidia referred to their processing units as CUDA Cores because they implemented a SIMT strategy, whereas AMD's ATI derived GPUs were true streaming multiprocessors relying on vector processors [27], however, the distinction has largely become muddied, as nVidia now refers to the groupings of CUDA Cores and control logic for the core that are used as gestalt computation nodes in warps as streaming multiprocessors in some of their literature [28].

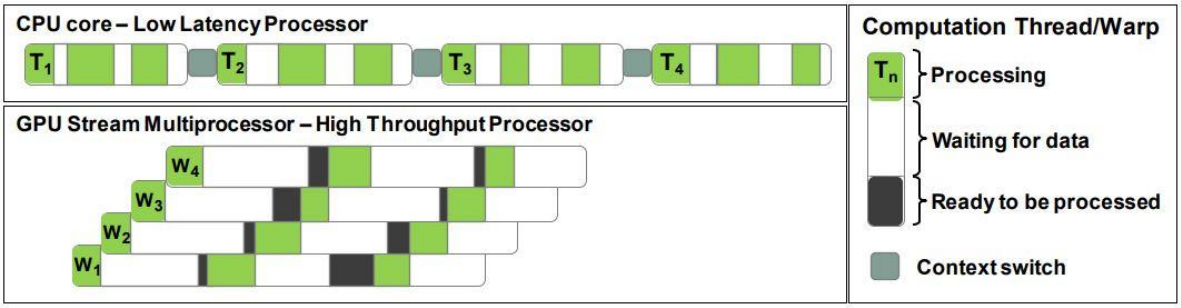


Figure 18. Warp latency hiding. Image source: nVidia Corporation.

Each CUDA core is analogous to an individual core in the host in terms of being a lane to process data on, however this isn't a perfect analogy, as a GPU SM encompasses many cores, and the SM holds not only the control units, registers, and pipeline associated with a single core in a CPU, it also encompasses limited caching capabilities, and multiple SMs share a SM scheduler to coordinate the warps [29] [30] [31]

2.3.4.2 Control Units, Flow, Pipelines, and Branching

The control units in GPUs are significantly atrophied in comparison to CPUs (Figure 17), but ideally this does not matter as explained below. However, this is one of the strongest points of divergence in the programming paradigm of GPUs due to the hardware, as many of the techniques discussed in the host are subtly changed or simply don't exist in the GPU.

When a kernel is launched, the programmer explicitly dictates how many threads are launched,

subdivided into blocks and grids. The number of blocks must be explicitly dictated by the developer with the kernel launch, however the GPU will manage the aggregation of blocks into grids. Blocks are effectively the aggregation of threads as they will be encountered by the SM, and as such are limited to 1024 threads (or 512 for pre-CUDA 10.0 versions). A single SM may manage several blocks but only ever one grid.

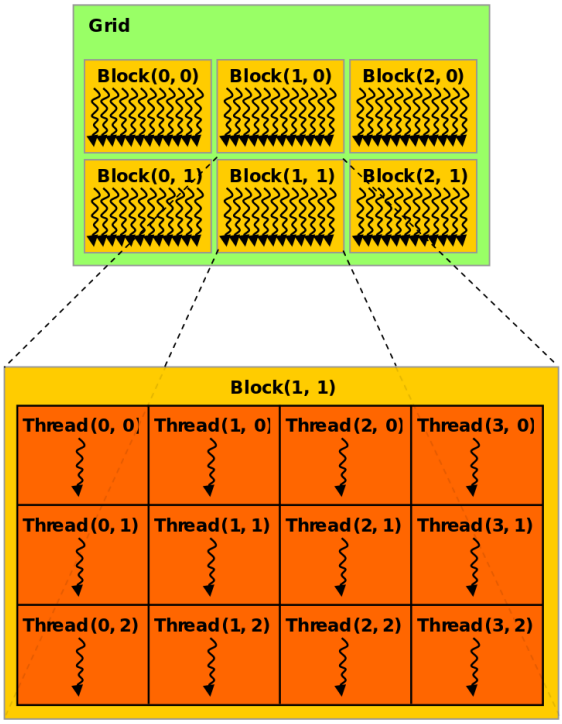


Figure 19. CUDA Blocks and Grids

2.3.4.2.1 Warps, Warp Scheduling, and Latency Hiding

Warps are the gestalt unit of CUDA instruction parallelism, below which nothing smaller is possible from the perspective of SIMT. A warp consists of 32 independent threads which share a scheduler and instruction unit. All threads in the warp share the same instruction and perform the same operations, they are simply directed at different data (SIMT). This means that when jobs are launched on the GPU, they ideally use some multiple of 32 threads so that each warp is completely filled. The extent to which the warp is filled is referred to ask occupancy, and poor occupancy effectively means that resources will be used but make no progress – regardless of any programmer input, a warp will always instantiate with 32 threads, whether or not each thread has useful work to do.

Understanding latency hiding is critical to understanding the problem spaces where GPUs out-perform CPUs and it derives from the scheduling of warps. As detailed throughout §2.3.2, in a CPU, time spent transferring memory is time functionally inert, as data must be located elsewhere, then sent to the right location, during which time no modification is done, and so a great deal of work is done by the control unit in CPUs to minimize time spent fetching and transferring data and making sure the data that is fetched is loaded fast caches. But the cost of retrieving data cannot be totally eliminated, it can only ever be diminished. Yet in the case of GPUs, if the algorithm is well suited and the kernels well implemented, workloads can be shuffled around via instruction scheduling and context switching to do work continuously so that there is almost no time when the GPU is idle due to loading memory. This is what is meant by latency hiding, depicted in Figure 18.

Scheduling on the warp follows somewhat similar schema to the retirement schemas of cache memories. Some architectures schedule warp instructions in a round robin manner, where it is agnostic to the events and scheduling of other warps (beyond avoiding structural hazards). There's also a least recently fetched

schema, where the warp that least recently held a compute context is reloaded, with the expectation that this balances the load in a FIFO-analogous manner.

The reader should note that a GPU is generally much slower and that far more time is spent in memory transfer and loading, however, because of context switching within streaming multiprocessors, if memory is properly managed the GPU will functionally be continuously churning and modifying data, whereas the CPU will virtually always at some point be forced to sit idly while memory is transferred. If memory on the GPU is poorly managed, it can be considerably worse.

2.3.4.2.2 Instruction Pipeline

GPUs do implement ILP through instruction pipelines [32], however, whereas CPUs implement instruction decoding for each ALU for a given instruction, GPUs implement a single decoder for a large grouping of ALUs (Figure 16). This tethering of more than two dozen execution contexts and ALUs to a single instruction decoder considerably speeds up processing time and is fundamental to SIMT, but it is also why the 32 threads of a warp are tied together in lock-step except through predication.

As far as this author was able to determine, nVidia does not have explicit external documentation of the mechanics of the hardware with regards to reservation stations, re-order buffers, etc. within their documentation (assumedly because such knowledge isn't generally exposed to ordinary developers for host programming), although users can surmise that the performance gains from such techniques would have some manner of implementation within the GPU. There is the notable exception that nVidia held a patent from 2003 to 2010 for "Across-thread out-of-order instruction dispatch in a multithreaded microprocessor" [33], and that presumably they hold another patent to perform a similar task in contemporary GPUs since that one's expiry. In fact, some of the best resources on GPU memory hierarchy come from microbenchmarks reverse engineering the structure [34] [23].

2.3.4.2.3 Branch Prediction

Within branch prediction's importance in optimization generally, it is particularly so in GPU optimization, in that it should generally be avoided. nVidia provides no end of documentation detailing how to prioritize and minimize branching varying from moving conditional examinations further up the pipeline so as to resolve them sooner to Z-culling, a powerful tool in graphical problem spaces to exempt calculation for visual spaces that are occluded, but less useful in generalized non-spatial computation (although in spatial-oriented tasks such as imaging there may still be utility).

Branching is considerably different within the GPU. GPUs are capable of true branching, however, because there is one CU for each warp, the branching will occur for all threads within the warp, which, for many

GPGPU purposes, is not ideal, as in e.g. iterative ODE solvers that need to determine when to stop iterating via the residual, which relies on branch comparisons.

Pascal and earlier nVidia GPUs execute warps sharing a single program counter across all threads in the warp, combined with bit masking. Bit masking uses a single bit in the instruction to declare certain threads active or inactive at any given time. This means that divergent execution paths leave some threads inactive, serializing execution for different portions of the warp. The original mask is stored until the warp reconverges at the end of the divergent section, at which point the mask is restored and the threads run together once again.

2.3.4.2.4 Predication

Predication is generally how GPU's control flow in branching code at least historically [35], but it is effectively static branch prediction where, instead of selecting one or the other branch to take and then sorting out which is correct, predication allocates resources for both branches, runs both possibilities through the pipeline, then discards the results of the incorrect one. Predication assigns both the true and not-true value taken when instructions are fed into the pipeline, and the instructions in each half of the branch must check the condition code before writing to registers. This means that instructions in the correct branches are the only ones to write their output, but all branch points cost as much as both parts of the branch and branching should be used sparingly on such architectures.

2.3.4.3 GPU Memory

Memory management on CUDA devices is a key aspect of writing GPU parallelized code – management across the complete array of available memory is consistently indicated as critical to maximizing hardware performance across the literature [4]. As with host memory, there is a hierarchical memory management system within CUDA, it employs paging, virtual addressing, and TLBs [34]. There are several classifications of GPU memory: local, global, register, shared, and the cache.

2.3.4.3.1 Local Memory

Local memory resides in device memory, i.e. on the GPU card, but outside the streaming multiprocessors. Physically it exists as DRAM (cf. §2.3.2.2) and therefore has similar performance and problems with the memory wall, making data accesses from local memory expensive – in fact, it requires approximately 45 times to retrieve from the GPU cache as it does the registers [4]. As the name would suggest, it extends local scope, meaning only the thread it is allocated to can access its contents, not even identical threads within the same warp cannot access it. It is analogous to main memory in a host environment.

Local memory is ideal when the data is larger than can fit in registers, the cache, or shared memory, but the programmer wishes to avoid the possibility of accessing the data via global scope from other threads, as that could lead to race conditions.

The compiler automatically assigns to local memory when it determines there is not enough register space to hold data. This is referred to as "register spillage". The amount of occupancy in the registers is referred to as register pressure, because at excess "pressure" the storage vessel "gives" and data "overflows", although unlike how the name would lead one to believe, it is assigned to local memory in an orderly(ish) fashion. Register spillage is a common cause of poor performance.

2.3.4.3.2 Global Memory

Global memory is not a separate instance of memory such as the registers or the cache but an abstraction within the physical local memory in the same manner that the stack and the heap are abstractions within the main memory, making it also as expensive to access as global memory. Its scope is however global, meaning any streaming multiprocessor may access the data and so global memory is both a key resource for moving data between warps, streaming multiprocessors, and therefore also a potential site for race conditions. Programmatically, the only distinction between local and global memory is the scope it holds for which threads can (in the case of global memory) or cannot (in the case of local memory) be accessed.

2.3.4.3.3 The Cache

While this was not always the case, modern GPUs, particularly GPGPUs, support cache memory. This significantly improves performance for all the reasons laid out in §2.3.2.3, however, it introduces those same problems, and cache missing is particularly problematic for GPUs, as, if even one thread should miss, the entire warp, conceivably 31 other threads, can stall as the data is fetched.

Every SM holds its own L1 cache while the L2 cache exists outside the SM and is shared by all of them. It is unified for instruction, data and page table access. Furthermore, nVidia GPUs use a page table to map virtual addresses to physical addresses, and the table is typically stored in the global memory. The TLB is the cache of the page table. Once a thread cannot find the page entry in the TLB, it would access the global memory to search the page table, which causes significant access latency

2.3.4.3.4 Register

Registers in GPUs function in the same manner as described in §2.3.2.4 for the CPU, but whereas in the CPU they are the rare and prized resource at the narrow apex of a pyramid built on main memory as a foundation and then caching, much of the cache from the CPU has been redistributed in the GPU as registers. However, that does not mean they are plentiful - their scope is limited to the GPU core/ALU they are

attached to, and because there are so many magnitudes more cores/ALUs in a GPU, there isn't a wider register file to work with per se within each core.

As in the CPU, they have the greatest locality of any memory, meaning that any data which can be assigned to the register is handled fastest out of any memory bank, often requiring no clock cycles per instruction in most instances. However, delays can occur due to cache misses, bank conflicts, or particular dependencies such as RAW hazards, which, in the latter's case, can consume a couple dozen clock cycles.

In addition to the read after write latency, register pressure can severely detract from the performance of the application. Register pressure occurs when there are not enough registers available for a given task. When this occurs, the data is "spilled over," consuming local memory and greatly reducing performance. For this reason, CUDA allows a more direct allocation of register resources on the device than host environments allow, permitting users to restrict or loosen restrains on how registers are allocated to each core such that occupancy via thread count can be improved, or reduced to avoid register pressure.

2.3.4.3.5 Shared memory

Shared memory on the GPU allows communication within the same thread block within a streaming multiprocessor without going to the local or global memory, which, according to nVidia, is up to 100x faster than a global or local memory fetch [36]. It is the only GPU memory that can be declared as an unsized extern, although that must be declared with the kernel launch.

Importantly, as the Global Memory is to the Local Memory in needing to be apportioned by the developer, so too does the Shared Memory and the cache split the same memory resources. Historically the developer needed to explicitly declare which of the two would be the dominant memory resource, either 48KB of shared memory and 16KB of cache, or vice versa, with the device defaulting to more shared memory, although later device architectures do no require this. [37]

2.3.4.3.6 Unified Memory

Whereas in host/CPU environments, memory has grown so cheap that the limiting factor in performance is typically accessing speed rather than capacity, the limiting factor in GPU architectures is capacity. As with CPU optimization, GPU optimization requires working as locally on the GPU as possible, particularly for iterative applications using the same data multiple times or with a high flops/byte ratio. The limited memory capacity of GPUs means real-world codes must selectively use data on the GPU, allocating only necessary parts of the working set to GPU memory.

Traditionally, GPU developers have only been able to explicitly copy memory to transfer data from the host to the device. Unified Memory shares data across the host and device, combining the advantages of explicit

copies and zero-copy access. Unified memory allows the GPU to access any page of the entire system memory and simultaneously migrate it to its own memory [36]. While explicit management still gives the best performance [37], it requires attentive management of GPU resources and also predictable access patterns of the data. Zero-copy access provides direct access to the system memory, significantly improving legibility and maintenance, however, then the bridge (PCI-e or NVLink) becomes the limiting factor, and locality still can’t be leveraged.

3 The Tensor Template Library

3.1 Templates

The TTL provides two user-defined templates, Index templates and Tensor templates. An Index is simply a template that converts 8-bit integers into unique types from some regular expression. Indices appear in Tensor binding operations and implement symbolic matching between tensor dimensions within expressions. Tensors manipulate and shape data storage.

3.2 Indices

The Index class template creates unique types for each character regular expression that it is parameterized with. Index values that have the same class are treated as the same index i.e. there is no collision. The resulting type can be manipulated by the TTL internal infrastructure at compile time in forming tensor expressions. This allows TTL to perform various operations based on the indices bound to the tensors in order to perform index matching and code generation for expressions.

```
static constexpr ttl::Index<'i'> i;
static constexpr ttl::Index<'j'> j;
static constexpr ttl::Index<'k'> k;
static constexpr ttl::Index<'l'> l;
```

Figure 20. Index template instantiation

Source-level indices only occur in constant, compile-time contexts and thus it is common to see them declared as constexpr, const, or both. An example instantiation of index templates is shown in Figure 20. Index template instantiation

3.3 Tensors

The Tensor template defines a square, multidimensional, row-major array structure of statically known array size. In alignment with the definition this author provided in §2.2.1, from a data storage perspective, a Tensor and a statically-sized multidimensional array are the same thing, as shown in Figure 21.

```
ttl::Tensor<4,3,double> A;
A[1][2][0][1] = 1;
double a[3][3][3][3];
a[1][2][0][1] = 1;
```

Figure 21. Data storage equivalence of Tensors and multidimensional arrays

TTL Tensors follow the form <Rank, Dimension, Datatype> to declare the tensor rank, dimensionality of each rank, and the type of data the contents are. As any other template this must be declared at compile time due to the need for template deduction. TTL Tensors can be initialized elementwise, using initializer lists, or expected copy/move operators. If the data type is a pointer such as double*, the Tensor needs to initialized with a reference to an external buffer of the underlying type, and this will not be a deep copy, operations will effect that external storage. Externally allocated Tensor storage can be useful with legacy code but it is primarily intended

```
double external[3][3][3][3];
Tensor<4,3,double*> A{external};
A(1,2,0,1) = 1.0;
```

Figure 22. External data allocation

and useful in the context of CUDA driven GPU code where explicit memory management via mallocs is required. Data that is pointed to on the host can be migrated and pointed to on a kernel side Tensor.

The principal difference as an API between a TTL Tensor of rank N and a statically sized square array of N multidimensionality is that a Tensor can be "bound" using indices to implement its family of operator()(...) overloads that define the TTL expressions. A Tensor can be bound either using integers in the range [0,Dimension) or Index<> types (cf. §3.2), or both. Unlike how most objects are viewed in object oriented programming (OOP), raw TTL Tensors are rarely useful; while one can make raw TTL Tensors, in much the same sense that tensors define operations between tensor objects, the utility of the TTL comes through binding to access the expression library. There are a handful of constant expressions that do not allocate space but allocate pre-determined values around 1 and 0 (operations using the zero tensor, levi-cevita, the identity tensor, etc.). Supported invocations of the different Tensor expressions are shown in

```
Scalar Operations: B(i,j,k) = s * A(i,j,k);
Tensor Products: C(i,j,k,l) = A(i,j) * B(k,l);
Tensor Inner Products: C(j,l) = A(i,j,k,l) * B(i,k);
Tensor Contraction: s = A(i,i);
Arbitary Index rearrangement: A(i,j,k).to(k,j,i);
Zero Tensor: A(i,j) = zero(i,j);
Identity Tensor: A(i,j,k) = identity(i,j,k);
Levi-Civita Tensor: A(i,j,k) = epsilon<3>(i,j,k);
Delta Tensor: A(i,j) = delta(i,j);
```

Figure 23. Tensor Expressions

TTL supports not only tensor expressions but a small number of library functions that can be applied to square rank 2 Tensors or which can be reinterpreted as a square matrix. These include transposing the contents, solving a determinant (ill-advised for large matrices), generating the inverse, and a choice of several linear solvers.

4. Methodology

4.1 Hardware

All unit test work and microbenchmarking that was host driven was done using the resources of Notre Dame's C-SWARM compute cluster. Host side testing was run on Intel Xeon E5-2620 v2s. These offer 12 cores per chip, with up to 2.6 GHz clock speed, 64KB of L1 cache, 256 KB of L2 cache, and 15360 KB of L3 [38]. Characterization work with loads more resembling deployment were done on LLNL's (Lawrence Livermore National Laboratory) Pascal cluster, which implements nVidida P100s for GPGPUs and Xeon E5-2695 v4s for host processing [39]. P100s utilize the Pascal architecture and offer 4700 GFLOPs of double precision and 9300 GFLOPs single precision compute power, spread across 3584 cores [40], while the Xeon E5-2695 v4s offer up to 3.3Ghz processing speed, a 45 MB cache, and 36 cores across 2 sockets per node [41].

TTL unit testing and microbenchmarking for the GPU were run on a GeForce GTX Titan Black, which implements the Kepler architecture (Figure 15). The unit offers 5120.6 GFLOPs of single precision and 1706.9 GFLOPs of double precision compute power across 2688 cores, those cores distributed among 14 streaming multiprocessors. Each SM has an average clock speed of 900 Mhz but is capable of 1Ghz speed if the specialized application specific integrated circuit (ASIC) vector processor units can be utilized. Memory wise it holds over 6 MB of GDDR5 memory with 64 KB of cache [42].

Additional testing was done with the GCM on a Tesla V100 GPU Accelerator using Indiana University's Juliet cluster. This was done primarily to utilize the Volta architecture, which offers warp thread voting to reschedule low-occupancy warps and improve poor performance due excessive context holding in non-uniform thread resolution. This was done to examine the importance of thread divergence in the GCM due to its iterative Newton-Raphson methods. The V100 has 5,120 cores, 640 tensor cores (vector processors), 7.8 TeraFLOPs of double precision and 15.7 TeraFLOPs of single precision performance, and 32GB of memory and 900 GB/sec of bandwidth. [43] [44]

4.2 Compilers

The Gnu Compiler Collection (GCC) 7.1.0 was used in generating almost all host-exclusive code. MPICH-gcc-7.1.0 was used in generating MPI code to examine distributed-memory parallel processing (DMPP) for the GCM (this is a common MPI wrapper compiler for a host compiler, in this case GCC 7.1.0). In microbenchmarking, GCC 7.1.0 was utilized to generate the host side code, and the NVCC 10.0 was utilized to develop both host side and device side assembly for comparison. With regards to the GPU, very early work in building the test suite to verify functionality was done via NVCC 9.2, but final verification and later work was done via the NVCC 10.0.

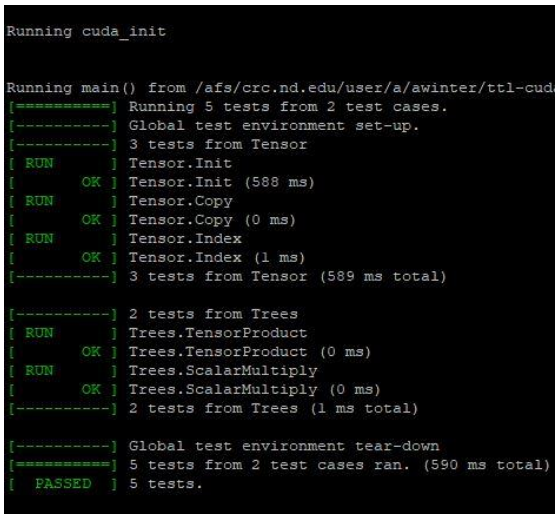


Figure 24. Successful GTest output for test submodule, cuda_init, examining object constructors

4.3 Tools

The GoogleTest package from Github [45] was used for building the unit tests that were needed to examine the functionality of the TTL in a GPU environment. It provides an easy interface to build independent tests that compile together but can be tested and debugged individually when needed. Googletest examines code via macros resembling function calls in a similar manner as static assertions in the C++ library. Developers test their code by "asserting" about its behavior, and if an assertion fails, googletest prints the source file and line number location causing the failure (Figure 24). It also has an easy to work with framework in CMake.

Verification of GPU functionality of the TTL was performed using CUDA unified memory (cf. §1.4.2.6) as performance was irrelevant for verification purposes. The earliest implementation of the GCM which deployed the TTL also utilized unified memory – this was originally predicated on nVidia's guarantees of unified memory offering "the performance of local data on the GPU, while providing the ease of use of globally shared data". [46] [36] We found that to be half true, but the untrue half proved pertinent for performance profiling, and other investigators have had similar difficulties [37], so we eventually reverted to explicit memory management for both the GCM and the microbenchmarking.

All compilation was managed through CMake 3.13, which generates all the necessary files for Make to process the source code to binary files. NVProf was used in early work to examine performance of the GCM and search for areas to optimize performance.

The microbenchmarking test suite and assembly generating tests to examine particular TTL operations collected data using the standard library high resolution clock as it provides the smallest tick period possible. Execution with the steady clock utility was examined, however, the principle benefit of the steady clock is that the OS cannot interrupt or back-update clock times, but this is insignificant on the distributed nodes of the server as they hold no other tasks, so the precision utility was more appropriate.

4.4 Updating the TTL to the GPU/CUDAfication

The principle concern of migrating the TTL to operate within a GPU context was if there might be any library function within the TTL that was unsupported in the kernels, as well as how the template metaprogramming would possibly impact the NVCC's compiler heuristics.

A design objective for the TTL was to be an environment agnostic API, i.e. there should be no or minimal distinction for the client between developing for a host or device environment. This meant all functions needed to be invocable within the context of the device, requiring markup for the NVCC to recognize which parts of the code the NVCC needed to compile into cubins and which needed to be forwarded to the host side compiler to generate regular binaries. In general this meant all functions, from the constructors to operator overloads, needed to be decorated, and a handful refactored.

It was unclear in early development to what extent the TTL would need to be refactored to operate within the GPU, because it was unclear to what extent the NVCC was able to capture in a device environment all the C/C++ library functions that operate in the TTL host-side, given that the NVCC does not actually compile host-side code but forwards it to a host compiler, clearly indicating that the NVCC's functionality is not built upon the other compiler but is a separate and potentially limited extension.

First and foremost was the question of how, if at all, the NVCC would handle templated compilation for the expression tree structures the TTL would be building. The reason for this suspicion was that the ISO accepted the C++ 11 standards in 2011 [47] which provided the essentials of metaprogramming capabilities (although there have been extensions since), but it wasn't until late 2015 that CUDA had support for any kind of metaprogramming [48]. There are also certain library features that would, in general programming parlance, be considered essential, e.g. error catching and throwing, yet which are impossible on the GPU (albeit at least in that instance for understandable reasons e.g. how would one throw an error in a single GPU thread? Should it be thrown back to the host and stall all work, or, because all threads in the warp are driven by a single control unit, should it stall all the threads in a war [which assumes it can be caught]? How would one implement a "try" clause without the speculative execution units a CPU has and which the GPU doesn't?). It was unknown what other similar incompatibilities between host and device code compiling through the NVCC might reveal in the TTL.

4.4.1 Linear Solver

Additionally, the TTL did not have an independent linear algebra package or solver for the DMPP system implementation that parallelized computation through MPI. Rather, that functionality was provided by Linear Algebra PACKage (LAPACK) and Basic Linear Algebra Subprograms (BLAS). Originally this was an excellent decision, as both BLAS and LAPACK are well known software packages, widely used in HPC because of its excellent performance – the original LAPACK article [49] has been cited over 700 times, and an update to the BLAS library has been cited more than 690 times (for which one of this thesis's committee members was co-author, Andrew Lumdsaine) [50]. This would allow easier integration for other researchers who might wish to use the TTL when originally developer for distributed memory systems, however, neither BLAS nor LAPACK have direct support within CUDA.

CUDA does offer their proprietary implementations of BLAS and LAPACK, CuBLAS and CuSOLVER, respectively, both adopting the APIs of their namesakes, however, neither were well suited to integration with the TTL as a solver for the API the TTL was intended to have. The tensors needed by PGFEM are small and dense, overwhelmingly rank 2 or 1, less than a handful being rank 4. The parallelization strategy within the GCM was that each thread should process a single tensor object. In contrast, cuBLAS is intended for and best served by individual large matrices, not a multitude of small dense ones [51]. While cuBLAS can manage multiple independent matrices via CUDA streaming, this wouldn't operate well at the scale the TTL would need to in PGFEM (or most any other finite element modeling). Doing multiple matrix-matrix multiplies or solutions would require feeding the data into the GPU via a stream, but "The application [must] conceptually associate each stream with each task" [52] i.e. each tensor would need its own separate stream. The bottleneck for this would be orders of magnitude out of scope for the number of tensor objects the PGFEM needs to examine.

Instead, an in-house solver was developed. This was built from a high-performance BLAS and LAPACK package Dr. Andrew Lumsdaine provided. It implements LU factorization to determine inverses and solve systems and incorporates a partial pivoting strategy to guarantee numerical stability. This author contributed a rook pivoting routine to provide a more numerically stable solver than partial pivoting with near partial-pivoting computational efficiency, as well as a QR factorizing and Cholesky solver.

4.5 Building Test Programs

All test code is available with TTL on github at https://github.com/C-SWARM/ttl/

4.5.1 Unit Testing

As mentioned in §4.3, developing the testing capabilities to examine GPU functionality was built using the GoogleTest framework. This provided a foundation for the test subroutines as it has an easy API to implement testing, can pinpoint conflict areas in debugging, as well providing a clean output of expected outputs from given inputs using a macro-like system (Figure 25)

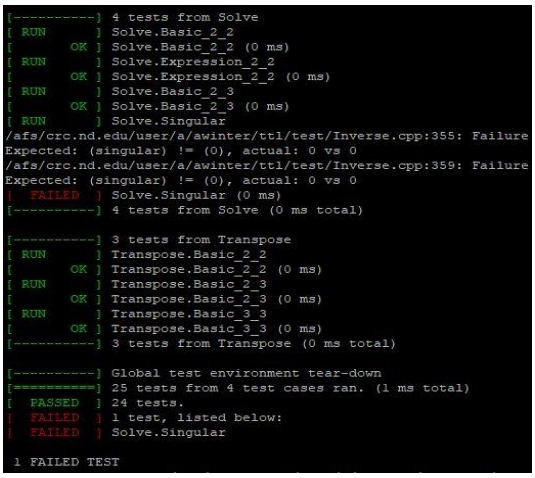


Figure 25. Failed GTest. In this instance, the GTest actually failed to resolve correctly.

Broadly tests were broken down categorically to evaluate the different ways the TTL might be used. This meant examining tensor operations and expressions, the library's syntactic coherency for expected expressions, and the stability of the tree structures it builds to mimic tensor data. Testing tensor operations and expressions meant examining that tensor objects could be properly initialized, basic linearity properties (multiplication/division, addition/subtraction, modulo) tensor operations (inner and outer products across various tensor ranks). Examining the lexical coherency meant checking that compiler deduction via autotyping was consistent in generating entirely new tree structures (e.g. from two given rank 2 tensors of dimension 3, could the compiler deduce the structure needed of a third autotyped "entity" from an expression of the former 2?).

4.5.2 Performance Characterization in PGFEM

Selecting an area of the PGFEM to parallelize for the GPU was a difficult task. Although several modules of the PGFEM utilize the TTL, the ways in which data migrated between different submodules and were interdependent was sprawling (Figure 26). Determining how to quantify TTL's performance and potential speedup in the GPU required examining submodules that were small enough they could modified quickly yet independent of other modules, and the GCM seemed the best fit in this regard, as generating data for it

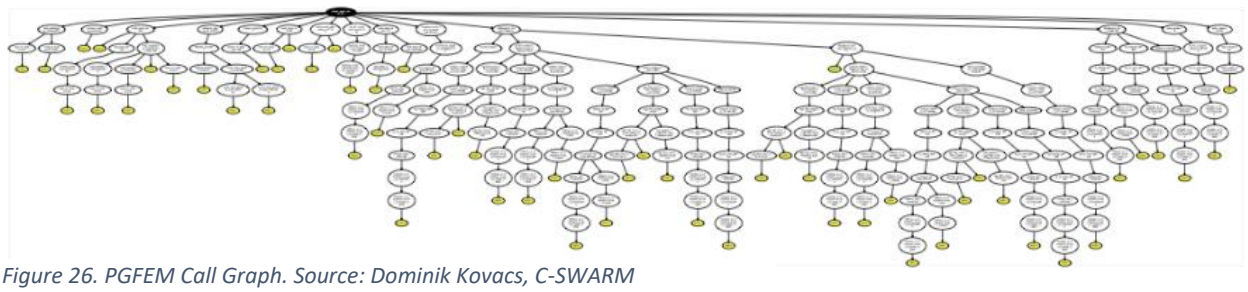


Figure 26. PGFEM Call Graph. Source: Dominik Kovacs, C-SWARM

to process was easier and working with it separately required less refactoring of the code, which would alter a true assessment of the performance.

In parallelizing the GCM, each crystal grain tensor element would be handled by one thread. This would allow each tensor object to exist independent of the others and was the smallest possible coherent parallelization of the intended computation (updating the elasticity of the grains as they deform with stress/heat) in addition to being an intuitive one. The hope with this was that it would not only improve legibility by operating within single tensor objects, but maximize total thread occupancy to concurrently utilize as many GPU resources as possible when the GPU was invoked.

Examination of the performance would be handled by timing testing for general profiling as well as the nVidia Profiler (NVProf) for diagnostics. NVProf is a graphical profiling tool which displays a timeline of an application's activity, both host and device, and includes an automated analysis engine to identify optimization opportunities.

4.5.3 Microbenchmarking

The microbenchmarking tests were necessitated after examining performance of the GCM with TTL implemented within GPU kernels, specifically after the performance of the TTL-augmented GCM operating on a Volta architecture GPU did not resolve the TTL's lagging performance in comparison to a raw loop implementation. While the unit testing in §4.5.1 would verify that the TTL compiled through the NVCC could build the appropriate operations from source code and that those outputs were valid, there were no guarantees about the quality of that performance, and as examined in §5.2, there was a discrepancy in the performance of the TTL and the raw loop implementation in the GCM that warranted further investigation.

```
Testing DOUBLES for 1000000 iterations.

Operation      Dim     Rank    Device  Format  Data (Mb)       runtime
Inner Product  2       1->0    CPU     TTL     8.00006         0.000471412
Inner Product  2       1->0    CPU     Raw     8.00006         0.000613148
Inner Product  2       2->0    CPU     TTL     8.00009         0.000451581
Inner Product  2       2->0    CPU     Raw     8.00006         0.000832946
Inner Product  2       3->0    CPU     TTL     8.00015         0.000457366
Inner Product  2       3->0    CPU     Raw     8.00015         0.00158258
Inner Product  2       4->0    CPU     TTL     8.00028         0.000464631
Inner Product  2       4->0    CPU     Raw     4.00028         0.00317975
Outer Product  2       1->2    CPU     TTL     32.0001         0.00375735
Outer Product  2       1->2    CPU     Raw     32.0001         0.00367538
Outer Product  2       1,2->3  CPU     TTL     64.0001         0.00729691
Outer Product  2       1,2->3  CPU     Raw     64              0.00739557
Outer Product  2       1s->3   CPU     TTL     64.0001         0.00773477
Outer Product  2       1s->3   CPU     Raw     64              0.00770963
Outer Product  2       2->4    CPU     TTL     128             0.0145283
Outer Product  2       2->4    CPU     Raw     128             0.0149085


Inner Product  3       1->0    CPU     TTL     8.00007         0.000456972
Inner Product  3       1->0    CPU     Raw     8.00007         0.000680564
Inner Product  3       2->0    CPU     TTL     8.00017         0.000455014
Inner Product  3       2->0    CPU     Raw     8.00007         0.00176158
Inner Product  3       3->0    CPU     TTL     8.00046         0.000453722
Inner Product  3       3->0    CPU     Raw     8.00046         0.00529415
Inner Product  3       4->0    CPU     TTL     8.00132         0.0471839
Inner Product  3       4->0    CPU     Raw     4.00132         0.083263
```

Figure 27. Microbenchmark output

Whereas examination of the TTL in the GCM was targeted at an implementation environment which necessarily had many more variables from which performance might suffer, the microbenchmarks were developed to isolate the TTL's performance for specific operations across a spectrum of tensor ranks (i.e. number of multidimensional arrays), expression trees, and data types (to examine the impacts of memory alignment on register pressure). The microbenchmarks focus on the distinctly tensor-ish behaviors that were more likely to encounter problems with the compilers' heuristics, i.e. inner and outer products (essential to many

of the TTL's other functions) across varyingly complex expressions and bindings. An example of the microbenchmarking results is shown in Figure 27.

4.5.4 Assembly and Compiler Testing

The results of the microbenchmarking indicated either that the testing was flawed, that the problem lay with the GCM implementation, or that the compiler itself was the next potential culprit for the performance disparity, however, the exact nature of any such inconsistencies was unknown, and so an examination of the assembly of the binaries and cubinaries was necessary.

In order to test over the range of data types, tensor ranks, and dimensionality used in the microbenchmarking suite, considerable templatization was used. The assembly tests were built with no templatization except that intrinsic in the TTL to limit compiler heuristics and isolate those heuristics being used in constructing the TTL objects. It was also reduced to examine those tests that were most commonly encountered within PGFEM, 1st and 2nd order tensor inner products.

A simple text comparator was used to identify points of divergence in the assembly code generated by the different TTL configurations and compilers and then assembly instructions were marched through by hand to compare implementations. Compilation was performed under the GCC and NVCC.

5. Results

Certain figures, either too numerous or requiring too much compaction to be fit within the body of this work for clear details beyond general patterns, are provided in Appendix A. Additional Figures.

5.1 Unit Testing

Once updating the TTL to operate within the GPU was complete, the first task was verifying that the CUDA decoration and refactoring of the essential functions was generating valid outputs and behaving as expected. This was completed with over 400 lines of modified TTL code and 8000 lines of unit test code building the kernels and host code to drive testing, as well as the creation of an independent linear solver library. Sample output from the GTest unit tests are provided in Figure 28. TTL features operable within the GPU are expressed in Table 1.

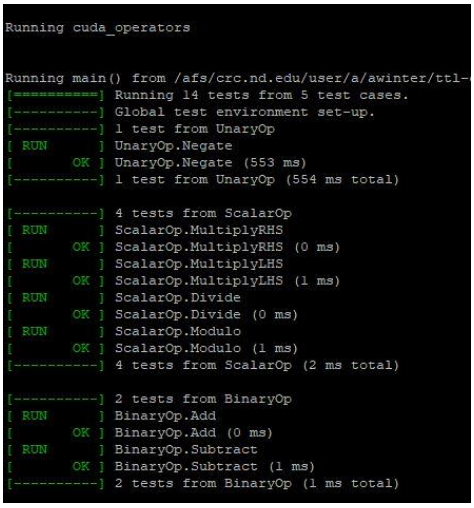


Figure 28. GTest demonstrating successful CUDA build for operator testing.

It is noteworthy that the bussing time for even these extremely small data sizes (often less than 100 bytes) always incurred a penalty (typically approximately 500 ms) in the first unit test as the cost of bussing data across the PCIe, whereas the unit tests themselves typically ran on the order of 2-3 ms at most, many operating below the measurable threshold. Hopefully this provides the reader with some context for how many orders of magnitude more expensive it is to migrate data between the GPU and CPU than it is to hold data to the GPU and do as much processing as possible with data in that context.

5.2 PGFEM and GCM Performance Analysis

An early characteristic run of the GCM on the LLNL Pascal architecture utilizing the TTL compared to a baseline implementation of the GCM implementing raw for-loops is shown in Figure 29. Performance suffered in the GPU in comparison to a host-side MPI driven solution, which initially was expected given that the MPI model was more well understood to the PGFEM developers and therefore optimized, nor is such a result unexpected from the literature [53] where there is "evidence that programmers should rethink algorithms instead of directly porting them to GPU", despite the general expectation that in porting to GPUs there will be some manner of improvement for sufficiently

Table 1. CUDA-capable TTL Features.

Binding	✔
Delta tensor	✔
Determinant	✔
External memory operations	✔
Linear solver	✔
Levi-Civita	✔
Identity	✔
Index behavior	✔
Index mapping	✔
Inversion	✔
Scalar interactions	✔
Tensor operators	✔
Tensor constructors	✔
Transpose	✔

parallel systems regardless of potentially poor configuration (and for which there is increasing evidence to be suspect of the validity of such sweeping improvements generally) [54].

What was less expected was that only the for-loop implementation scaled better on the GPU than MPI, and moreover, it was not until full thread occupancy of the P100's 3584 cores that the GPU showed any performance gain over the MPI implementation (Figure 29) (admittedly it is somewhat misleading to say that that is full occupancy, as that would actually have occurred earlier due to the branch-points in the Newton-Raphson solver requiring predication, i.e. each possible outcome would hold a thread).

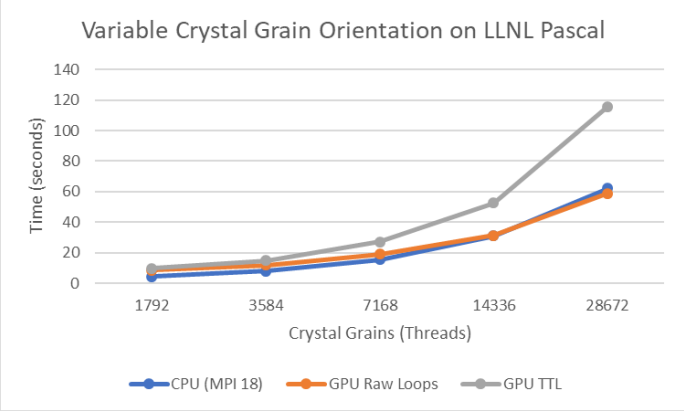


Figure 29. Variable Crystal Grain Orientation on the Pascal Server

After cleaning up the code and moving away from unified memory in the GCM, NVProf was used to examine where intransigent stalls might be occurring. In certain block and thread configurations we were able to push occupancy very high, in the range of 80-90% (Figure 30) however, two areas that appeared consistently as targets for improvement were memory (both for throttling and dependency) (Figure 31) and warp execution efficiency (Figure 30).

More than two thirds of the stalls could be accounted for in memory throttles and memory dependence (Figure 31). Memory throttles occur when a large number of pending memory operations prevent progress, the solution to which is generally to combine memory transactions or better utilization of shared memory. In contrast, memory dependence occurs when load/store operations can't be made because either the resources are fully utilized or too many such operations are occurring simultaneously (effectively a load balancing problem). This can be improved by optimizing memory alignment, not in the historical sense of using different byte-sized type as in floats vs doubles (although this might help in the instance of register pressure) as that kind of memory alignment is managed best via padding by the compiler; rather, it's using the intrinsic variables CUDA provides for indexing to parallelize intuitively rather than through unorthodox pointer arithmetic [55]. Moreover, improving access patterns, i.e. either

Grid Size	[240,1,1]
Block Size	[128,1,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
Efficiency	
Local Memory Overhead	22.2%
Warp Execution Efficiency	65.1%
Not-Predicated-Off Warp Execution Efficien	62.6%
Occupancy	
Achieved	86.3%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

Figure 30. NVProf Diagostic.

altering the order of executions to improve locality, or altering the striding mechanism of multidimensional array indexing data structures (such as TTL) are generally the best means to improve dependency bottlenecks.

Warp execution efficiency in Figure 30 was unfortunately an ambiguous metric in our context. Normally it reflects how many threads in the warp are active, which is usually a reflection of conditional control, which in this context might reflect divergence within the Newton-Raphson residual calculation, however, if the compiler was failing to unroll the TTL loop structures and was instead updating the IR to jump to other areas in the memory, the examination would correspond to either branch or predicated warp threads (explaining the low warp efficiency) and could also possibly account for the memory dependence stalls, as the loops otherwise normal access pattern would then be disrupted. The possibility however that the compiler could fail to unroll the loops however seemed low given that the TTL builds the loop structures recursively from the template expressions, and yet it couldn't be discounted.

Figure 31. NVProf GCM Stall Analysis

Unfortunately, the NVProf doesn't prognosticate from the topics it outlines as priorities are most likely to provide the most significant returns for improvements. For example, with regards to memory dependency, the operations being performed by the TTL broadly are general matrix multiplication (GeMM) and for which array striding and indexing are highly regular, and in the TTL's case, must be known at compile time, all of which diminish the likelihood that access patterns were a source of performance bottlenecking and indicating that this was likely a load balancing problem with utilizing the same resources too much. Other work with GeMM operations, some explicitly for tensor operations on GPUs, [56] are rarely bottlenecked by this.

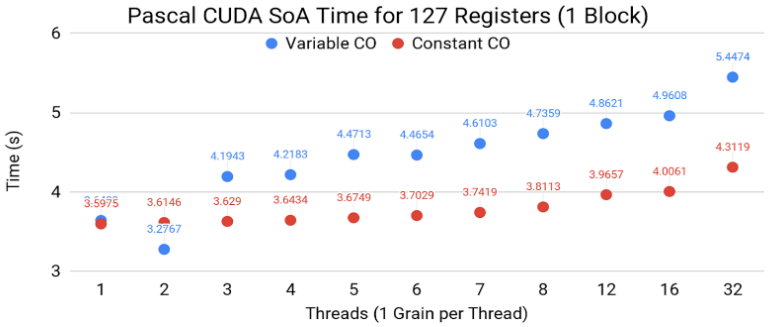


Figure 32. Crystal Orientation Driven Warp-Thread Divergence

In examining areas causing performance stalls, it became apparent that the orientation of the crystal grains being processed by the GCM were not uniform, and this lack of uniformity would result in significantly differing compute times for each thread to resolve the residuals, creating non-uniform branch predictions and stalls based on the divergence of threads within the warp. This would also prevent warps from surrendering their context regardless of thread occupancy and disrupt latency hiding, as a mostly unloaded warp that had solved all but one thread would continue to churn. While crystal orientation cannot be guaranteed in deployment runs for experimental purposes, re-orienting the crystal grains to be uniform would normalize divergence and determine to what extent divergent iteration from Newton-Raphson was the bottleneck in GPU TTL GCM performance. Figure 32 shows how normalizing crystal orientation reduced branching and improved thread scaling within the warp, which is essential to strong GPU performance.

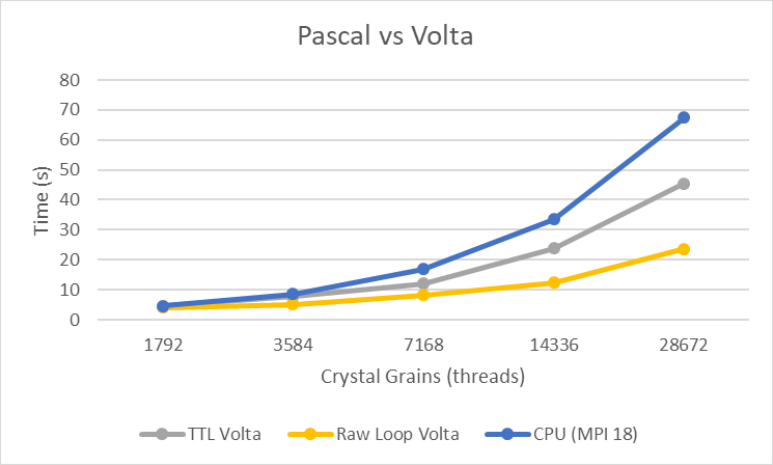


Figure 33. GCM Run on Volta Architecture

Even with normalizing the crystal orientations to balance the load however, other elements of the code remained which controlled flow through branching that might impact performance. Evaluating on a Tesla architecture GPU would reduce this penalty, as Volta allows independent thread scheduling. From nVidia's white paper on the volta architecture [57]:

"Volta's independent thread scheduling allows the GPU to yield execution of any thread, either to make better use of execution resources or to allow one thread to wait for data to be produced by another. To maximize parallel efficiency, Volta includes a schedule optimizer which determines how to group active threads from the same warp together into SIMT units. This retains the high throughput of SIMT execution as in prior NVIDIA GPUs, but with much more flexibility: threads can now diverge and reconverge at sub-warp granularity, and Volta will still group together threads which are executing the same code and run them in parallel."

The results of the TTL-GCM running on Volta architecture is shown in Figure 33. While we then saw performance gains comparable to the literature for porting to a GPU in the raw loop implementation, [54], and there was commensurate improvement with the TTL implementation, clearly something in the source of the GCM utilizing TTL was holding back performance (Figure 34) but there were many means by which this could occur, and the presence of one didn't necessarily preclude the others, and so a better means of isolating the TTL to identify where, if anywhere, problems might lie in the library was needed.

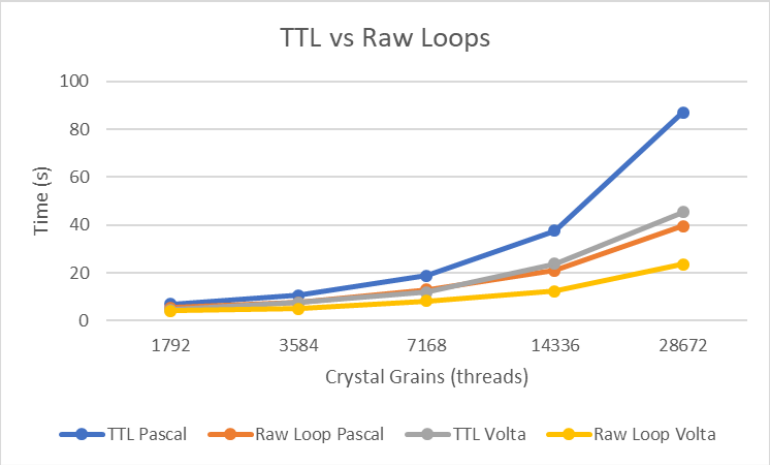


Figure 34. TTL vs. Raw Loops Across Architectures

5.3 Microbenchmarking

The question emerging from Figure 34 was where and potentially why was the TTL consistently less performant than the raw loop implementation on the GPU, despite both correcting for one of nVidia's most commonly-cited sources of poor GPU performance (divergent branching) [58] by restructuring the data to be near-uniform in convergence, as well as characterizing on architecture that would smooth any other sources of warp divergence. When the NVProf diagnostics didn't provide certainty in answering where the bottleneck might lie, at least not in terms of whether the problem were within the looping structures the TTL was generating, a close examination of the TTL with microbenchmarking became the best way forward to isolate sub-optimal code.

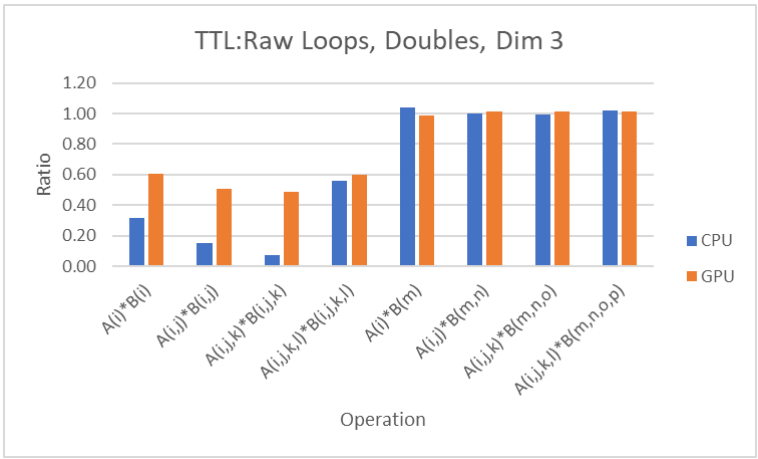


Figure 35. Microbenchmarking, Doubles, Dim3 Tensors

Examination across various tensor operations, expressions of varying order complexity, data types, and dimension revealed several patterns. First, at the level of the microbenchmarks, performance across the GPU and CPU for outer products examined as the ratio of the TTL to raw loop implementation runtimes

were functionally identical (Figure 35) within measurement precision, regardless of virtually any parameterization, i.e. independent of: complexity of parsing the expression, the order of the expression, (Figure 35), the data type(Figure 36), dimensionality, all were functionally identical in performance, indicating outer product expressions were an unlikely candidate as a bottleneck.

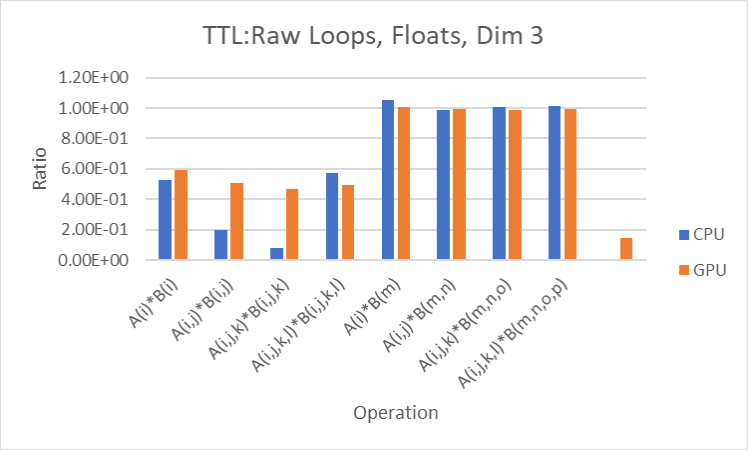


Figure 36. Microbenchmarking, Floats, Dim 3 Tensors

With regards to contraction and inner products however, there was a marked difference in microbenchmarking, yet it was the TTL consistently outperforming the compiler's raw loop implementation. Figure 35, Figure 37, and Figure 36 depict the ratio of TTL:Raw loop performance, so any column values less than 1 indicate TTL is more performant, values above 1 indicate the raw loop was more performant and in agreement with the GCM results. This was true across: every expression, requiring different levels of compiler heuristics to bind the expressions into different loop structures; different compute complexities, which would require different memory accessing patterns and caching; and different dimensionalities and data types to capture (Figure 35, Figure 37, and Figure 36). Such a strong inversion of performance to favor the TTL was not wholly surprising in the larger body of work with TTL – previously in work beyond the scope of this paper, it was shown that the TTL in MPI was much faster than certain loop structures. Yet this directly contradicted the in situ tests done with the GCM, raising further questions which could only be resolved with a close examination of the assembly code.

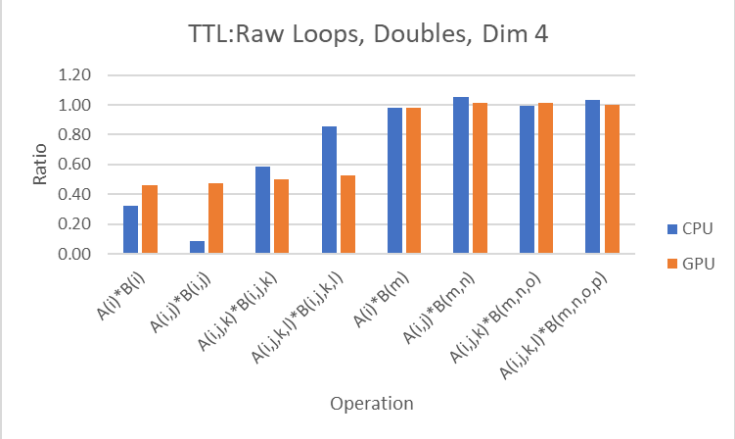


Figure 37. Microbenchmarking, Doubles, Dim 4 Tensors.

These results pushed the question back on whether the problem was intrinsic to the TTL and placed focus on either the compiler or potentially all the way back to the ancillary changes in the GCM needed to implement the TTL. If the problem was within the TTL, the next logical place to examine was the compiler, under the assumption it was failing to optimally construct the loop structures from the TTL expression trees. Were that the case, the question was then where or under what circumstances was the compiler not properly optimizing. Examining across not only tensor expressions but also different compilers, removing any non-TTL metaprogramming, and using the lightest code framework to reduce compiler heuristics might provide some insight into what was motiving these results that conflicted with the PGFEM performance analysis.

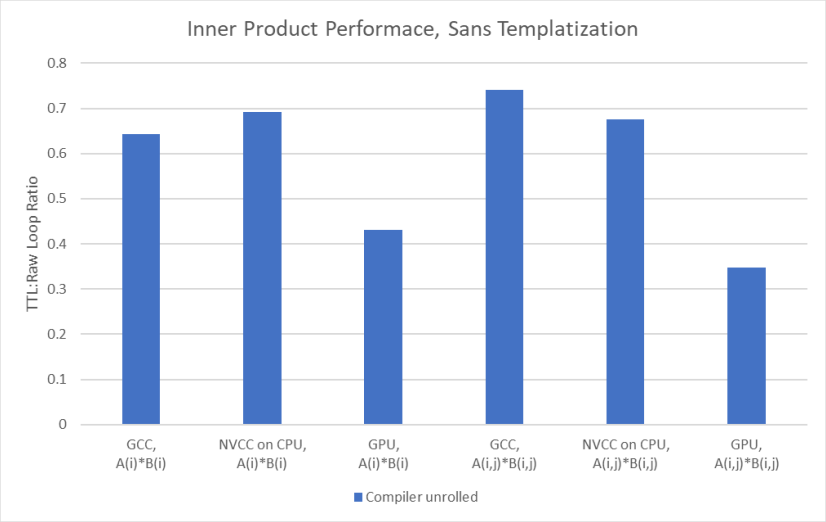


Figure 38. Inner Product Performance, All Test Templatization Removed.

5.4 Assembly and Compiler Testing

From the inverted performance of the TTL in the microbenchmarks in comparison to the GCM, the question of the compiler's heuristics became the focus. The fact that the TTL had been at least as performant as the raw loop structures in outer products and in the inner product expressions the TTL was consistently and sometimes overwhelmingly more performant begged the question of how was the compiler optimizing the loops it was fed, both in the microbenchmarking (in case the results were in some unaccounted way artificial) and in the GCM, and this could only be verified through the assembly code. However, examining the complicated GCM assembly would have been laborious, and while the microbenchmarking tests were not as large to sort through, much of the test was built through templatization, which, given the hypothesis was that serial compilation was impeding

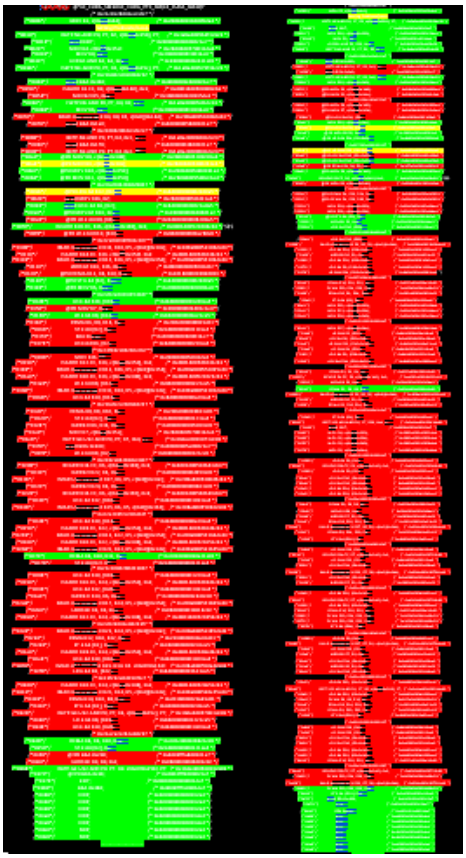


Figure 39. PTX of raw loop implementation vs TTL. Lefthand is the raw loop implementation

optimization, potentially confounded by templatization, it was necessary to remove this compounding factor.

```
__global__
void ttlKernel(double *C, double* dA, double *dB, const unsigned int iter){
    ttl::Tensor<1,3,double*> A { dA };
    ttl::Tensor<1,3,double*> B { dB };
    for(unsigned int ii = 0; ii < iter; ii++)
        C[ii] = A(i) * B(i);
}
```

Figure 40. TTL Assembly Kernel

The performance for the most commonly encountered tensor inner product expressions in a simplified test routine absent any metaprogramming are shown in Figure 38, the same pattern as examined in the microbenchmark tests for inner products in Figure 35, Figure 37, and Figure 36. These tests removed all templatization from the testing program, the only remaining metaprogramming being in the TTL, and examined direct compilation via the GCC on the host, compilation for the host via the NVCC, and NVCC compilation for the GPU. Across every compiler and environment combination, the TTL implementation remained superior.

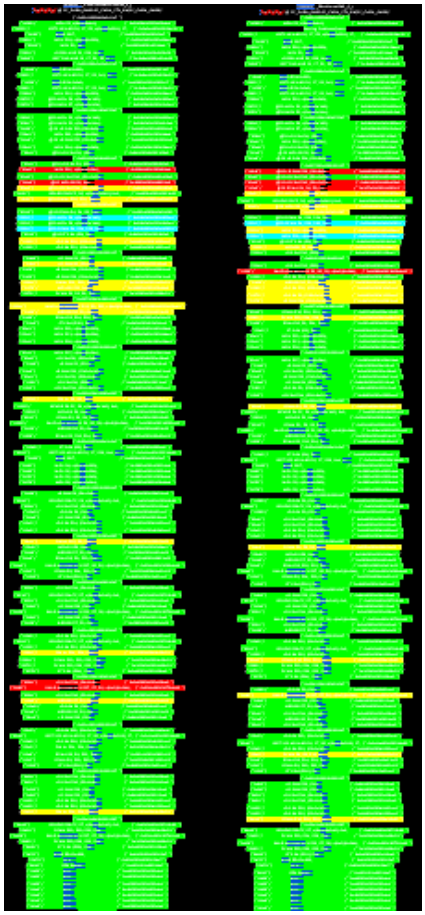


Figure 41. Unrolled vs TTL PTX. The unrolled implementation is the lefthand.

While many more expressions were examined, the simplest to discuss is the inner product of two rank one tensors across a bound index, $m_i n^i = c$. Sample assembly code contrasting a raw loop implementation to the TTL in a GPU kernel is presented in Figure 39, respectively, and their source kernel codes are presented in Figure 42 and Figure 40. In Figure 39, green bars indicate a point at which the instructions are using the same code to perform the same operation, yellow indicates an instruction where the compiler has made an equivalent substitution from the ISA but the operation being performed is fundamentally the same, red indicates a block of code where some other task not present in the opposing assembly has been injected, and cyan (not pictured in Figure 39 but in Figure 41).

```
__global__
void unrolledKernel(double *C, double *arrA, double *arrB, const unsigned int iter){
    for (unsigned int jj = 0; jj < iter; jj++){
        C[jj] = arrA[0] * arrB[0] + arrA[1] * arrB[1] + arrA[2] * arrB[2];
__global__
void rawKernel(double *C, double *arrA, double *arrB, const unsigned int iter, const unsigned int dim){
    for (unsigned int jj = 0; jj < iter; jj++){
        for (unsigned int ii = 0; ii < dim; ii++){
            C[jj] += arrA[ii] * arrB[ii];
```

Figure 42. Raw Loop Assembly Kernel

In examining the two, it became clear that the compiler was failing to unroll the raw loop instructions. The raw loop implementation was using branching to move the instruction counter upstream to rerun the same instructions. The question then was how to normalize the performance such that their performances were identical. Enforcing a manual unrolling of the loop to avoid instruction branching was expected to resolve the disparity. The unrolled loop source code is presented in Figure 43. The PTX assembly code this generated is presented in Figure 41.

Aside from the different register naming between the two, when manually unrolled the PTX are virtually identical across 155 instructions, with a handful of order substitutions (yellow) and, in two cases, instruction order exchanges. The raw loop implementation tended to use MUL instructions where occasionally the TTL implementation would utilize fused multiply-addition (DFMA), which directly adds a product to a destination register. In terms of actual performance as well, the manual loop unrolling normalized performance across inner products to be consistent for both the TTL and loop implementation (Figure 44).

From these results however rose the question of the host's compiled code. As explored in §5.3, examining why the compiler would implement different instructions, especially when template metaprogramming was

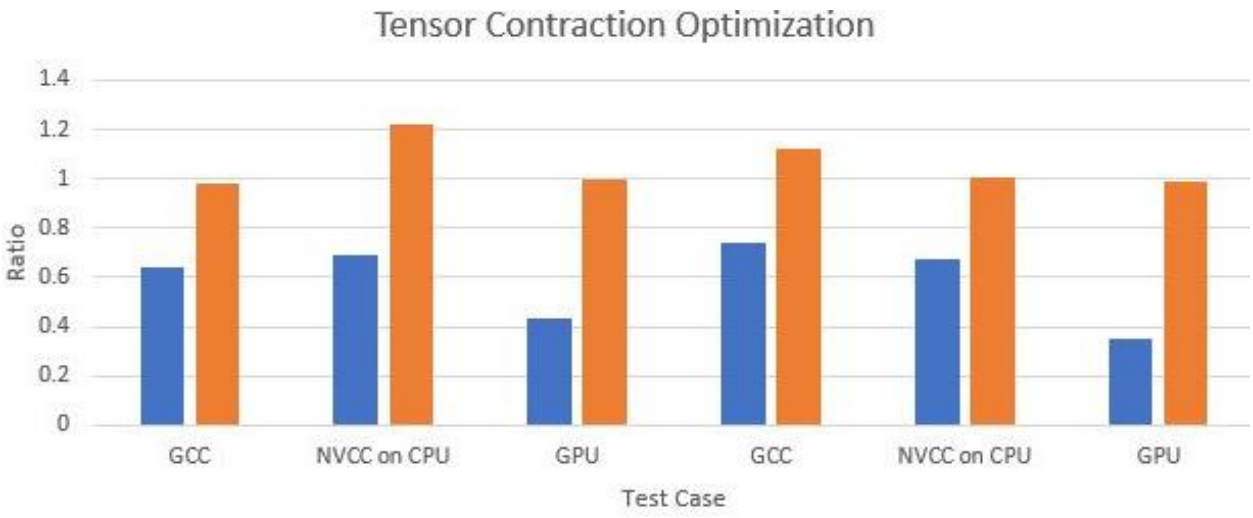


Figure 44. Normalized TTL/Unrolled Loop performance.

removed, begged the question of what was NVCC parsing doing to host code when pushed through the NVCC for implementation host side, which nominally should have no impact, in comparison to direct compilation through the host compiler.

In examining the host x86, the exact same loop instructions were used as the kernels as depicted in Figure 40 and Figure 42 albeit with appropriate CUDA markup indicating the destination architecture. The comparison of the x86 assembly for the host code in a raw loop implementation and TTL is shown in Figure 45 and Figure 46. Manually unrolling was able to correct for this in the assembly, shown in Figure 47 and Figure 48, and reflected performance wise in Figure 44.

Figure 45. Host x86 assembly, raw loop implementation

Figure 46. Host x86 assembly, TTL implementation

Figure 47. Host x86 assembly, unrolled loop implementation

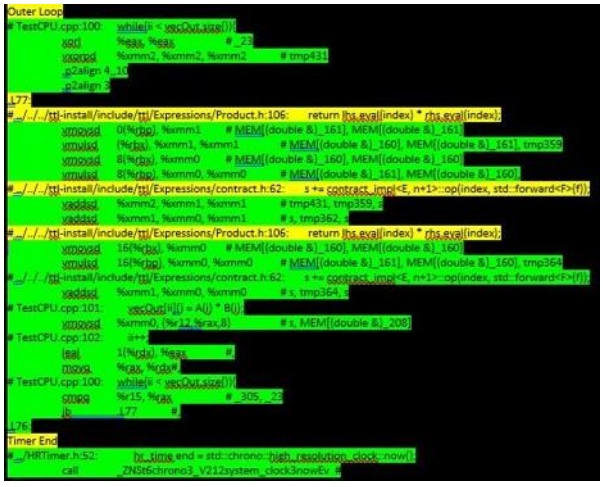


Figure 48. Host x86 assembly, TTL implementation

6. Discussion

6.1 Unit Testing and Porting

It must be said that, aside from the minutiae of decorating the source code with CUDA markup, adaptation of the TTL to operate within the GPU was considerably more straightforward than initially anticipated, and, for being a wrapper compiler, the NVCC was remarkably serviceable and quite performant, at least in terms of parsing a complicated templated metaprogramming source code and generating functional cubins that provided accurate output. In terms of adapting an existing library to operate within the GPU, the process was overall none too difficult, and the outputs were exactly as expected. That said, the *optimization* of that parsing in generating the assembly code for the GPU left much to be discussed in 6.4.

In this author's experience, Googletest is an ideal testing platform from a usability standpoint. The work with unified memory in the unit tests was simple and streamlined in terms of legibility and ease of use – nothing can make a user appreciate the value of not managing garbage collection or something like the Python Memory Manager quite like working in C/C++, and even moreso in a GPU/CUDA context.

6.2 GCM Performance

"Optimizing memory on a GPU is somewhere between science and art".

-Peter Messmer, Nvidia Senior Manager

While it must be acknowledged that the initial results on the GPU were not the orders of magnitude reductions claimed in some GPU implementations in the literature [59] [60] [61] [62] [63] this does not necessarily reflect on the core interest of this work, the TTL (as will be discussed in 6.4), nor is it necessarily a reflection of the correctness of GPU parallelization for the PGFEM - there have been other works utilizing similar constructions as the TTL does under the hood in template expressions that have been successful in GPUs [64]. Moreover, the excellent work done by Lee et al. indicates strong skepticism in expecting such sweeping improvements for GPU parallelization are generalizable. [54].

It must also be said, there are reasons to be suspicious of the extreme performance gains seen in publication from various noteworthy GPU parallelization papers in the literature as not strictly universal or uniform in all applications [54] in general. An excellent paper by Lee et. al discussed in detail the reliability of some of these extraordinary results, concluding scaling on the order of 2x to 9x is generally a much more reasonable expectation for improvements in GPUs over their CPU counterparts for comparably optimized code. As Tim Warburton has stated, "when you see a 100x performance gain in the GPU, but ordinarily we would expect a 2x to 9x speedup from the hardware specifications, what does that tell you? Your CPU code was [expletively poor]." [21]. In adapting an existing code base to perform on the GPU, there is necessarily

some refactorization, and often as developer go to re-examine code, additional optimization is done that, had nothing been done to retrofit source code to work in a GPU environment, researchers may still have seen considerable improvements.

A number of the NVProf analytics didn't strongly align with poor TTL performance per se and indicated ancillary aspects of refactoring the GCM to enable the TTL expressions were potentially negatively impacting performance or the compiler's ability to optimize through those changes. Specifically from Figure 31, local memory overhead was nearly at maximum – that reflects how the data was managed to launch kernels rather than the operations of the TTL. Moreover, it was a point of concern in the testing that performance struggled to develop the characteristic parallel scaling expression. Speaking broadly, parallel high performance computation is generally limited by Amdahl's Law (or one of its derivatives like Gustafson's Law, or Amdahl's law adapted for modern multi-core architectures [65]). Amdahl's law inversely relates the total potential improvement possible from parallelization to the total computation that must be performed serially), and most HPC profiling follows a characteristic asymptotic curve representing the fundamental serial limitation of the examined algorithm regardless of compute resources thrown at it. Amdahl's model doesn't completely fit within a all contexts (cf. Gustafson's Law), and particularly a GPU context, as ideally, nothing should be processed serially on the GPU and any such unavoidable computations should (potentially be offloaded to the CPU, but analogously, there are GPUs bottlenecks to parallelization akin to serialization such as bussing data (although this can be slightly mitigated with streaming to allow the bus to load data as it is resolved and shuttled off the GPU) and on-device memory capacity [54].

An interesting perspective to examine the actual cost effectiveness of the implementation of the TTL for the GCM is to examine it alongside the cost in silicone for the different hardware, which gives a better perspective on how much performance lagged. nVidia carefully manages the transistor count on GPUs to approximately balance the manufacturing cost – the additional FLOP capability of GPUs isn't a cost savings on its own per se, those transistors have only been redistributed from elsewhere on a CPU chip. Using the purchase price of the two units used to run the test (the P100 cost $7,374 [66] whereas the Intel Xeon E5-2695 v4 for host operations cost $2424 [41]), accounting dollar for dollar in silicon, the GPU performed even worse than Figure 34 might lead one to believe - excluding the cooling and power costs, scaling the runtimes by dividing by the

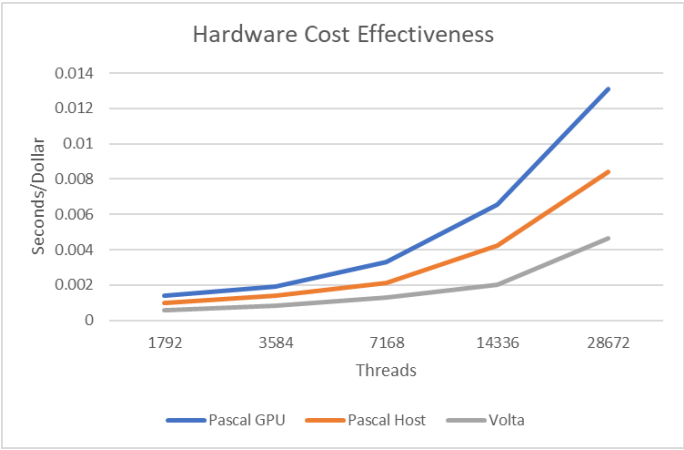


Figure 49. Cost Effectiveness of the TTL in the GCM

up front cost of the hardware results instead in Figure 49. Unless the compute time were exceedingly long, this would indicate, as currently implemented, the MPI host solution is more cost effective.

There are so many avenues by which GPUs can be optimized and for which such optimizations simply were not pursued for the GCM, yet properly utilizing all the memory resources of the GPU once data is loaded is strikingly valuable in optimization [34]. A number of variables that were material constants or could be utilized throughout all crystal grains simply were not implemented in the global memory space, which would crowd

6.3 Microbenchmarking

The pattern of inconsistent improvement in Figure 35, Figure 37, and Figure 36 was initially cause for concern about the validity of the test – a result that not only conflicted with the previous results, but strongly so in the opposite direction, and which did not seem to have stability in the extent of its performance improvement, sometimes indicate a poor test. However, upon evaluating the test and some consideration, it was a complete success for the goals of the microbenchmarking even if the outcome was unexpected – it clearly isolated a particular kind of tensor operation as a potential source of instability, and it did so consistently and independently of any other parameter, which would in most contexts indicate a consistent, solid problem in the implementation. This was strong validation for the decision to build a test to isolate inner product operations and examine the assembly produced from those operations.

Moreover, the pattern of "inconsistent" scaling – apparently improving scaling followed by precipitous losses of performance, but which still reflected overall superior performance - likely reflects a combination of the reduced memory locality from the larger striding needed in the data storage for the high dimensional tensors, but primarily the superior caching capability of the host. The smallest inner product only required 120 KB of data, and while there is 1.5MB of GPU cache available on the Titan Black, when distributed over the 15 streaming multiprocessors, each can only ever access 100KB of cache, so even the smallest operations couldn't fit fully within the cache. In contrast, the many MB of cache on the host CPU would allow much of the data to sit in cache. The point at which there is a precipitous change , such as the precipitous change between the third and fourth order inner product in Figure 36, reflects moving from one tier of cache to the next, as there is a similar adjustment between the second and third rank inner product in Figure 37 which has a larger dimension (and thus total memory requirement), and no such precipitous changes for any data type in the inner products with two dimension.

6.4 The Assembly and Compiler

The question of the GCC's role in NVCC compilation is an interesting one and in this author's estimation, a significant factor for the oscillating behavior of the TTL for the different test environments, which in

some sense exonerates the TTL's implementation from the performance seen in the GCM, but also potentially the most generally interesting result from this work.

While explained elsewhere in this work that the manner in which TTL constructs the loop structures from the expression trees makes the TTL itself seem an unlikely source of compilation complication, as the TTL builds those loop structures through variadic template recursion which should unroll them, in the context of the GPU that explanation may be less sturdy, because source code must be filtered through four layers of optimization heuristics - the NVCC's metaprogramming heuristics, the NVCC, the GCC metaprogramming, and finally GCC compilation. Poor loop construction could account for the performance disparity both because, as a truism, each additional tensor rank/depth of looping is exponentially expensive, and many of the loop structures needed by the GCM through TTL were multi-rank nested loops, and as expressed in §2.3.1.3.2, non-unrolled loops are points of control flow that could also account for the suboptimal warp execution efficiency shown in Figure 30.

There is a lot to suggest the NVCC should, on the scale of compilers, be robust - the CUDA toolchain uses EDG (from Edison Design Group) for the frontend [67], which is the same frontend utilized by the ICC (intel compiler) [68], and Intellisense for the Microsoft C/C++ (the Visual Studio compiler) (interestingly, nVidia utilizes the EDG as a legacy decision from purchasing the Portland Group, who prior to nVidia's acquisition had used EDG), followed by NVVM (derived from LLVM, a ubiquitous compiler frontend) for the translation of device code to PTX. Such large entities and disparate array of users should create large lists of bugs to narrow down the ways in which the compiler might struggle.

And yet, it is not difficult to discover discussion of various NVCC bugs focused on templatization which nVidia acknowledges when reproducible [69] or even entire git repositories dedicated to documenting unresolved issues. As stated by Thibaut Lutz, an nVidia engineer responsible for characterizing nVidia's compiler, after an extended lecture on the robustness of the NVCC concluded "all compilers are buggy" [67].

It is interesting to examine exactly how it was that unrolling the loops normalized performance between the TTL and different loop implementations. Comparing the instructions utilized in Tables 2 and 3, which compare the inner product occurring on the kernel for the TTL, raw loops, and unrolled loops, for both rank 1 and rank 2 inner product, both TTL and manual unrolling were interestingly far more reliant on loading memory than the raw loop implementation, with 2.3 times as many LD instructions, however, the TTL and manually unrolled implementations had half the branching, ¾ the predicated instructions, and 2/3 of the test condition instructions. While we normally envision loading and memory accesses as expensive in a host context, this clearly shows that with many times the instructions and direct accesses to memory, branching and predication are much more expensive on the GPU.

Moreover, a very interesting fact reveals itself from comparing the TTL and the manually unrolled instructions. Lexically, the two are different in the source but syntactically are very nearly identical, and overall, the compiler's heuristics do a good job of preserving this – the two have very nearly nearly identical instructions for data management and control flow, varying only with one additional test condition code in the rank one inner product. In the mathematical operations however, at a handful of locations, the unrolled loop implemented a DMUL instead of a DFMA – it scaled some set of operands rather than doing a fused multiplication-addition, where the product of two entities are directly added to a destination register (which highly resembles what we want in an inner product). In one sense, these are equivalent operations, the output is safely going to be the same between the two, but critically, *they are not identical*, and they demonstrate that the compiler, moving through the heuristics for the TTL and the atomically simplest syntactic expression of the same loop structure sans recursive expression templates, can and does generate different instructions of different performance for what is syntactically the most synonymous yet distinct source code.

Prior to examining the assembly, this author's understanding of nVidia's use of "wrapper compiler" was in the conventional sense of computational "wrapper", i.e. an entity which does not alter input but forwards that data to an API that can and does perform some computation or task, which in a compiler would, in general parlance, this author would argue, be understood to mean forwarding source code unaltered, and this sense of "wrapper" as unmodifying is clearly reflected in OpenMPI's discussion of MPICC, the compiler for the MPI distributed memory computing model:

Table 2. PTX Instructions in Rank 1 Inner Product

	Raw Loop	TTL	Unrolled
Data Management			
Load operations	18	42	42
Move	15	25	25
Stores	7	7	7
Operations			
DFMA	7	21	14
DMUL	0	0	7
IMAD	13	6	6
ISCADD	13	6	6
Control			
Compare Predicate	8	6	6
Branch	8	4	4
Test condition code	9	7	6

"Note that Open MPI's wrapper compilers do not do any actual compiling or linking; all they do is manipulate the command line and add in all the relevant compiler / linker flags and then invoke the underlying compiler / linker (hence, the name "wrapper" compiler)" [70].

There is no preprocessing of source code, only forwarding of compiler options, which is distinctly different from nVidia's use of "wrapper" compiler in the NVCC, which is not only responsible for the parsing of GPU metaprogramming and the compilation of the GPU assembly ("preprocessing"), it also must necessarily parse the host side metaprogramming to pass to the GCC:

Table 3. PTX instructions in Rank 2 Inner Product

	Raw Loop	TTL	Unrolled
Data Management			
Load operations	18	126	126
Move	18	25	25
Stores	7	7	7
Operations			
DFMA	7	63	56
DMUL	0	0	7
IMAD	16	6	6
ISCADD	15	6	6
Control			
Compare Predicate	8	6	6
Branch	9	5	5
Test condition code	9	7	6

"CUDA compilation works as follows: the input program is preprocessed for device compilation compilation [sic] and is compiled to CUDA binary (cubin) and/or PTX intermediate code, which are placed in a fatbinary. The input program is preprocessed once again for host compilation and is synthesized to embed the fatbinary and transform CUDA specific C++ extensions into standard C++ constructs. Then the C++ host compiler compiles the synthesized host code with the embedded fatbinary into a host object." [71].

If the reader is unfamiliar with fatbinary (it's an uncommon term in a post-Java virtual machine/Docker container/virtualization world), a fatbinary is an executable that can operate on multiple processor sets – this is needed because of the different architectures and ISAs of the device and host.

There are potentially two points of failure here. One is that the host compiler, in this instance GCC, only ever sees the extracted host code. Examining the intermediate files, the extracted host code is not always a verbatim copy of the host portion of the CUDA code: it frequently seems to be slightly re-stated in a functionally equivalent way, as discussed above in regards to Table 2 and Table 3, but functionally equivalent is rarely synonymous with equally optimal. This is in fact routinely leveraged by compilers to optimize, as is discussed in regards to the outermost smoothing loop and SSE but it can also step over optimal code for a sequence that is "safer".

From this work's perspective, there are two potential solutions to resolve the compiler's behavior. The simplest is to provide partial template specialization for those inner products among the most common tensor rank and dimensions (rank 1 and 2 tensors within the 3D space i.e. tuples of three elements) that are an explicit loop implementation. Rather than the recursive construction of loops utilized by other TTL objects through expression templates, these specializations would implement for loops. This specialization is currently in the most up-to-date TTL implementation.

This may seem counter-intuitive, as the recursive construction was, except in the GCM implementation, de facto superior, and was automatically unrolling the loops as demonstrated in Figure 41 and Figure 48, but

the reason is twofold: regardless of the underlying cause, there is an inconsistency in how the TTL optimizes between microbenchmark testing and implementation within a larger program. Whether or not specialization is more or less performant, consistency of performance is potentially more valuable in distributing the library to new developers who will have their own environments, and a performance penalty may be worthwhile if it guarantees consistency.

Second, if the problem is within the compiler, using the same structure as the loops rather than the recursive construction for the most commonly occurring tensor ranks and dimension guarantees they will be compiled similarly, and seeing as, if in fact the problem isn't in the GCM implementation, whatever heuristic the compiler is invoking to optimize the loop structures at that level of complexity is a superior working solution. That said, it must be acknowledged that in some sense this is not a tidy solution and steps around a design goal of the TTL to be problem space (i.e. material science) agnostic, as the assumption about what constitutes high-occurrence tensor rank and dimensionality are rooted in the PGFEM and a materials perspective. All the same, as an API, the distinction won't be something any clients would be aware of.

Another option would be to reconfigure the means by which TTL is compiled for GPU code. If the problem is in the NVCC's preprocessing of the template metaprogramming that's fed to the GCC, a test for a future work might be to examine fully separate host and device compilation for the client. Normally CUDA compiles the device code to be embedded into a host object, and nvlink links all the device code together, and then the host linker links nvlink's objects with all the host objects to create the executable. Separating the sources would cut the number of compiler heuristics used in half and allow only those heuristics that are suited for the environment in which the code will be run to do any parsing of its corresponding source code.

An interesting note on the cleverness of compilers, all implementations (the raw loop, unrolled loop, and the TTL) were utilizing the vectorized instructions for their operations – both were working through the xmm registers and v-INST instructions, which are reserved for the SSE instruction set on x86 architectures, which is a feature of the fastest compiler optimization flag, -O3. Initially this author believed that the compiler was vectorizing the operations of the inner product to move through the computation faster, however, after walking through the instructions, realized that they were instead being used to fill the outer batch loop - each of these unit microbenchmarks were examining operations that occur across 10s of nanoseconds, a time frame that pushes the boundary of what the high precision clock from the standard library can accurately measure, and especially when there may be system interrupt events that artificially inflate the runtime. To avoid this, each operation was invoked inside an outer loop that repeated the same instructions and loaded them to a very large vector that reflected the number of times the calculation needed to be done, both smoothing out any interruption events and buffing the runtime enough that reliable measurements could be made beyond the bounds of the measuring precision, something on the order of deci- or milli-

seconds instead of nano seconds. This was a very safe, clever optimization the compiler was able to implement, and it's also an example of the kind of unintuitive, unintended optimization that may be operating in an un-considered manner in the GCM as well and leading to such a disparate outcome in performance.

7. Conclusion

The TTL is currently fully operable within a GPU environment. This bears out across unit testing and implementation within a finite element package, the PGFEM. However, characterization of the performance of the TTL within a module of the PGFEM, the GCM, did not present the expected performance improvement – while performance improved with raw loop implementations on the GPU vis-à-vis a serial implementation in a distributed memory system utilizing MPI parallelization, performance under the TTL to represent the material tensors was less comparatively performant.

In order to determine exactly where, if anywhere, problems within the TTL might be occurring, microbenchmarking tests were developed to characterize specific operations of the TTL in isolation. These microbenchmarks revealed that a particular library operation, tensor inner products, were behaving divergently from the other TTL operations. Simplified, non-metaprogrammed tests were developed to examine assembly code for the host and device and determined that the original microbenchmarking tests were correct, but also that the compiler was in fact building different assembly code. This did indicate that there may be a problem with how the NVCC was parsing the source code that were likely at play in the GCM's sub-optimal performance of the TTL in a GPU environment within PGFEM. This did not however exonerate that ancillary refactorization decisions to accommodate the TTL weren't impacting the TTL's performance in GCM and which were implicated from some NVProf results, and overall poor utilization of GPU memory resources by the GCM code may still be impeding the performance in addition to potentially sub-optimal instruction assembly due to the compiler's heuristics.

Additional work is needed to resolve this problem with the PGFEM. Within the TTL, certain high-use tensors of known rank and dimension common to physical systems have been implemented as a partial template specialization to encourage the same compiler heuristic pathing used in parsing the for-loops that performed better than the recursive template expression design used originally. Additional likely candidates would be re-examining and refactoring memory loading from the host to the GPU to utilize more of the GPU's resources and making sure they are in agreement with the for-loop implementation.

Works Cited

[1] A. Brodtkorb, T. R. Hagen and M. L. Saestra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *Journal of Parallel and Distributed Computing,* vol. 73, no. 1, pp. 4-13, 2013.

[2] J. H. Park and W. W. Ro, "Accelerating forwarding computation of artificial neural network using CUDA," *International Conference on Electronics, Information, and Communications (ICEIC),* 2016.

[3] The Guardian, "Bitcoin's energy usage is huge – we can't afford to ignore it," 17 Jan 2018. [Online]. Available: https://www.theguardian.com/technology/2018/jan/17/bitcoin-electricity-usage-huge-climate-cryptocurrency.

[4] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Austin, TX, USA, 2008.

[5] G. Peterson, D. Yang, J. Sun and J. K. Lee, "Performance Comparison of Cholesky Decomposition on GPUs and FPGAs," in *Symposium on Application Accelerators in High Performance Computing*, 2010.

[6] E. Bertschinger, Introduction to Tensor Calculus for General Relativity, Cambridge, Massachusetts: Massachusetts Institute of Technology, 1999.

[7] T. Sochi, Introduction to Tensor Calculus, London: University College London, 2016.

[8] S. Suram, D. McCorkle and K. Bryden, "Proper Orthogonal Decomposition-Based Reduced Order Model of a Hydraulic Mixing Nozzle," in *Mechanical Engineering Conference Presentations, Papers, and Proceedings. 55*, 2008.

[9] E. Elmroth and F. Gustavson, "Applying recursion to serial and parallel QR factorization," *Ibm Journal of Research and Development,* vol. 44, no. 4, pp. 605-624, 2000.

[10] S. Samuel, "Maintaining High Performance in QR Factorization While Scaling Both Problem Size and Parallelism," University of Texas, San Antonio, 2011.

[11] Aravindh Krishnamoorthy; Deepak Menon, "Matrix Inversion Using Cholesky Decomposition," ST-Ericsson India Private Limited, Bangalore.

[12] L. Trefethen and R. Schreiber, "Average Case Stability of Gaussian Elimination," *SIAM Journal on Matrix Analysis and Applications,* 1990.

[13] Greg Fasshauer, "Chapter 7: Gaussian Elimination and LU Factorization," Illinois Institute of Technology, [Online]. Available: http://www.math.iit.edu/~fass/477577_Chapter_7.pdf.

[14] G. Poole and L. Neal, "The Rook's Pivoting Strategy," *Journal of Computational and Applied Mathematics,* vol. 123, no. 1-2, pp. 353-369, 2000.

[15] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," March 2005. [Online]. Available: http://courses.cecs.anu.edu.au/courses/COMP4300/lectures15/Free_Lunch.pdf.

[16] D. Bhandarkar and D. Clark, "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization," *ACM SIGPLAN Notices*, vol. 26, no. 4, pp. 310-319, 1991.

[17] E. Blem, J. Menon, T. Vijayaraghavan and K. Sankaralingam, "ISA Wars: Understanding the Relevance of ISA being RISC or CISC to Performance, Power, and Energy on Modern Architectures," *ACM Transactions on Computer Systems*, vol. 33, no. 1, 2015.

[18] S. Eyerman, J. Smith and L. Eeckhout, "Characterizing the Branch Misprediction Penalty," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, 2006.

[19] Intel Corporation: Chit-Kwan Lin, Stephen J. Tarsa, "Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions," Intel, Santa Clara, 2014.

[20] S. Mittal, "A Survey on Evaluating and Optimizing Performance of Intel Xeon Phi," Indian Institute of Hyderabad, 2019.

[21] T. Warburton, "An Intro to GPU Architecture and Programming Models I Tim Warburton, Virginia Tech," Virginia Tech, ANL Training, 25 September 2017. [Online]. Available: https://www.youtube.com/watch?v=lGmPy8xpT4E.

[22] N. R. Mahapatra and B. Venkatrao, "The Processor-Memory Bottleneck: Problems and Solutions," *XRDS: Crossroads, The ACM Magazine for Students - Computer architecture*, 1999.

[23] X. Mei, K. Zhao, C. Liu and X. Chu, "Benchmarking the Memory Hierarchy of Modern GPU," in *Network and Parallel Computing: 11th IFIP WG* , Ilan, Taiwan, 2014.

[24] J. Handy, "Objective-Analysis.com," June 2018. [Online]. Available: https://objective-analysis.com/uploads/2018-06-18%20Objective%20Anlaysis%20White%20Paper%20-%20New%20Memories%20for%20Efficient%20Computing.pdf.

[25] Peripheral Component Interconnect Special Interest Group, "PCI Express 3.0 Frequently Asked Questions," 10 July 2012. [Online]. Available: https://web.archive.org/web/20131019114456/http://www.pcisig.com/news_room/faqs/pcie3.0_faq/PCI-SIG_PCIe_3_0_FAQ_Final_07102012.pdf.

[26] X. Li and P.-c. Shih, "An Early Performance Comparison of CUDA and OpenACC," in *MATEC Web of Conferences*, 2018.

[27] "What is the difference between AMD's Stream Processor and NVIDIA's GeForce 8800?," November 2006. [Online]. Available: https://insidehpc.com/2006/11/what-is-the-difference-between-amds-stream-processor-and-nvidias-geforce-8800-or-is-crays-strategy-the-right-one-after-all/.

[28] nVidia Corporation, "CUDA Developer Toolkit Documentation: Hardware Implementation," November 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation.

[29] nVidia Corporation, "NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110/210," 2014.

[30] nVidia Corporation, "nVidia GEForce GTX 680 White Paper," 2012.

[31] nVidia Corporation: Peter N. Glaskowsy, "NVIDIA’s Fermi: The First Complete GPU Computing Architecture," 2009.

[32] N. Wilt, "Blocks, Threads, Warps, and Lanes," in *The CUDA Handbook*, 2013, pp. 211-215.

[33] B. C. S. S. M. nVidia Corporation: John Erik Lindholm, "Across-thread out-of-order instruction dispatch in a multithreaded microprocessor". US Patent US7676657B2, 2003.

[34] X. Mei and X. Chu, "Dissecting GPU Memory Hierarchy Through Microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems* , vol. 28, no. 1, pp. 72-86, 2017.

[35] nVidia Corporatio: Mark Harris. Stanford: Ian uck, "GPU Flow-Control Idioms," in *GPU Gems*, Taunto, Massachusetts, 2005.

[36] nVidia Corporation: Nikolay Sakharnykh, "nVidida Developer Blog: Maximizing Unified Memory Performance in CUDA," 19 November 2017. [Online]. Available: https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/.

[37] R. Landaverde, T. Zhang, A. K. Coskun and M. Herbordt, "An investigation of Unified Memory Access performance in CUDA," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2014.

[38] Intel, "Intel® Xeon® Processor E5-2620 v2," [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/75789/intel-xeon-processor-e5-2620-v2-15m-cache-2-10-ghz.html.

[39] Department of Energy, Lawrence Livermore National Laboratory, "Pascal," [Online]. Available: https://hpc.llnl.gov/hardware/platforms/pascal.

[40] nVidia Corporation, "NVIDIA® Tesla P100 GPU Accelerator," [Online]. Available: https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf.

[41] Intel, "Product Specifications, Intel® Xeon® Processor E5-2695 v4," [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/91316/intel-xeon-processor-e5-2695-v4-45m-cache-2-10-ghz.html.

[42] nVidia Corporation, "GeForce GTX Titan Specifications," [Online]. Available: https://web.archive.org/web/20151205101408/http:/www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications.

[43] Indiana University, "Juliet Systems," [Online]. Available: https://www.dsc.soic.indiana.edu/sites/default/files/FUTURESYSTEMS.pdf.

[44] nVidia Corporation, "nVidia Tesla V100 GPU Accelerator," [Online]. Available: https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf.

[45] Google Github, "Google Test," [Online]. Available: https://github.com/google/googletest.

[46] nVidia Corporation, "Unified Memory in CUDA 6," 6 November 2013. [Online]. Available: https://devblogs.nvidia.com/unified-memory-in-cuda-6/.

[47] H. Sutter, "We have an international standard: C++0x is unanimously approved," 12 08 2011. [Online]. Available: https://herbsutter.com/2011/08/12/we-have-an-international-standard-c0x-is-unanimously-approved/.

[48] nVidia Corporation: Mark Harris, "C++11 in CUDA: Variadic Templates," 26 March 2015. [Online]. Available: https://devblogs.nvidia.com/cplusplus-11-in-cuda-variadic-templates/.

[49] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammarling, J. Demmel, C. Bischof and D. Sorensen, "LAPACK: a portable linear algebra library for high-performance computers," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, New York, 1990.

[50] L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaurman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington and R.C.Whaley, "An Updated Set of Basic Linear Algebra Subprograms (BLAS)," *ACM Transactions on Mathematical Software,* 2001.

[51] nVidia Corporation, "The CUDA Toolkit Documentation: cuBLAS," November 2019. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html.

[52] nVidia Corporation, "cuBLAS Library," 2012. [Online]. Available: https://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf.

[53] A. Buluc, J. R. Gilbert and C. Budak, "Solving path problems on the GPU," *Parallel Computing,* vol. 36, no. 5-6, pp. 241-253, 2010.

[54] V. Lee, C. Kim, J. Chhugani, M. Deischer, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singal and P. Dubey, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," in *Association for Computing Machinery*, Saint-Malo, France, 2010.

[55] nVidia Corporation: Mark Harris, "nVidida Developer Blog: How to Access Global Memory Efficiently in CUDA C/C++ Kernels," 7 January 2013. [Online]. Available: https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/.

[56] P. Springer and P. Bientinesi, "Design of a High-Performance GEMM-like Tensor–Tensor Multiplication," *ACM Transactions on Mathematical Software ,* vol. 44, no. 3, 2018.

[57] nVidia Corporation: Luke Durant, Olivier Giroux, Mark Harris, Nick Stam , "nVidida Developer Blog - Inside Volta: The World's Most Advanced Data Center GPU," 10 May 2017. [Online]. Available: https://devblogs.nvidia.com/inside-volta/.

[58] nVidia Corporation, "nVidia Developer Toolkit Documentation," [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#branching-and-divergence.

[59] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith and J. Manferdelli, "High performance discrete Fourier Transforms on Graphics Processors," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Piscataway, 2008.

[60] M. Silberstein, A. Schuster, D. Geiger, A. Patney and J. Owens, "Efiicient Computation of Sum-Products on GPUs through Software-Managed Cache," in *Proceedings of the 22nd ACM Internation Conference on Supercomputing*, 2008.

[61] J. Tolke and M. Krafczyk, "TeraFLOP Computing on a Destop PC with GPUs for 3D CFD," *International Journal of Computational Fluid Dynamics,* vol. 22, pp. 443-456, 2008.

[62] F. Vazques, E. Garzon, J. Martinez and J. Fernandez, "The SParse Matrix Vector Product on GPUs," University of Almeria, 2009.

[63] Z. Yang, Y. Zhu and Y. Pu, "Parallel Image Processing Based on CUDA," in *International Conference on Computer Science and Software Engineering*, 2008.

[64] P. Wiemann, S. Wenger and M. Magnor, "CUDA Expression Templates," Braunschweig Technical University, Braunschweig, 2011.

[65] M. Hill and M. Marty, "Amdahl's Law in the Multicore Era," *Computer,* vol. 41, no. 7, pp. 33-38, 2008.

[66] Eliot Eshelman, "NVIDIA Tesla P100 Price Analysis," Microway , 1 August 2016. [Online]. Available: https://www.microway.com/hpc-tech-tips/nvidia-tesla-p100-price-analysis/.

[67] nVidida Corporation: Thibaut Lutz, Yang Chen, Vinod Grover, "CUDA Compiler Verification," [Online]. Available: https://materials.dagstuhl.de/files/17/17502/17502.ThibautLutz.Slides.pdf.

[68] Intel, "General compatibility of the Intel C++ Compiler for Windows," 2009. [Online]. Available: https://web.archive.org/web/20120207092727/http://software.intel.com/en-us/articles/intel-c-compiler-for-windows-general-compatibility-with-other-products/#17.

[69] R. Crovella, "NVCC bug when trying simple Template Metaprogramming," 2017. [Online]. Available: https://devtalk.nvidia.com/default/topic/1018200/nvcc-bug-when-trying-simple-template-metaprogramming/.

[70] Open MPI, "FAQ: Compiling MPI Applications," 20 May 2019. [Online]. Available: https://www.open-mpi.org/faq/?category=mpi-apps.

[71] nVidia Corporation, "The CUDA Toolkit Documentation: the CUDA Compilation Trajectory," [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#cuda-compilation-trajectory.

Glossary

ALU: Arithmetic Logic Unit, performs the mathematical and logic calculations to alter and examine data. The workhorse of the CPU and GPU, overwhelmingly more present on GPUs than CPUs.

AVX: Advanced Vector Extensions. Old Intel ISA for vectorization. See vectorization.

Cache: a form of SRAM intermediate between main memory at the processor, built on board the chip. Essential to hierarchical memory to overcome the memory wall.

CISC: Complex Instruction Set Computer. An ISA that receives instructions in larger forms that generally resemble they actual operation they perform such as “add”. In modern x86 processors, wihle technically CISC, under the hood they implement “micro-ops” that make them functionally RISC.

Clock generator: the circuit generating the pulse waves that coordinate the circuits of the CPU. Measured in Hz.

Clock time: the duration of time needed for an instruction, routine, or program to complete.

Execution context: the set of registers - instruction, stack, and program counter, and set of operands – which define a phase in a program. Not quite all of what defines a thread, but critical to the execution of one, any time a thread moves in or out of context, the various registers defining the execution context must be written to some lower hierarchical memory and reloaded when a thread resumes execution.

CPU: Central Processing Unit: The collection of integrated circuits that manipulate and process data. Composed of ALUs, Registers, Cache Memory, and Control Units.

Co-variance: Variance s.t. a vector scales with basis changes. A co-vector/dual/one-form. Also a row vector.

Co-vector: cf. dual vector.

Compile time: the point in program construction when a higher level language is translated into machine or assembly instructions.

Context: the set of memory states (program and stack counter, various registers) minimally necessary to allow a process or thread to be interrupted and restored. Multi-threading often uses context switching to conceal memory latency, using quick reads from the cache to reload an inactive process while storing the context of an active process during al long retrieval from memory.

Contra-variance: Variance s.t. a vector scales against basis changes. A true vector. A column vector.

Control Unit: the integrated circuit of a CPU handling control flow. Manages branch prediction, OoOE, instruction reordering, etc.

Cubin: CUDA binary. Developed by the NVCC.

Device: An environment driven by the GPU, used synonymously with GPU.

DRAM: dynamic random access memory. Made from transistor gated capacitors, slower than SRAM as it needs to be refreshed with power as the capacitor slowly discharges. Cheaper than SRAM as it needs fewer transistors.

Dual vector: a map transforming a vector in a scalar.

Dual basis: the basis for dual vectors. Constructed from the Levi-Civitia symbol.

FLOP: Floating point operation.

GCC: Gnu Compiler Compilation, a Linux compiler for C/C++.

GPU: Graphics processing unit. A massively parallel version of a CPU capable of orders of magnitude more FLOPs operating on a SIMT parallelization.

Hierarchical memory: a means of overcoming the memory wall by creating a tiered memory system of different speeds related to presence on or off the chip, smoothing processing time for data that needs to be retrieved from slow memory banks.

HPC: High Performance Computation. Computation focused on total optimization rather than usability. In modern computing environments, generally only necessary within scientific computation.

Host: A CPU environment, used synonymously with CPU and most runtime environments.

Index: a variable linking different components of a tensor, the shared index over which an array traverses. In tensor expressions or tensor objects, free indices are unique; repeated indices are bound.

Inner Product: a tensor operation across bound indices which produces a scalar by the sum product. Cf. Contraction.

ILP: instruction level parallelism. A measure of the componentization of instructions in a pipeline.

Instruction Pipeline: A technique for improving processing time by subdividing instructions. This builds an assembly line of component instructions, thereby allowing multiple discrete sub-actions to be performed simultaneously in parallel to complete the instruction faster, reducing clock time. Virtually all modern CPUs are pipelined.

ISA: instruction set architecture. The set of instructions that the CPU can invoke to process data. Example instructions might be “load”, or “add”. Often categorized as RISC or CISC to reflect how componential the instructions are and how they are pipelined.

IR: instruction register.

Levi-Civitia symbol: the basis for the dual space. Enables

Memory Wall: the bottleneck in memory due to bussing. Fetching data from main memory is orders of magnitude slower than the time needed to process the data due to a combination of DRAM and the latency of bussing. This problem has never truly been resolved, only sidestepped via on-chip caching.

Metric Tensor: a (0,2) tensor whose principal utility is as a map for vectors into dual vectors and inversely, duals into vectors.

MMU: Memory management unit.

NVCC: nVidia’s proprietary compiler. Generates ELF code and GPU binary for host-device interoperable binary. Requires a host-side compiler to generate the host code.

One-form: cf. dual vector.

Outer Product: a tensor operation across free indices which produces a larger tensor by scaling each index by the other free indices.

Permutation Tensor: cf. Levi-Civita Symbol.

Register: the fastest form of hierarchical memory, composed of SRAM. They are the source of operands, instructions, and address values.

RISC: Reduced Instruction Set Computer. A CPU whose ISA is small, simple tasks. Improves ILP by subdividing tasks, but instructions do not generally resemble the operations we intend such as “add”.

Run time: All events that occur in a program after compile time.

Scalar Processor: a processor which can execute at most one instruction per clock cycle.

SRAM: static random access memory. Made from transistor gated capacitors, faster than DRAM which needn’t be cycled to refresh the memory state, but more expensive.

SSE: Streaming SIMD Extension, Intel ISA for vectorization.

Superscalar Processor: a processor which can execute more than one instruction per clock cycle by dispatching to different execution units. Most modern CPUs are superscalar.

Tensor: a mathematical object that defines operations between other tensor objects and within tensors via index regular expressions. Several context dependent definitions, cf. §2.2.1

Tensor contraction: a tensor inner product where the bound indices occur within the same tensor object.

Variance: the scaling of a vector or co-vector with or against a change of basis. Reflects the difference of objects kinds between vectors and co-vectors.

Vector: An object with magnitude and direction. Also an array of scalars. Exists within a basis.

Vectorization: utilizing a separate ISA, in an Intel x86 context AVX or SSE, to perform concurrent instructions on chunks of data to create performance akin to a vector processor.

Vector Processor: A processor which applies a single instruction across multiple data elements in an array.

Bibliography

These works were read by author as part of exploring this thesis but are not explicitly cited in the work.

Crovella, Robert. “NVCC Preprocessing.” *Devtalk.nvidia.com*, 2018, devtalk.nvidia.com/default/topic/1039311/nvcc-preprocessing/.

Bhattacharjee, Abhishek, et al. *Architectural and Operating System Support for Virtual Memory*. Morgan & Claypool Publishers, 2017.

Computer Architecture & The Machine Cycle: The Processor. Chemeketa CC CS Dept., computerscience.chemeketa.edu/cs160Reader/ComputerArchitecture/Processor2.html.

“Control Unit, ALU, and Memory.” Oxford Robotics Institute. www.robots.ox.ac.uk/~dwm/Courses/2CO_2014/2CO-N3.pdf.

“CUDA Binary Utilities.” *CUDA Developer Toolkit Documentation*, by NVidia Corporation, docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

“CUDA Binary Utilities.” *NVIDIA Developer Documentation*, docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

“CUDA Profiler User's Guide.” *CUDA Developer Toolkit Documentation*, docs.nvidia.com/cuda/profiler-users-guide/index.html.

“Development of the Plane Stress and Plane Strain Stiffness Equations.” *A First Course in the Finite Element Method*, by Daryl L. Logan, Cengage Learning, 2012, pp. 329–376.

Gao, Hao. “Basic Concepts in GPU Computing.” *Medium*, Medium, 10 Oct. 2017, medium.com/@smallfishbigsea/basic-concepts-in-gpu-computing-3388710e9239.

Haase, Sven-Hendrik. “Alignment in C.” 2014, hps.vi4io.org/_media/teaching/wintersemester_2013_2014/epc-14-haase-svenhendrik-alignmentinc-paper.pdf.

Hummel, Rolf E. *Electronic Properties of Materials*. Springer, 2014.

Jacob, Bruce, et al. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers, 2010.

Jordan, Vincent. “Development Environment for Multi Platform CUDA Software.” *Development Environment for Multi Platform CUDA Software*, vjordan.info/thesis/nvidia_gpu_archi/devel_env.xhtml.

L14: The Memory Hierarchy, MIT Department of Electrical Engineering and Computer Science, 2017, computationstructures.org/lectures/caches/caches.html.

Lay, David C. *Linear Algebra and Its Applications*. Addison-Wesley, 2012.

Marr, Deborah T., et al. “Hyper-Threading Technology Architecture and Microarchitecture.” *Intel Technology Journal*, vol. 6, no. 1, 14 Feb. 2002, pp. 1–12.

Mutlu, Onur. “Computer Architecture: SIMD/Vector/GPU.” Carnegie Mellon Digital Archives. www.archive.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=seth-740-fall13-module5.1-simd-vector-gpu.pdf.

Nilsson, James William, and Susan A. Riedel. *Electric Circuits*. Pearson Education Limited, 2015.

nVidia Corporation. “Achieved Flops.” *Achieved FLOPs*, 2015, docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedflops.htm.

nVidia Corporation. “CUDA Binary Utilities.” *NVIDIA Developer Documentation*, 2019, docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

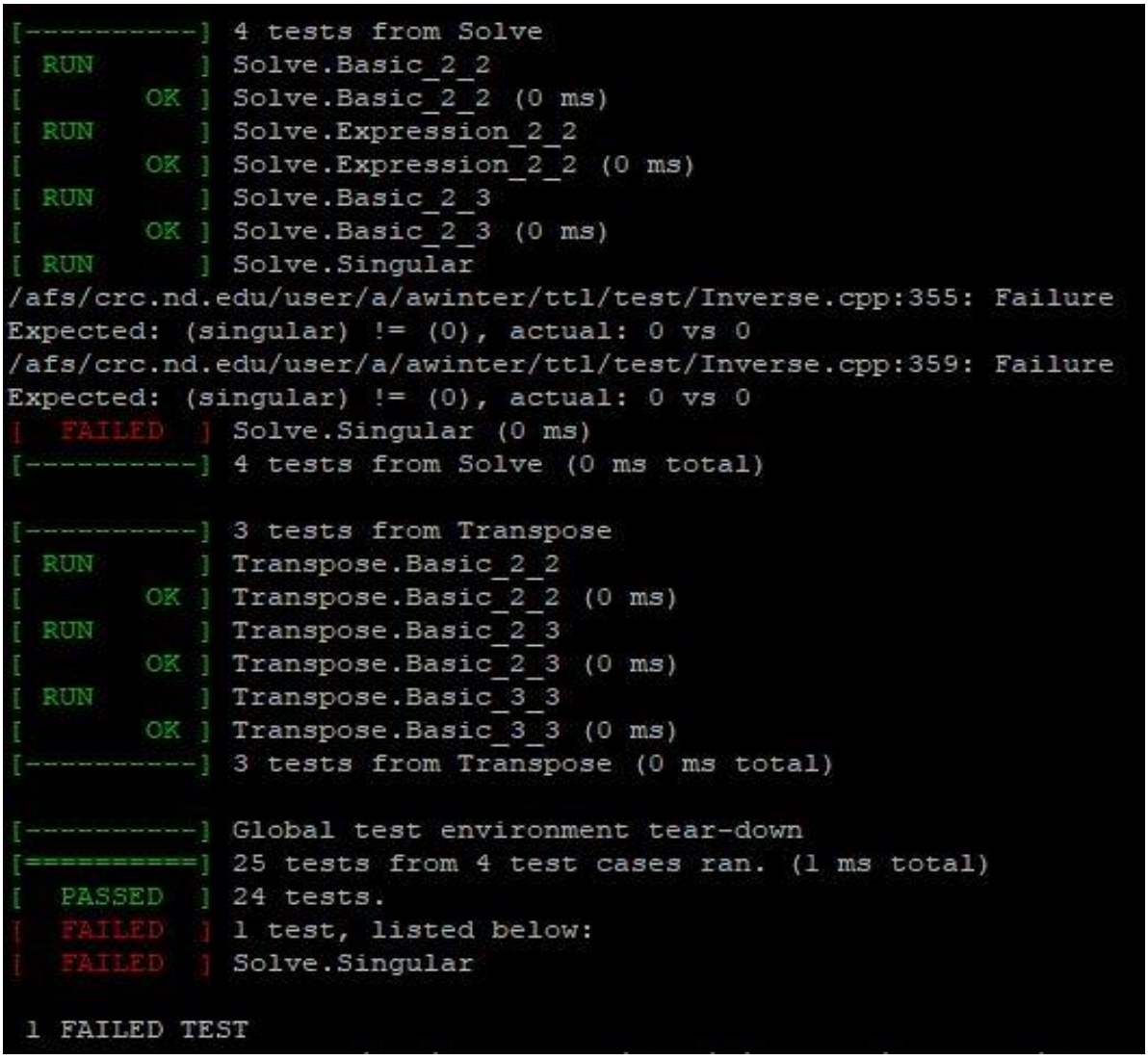
NVidia. *PTX: Parallel Thread Execution ISA Version 2.1*, developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ptx_isa_2.1.pdf.

Prabhu, Gurpur M. *Computer Architecture Supplemental Material*. Iowa State, web.cs.iastate.edu/~prabhu/Tutorial/title.html.

“What Is AVX and SSE?” *CodinGame*, www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx.

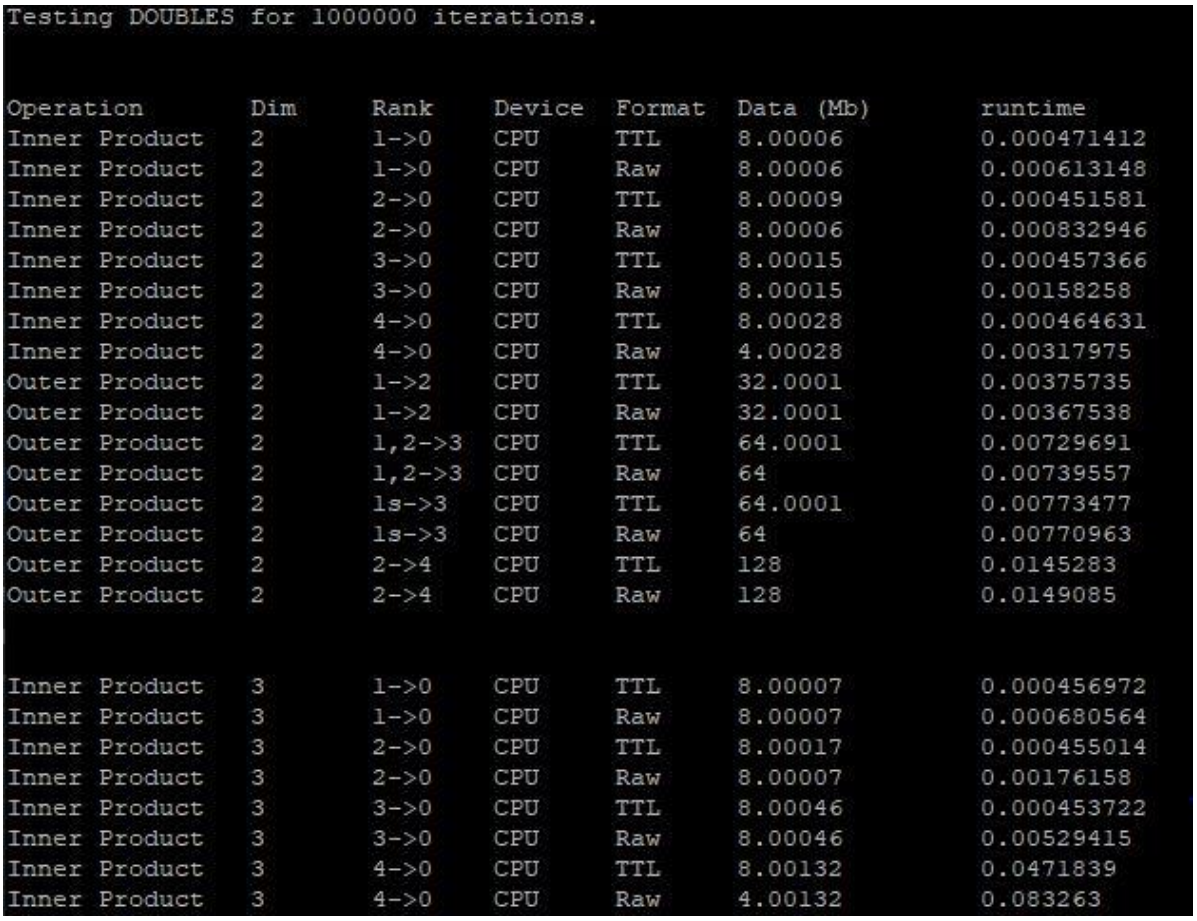
Appendix A. Additional Figures

Enlarged Figure 21. Example GTest results



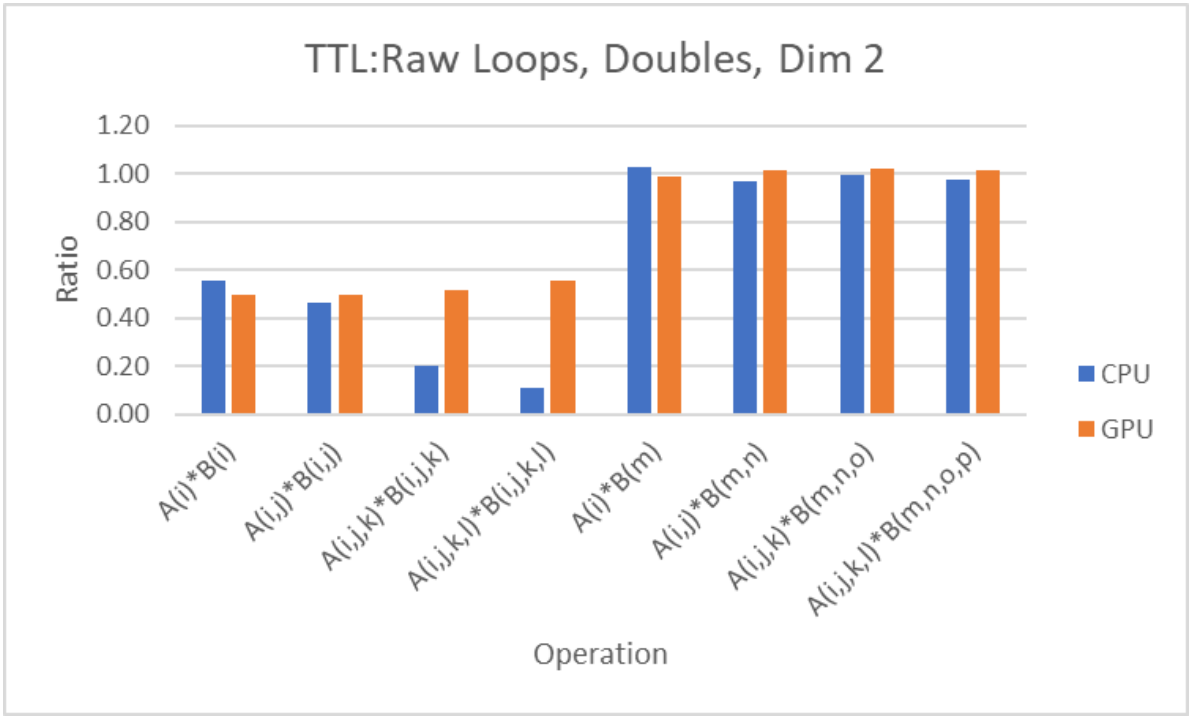
This actually revealed a problem in how the TTL was forwarding data, as the GTest was unable to correctly verify the result being generated (0) was equal to the expected result (0).

Enlarged Figure 23. Example Output from the Microbenchmarking Test

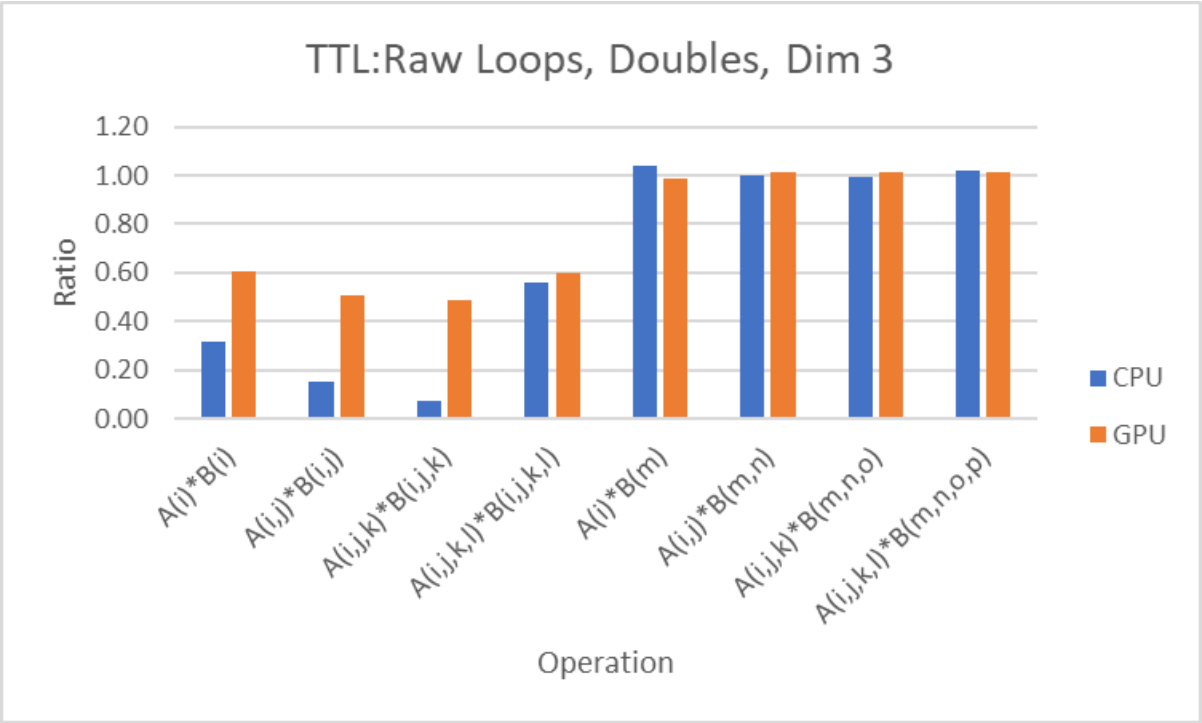


```
Testing DOUBLES for 1000000 iterations.


Operation       Dim     Rank    Device  Format  Data (Mb)       runtime
Inner Product   2       1->0    CPU     TTL     8.00006         0.000471412
Inner Product   2       1->0    CPU     Raw     8.00006         0.000613148
Inner Product   2       2->0    CPU     TTL     8.00009         0.000451581
Inner Product   2       2->0    CPU     Raw     8.00006         0.000832946
Inner Product   2       3->0    CPU     TTL     8.00015         0.000457366
Inner Product   2       3->0    CPU     Raw     8.00015         0.00158258
Inner Product   2       4->0    CPU     TTL     8.00028         0.000464631
Inner Product   2       4->0    CPU     Raw     4.00028         0.00317975
Outer Product   2       1->2    CPU     TTL     32.0001         0.00375735
Outer Product   2       1->2    CPU     Raw     32.0001         0.00367538
Outer Product   2       1,2->3  CPU     TTL     64.0001         0.00729691
Outer Product   2       1,2->3  CPU     Raw     64              0.00739557
Outer Product   2       ls->3   CPU     TTL     64.0001         0.00773477
Outer Product   2       ls->3   CPU     Raw     64              0.00770963
Outer Product   2       2->4    CPU     TTL     128             0.0145283
Outer Product   2       2->4    CPU     Raw     128             0.0149085

Inner Product   3       1->0    CPU     TTL     8.00007         0.000456972
Inner Product   3       1->0    CPU     Raw     8.00007         0.000680564
Inner Product   3       2->0    CPU     TTL     8.00017         0.000455014
Inner Product   3       2->0    CPU     Raw     8.00007         0.00176158
Inner Product   3       3->0    CPU     TTL     8.00046         0.000453722
Inner Product   3       3->0    CPU     Raw     8.00046         0.00529415
Inner Product   3       4->0    CPU     TTL     8.00132         0.0471839
Inner Product   3       4->0    CPU     Raw     4.00132         0.083263
```

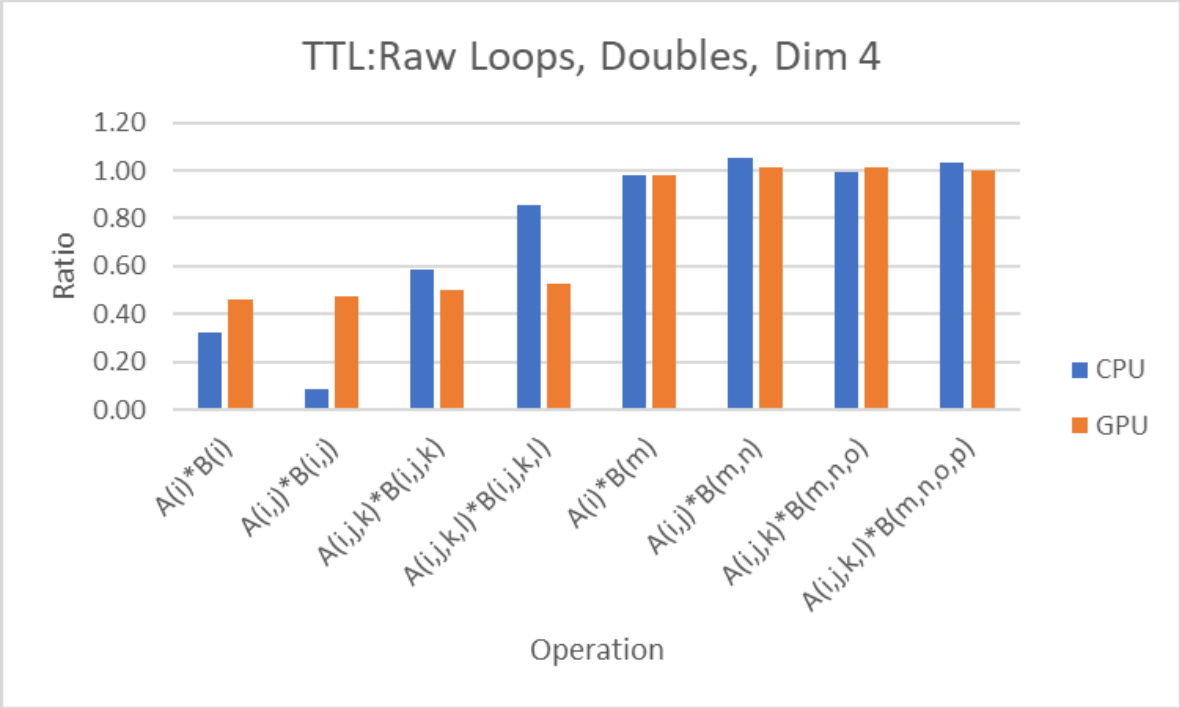
Figure 45. Microbenchmarking, Double, Dim 2



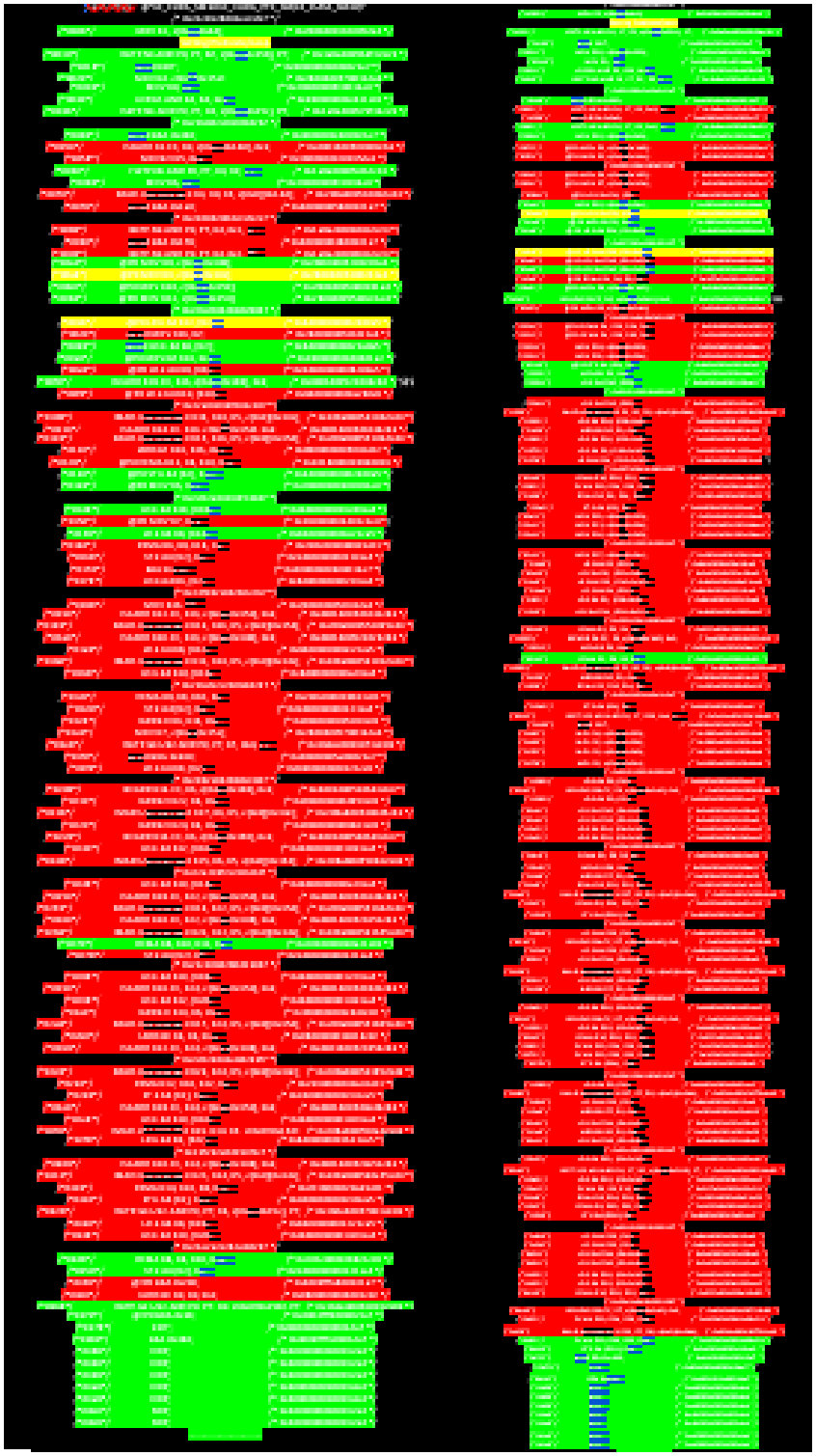
Enlarged Figure 31. Microbenchmarking, Double, Dim 3



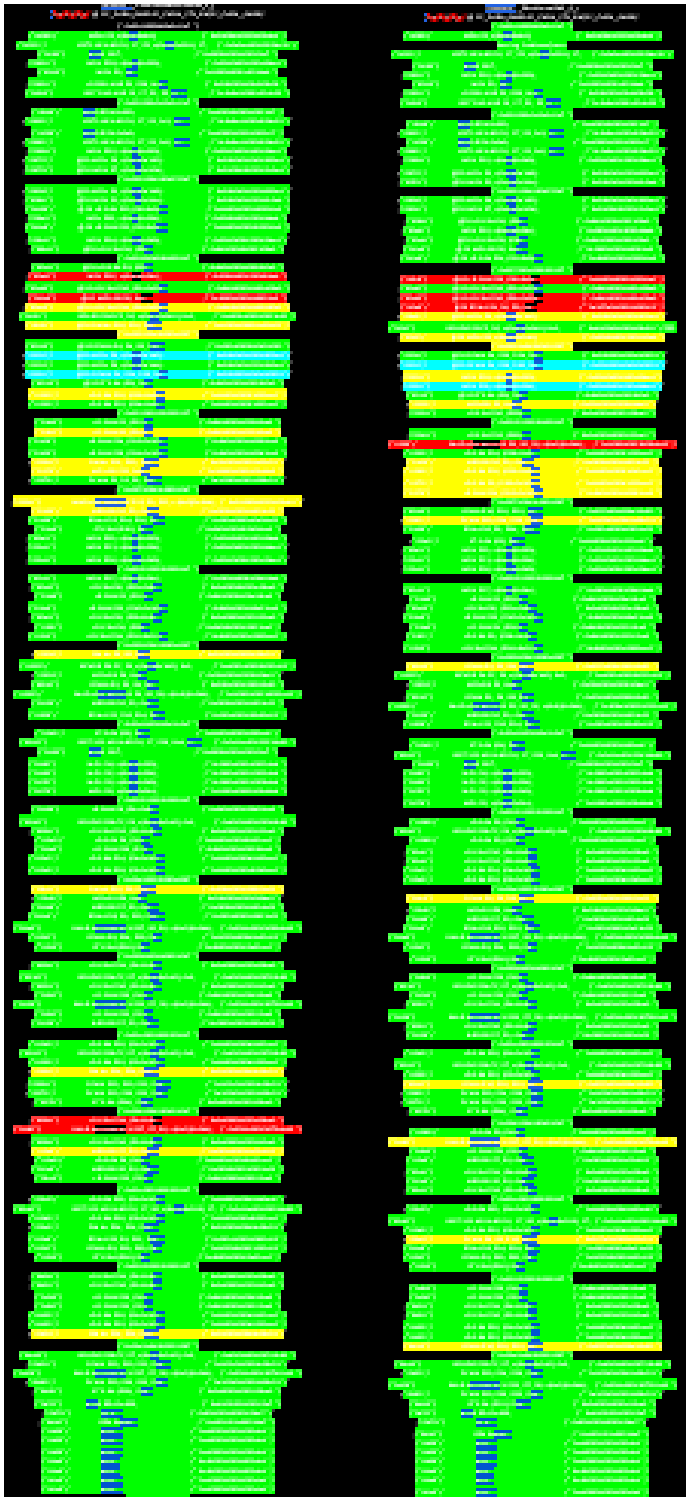
Enlarged Figure 32. Microbenchmarking, Double, Dim 4



Enlarged Figure 35. Comparison of the PTX of the inner product of two rank 1 tensors. TTL implementation on the right, for loop implementation on the left.



Enlarged Figure 39. Comparison of the PTX of the inner product of two rank 1 tensors. TTL implementation on the right, manually unrolled loop implementation on the left.



Enlarged Figure 41. X86 raw loop implementation

Enlarged Figure 42. x86 of the TTL implementation

Enlarged Figure 43. x86 of the Manually Unrolled Loop

Enlarged Figure 44. X86 of the TTL in Contrast to the Manually Unrolled Loop