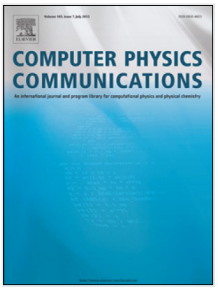
Accepted Manuscript



A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics

Roman Poya, Antonio J. Gil, Rogelio Ortigosa

PII: S0010-4655(17)30068-1
DOI: http://dx.doi.org/10.1016/j.cpc.2017.02.016
Reference: COMPHY 6164

To appear in: *Computer Physics Communications*

Received date: 1 November 2016
Revised date: 10 January 2017
Accepted date: 15 February 2017

Please cite this article as: R. Poya, A.J. Gil, R. Ortigosa, A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics, *Computer Physics Communications* (2017), http://dx.doi.org/10.1016/j.cpc.2017.02.016

A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics

Roman Poya [a,b 1], Antonio J. Gil [a2], Rogelio Ortigosa [a]

[a] *Zienkiewicz Centre for Computational Engineering, College of Engineering, Swansea University, Bay Campus, SA1 8EN, United Kingdom*

[b] *Institute for Computational Mechanics, Technische Universität München, Boltzmannstrasse 15, D-85748 Garching b. München, Germany*

Abstract

The paper presents aspects of implementation of a new high performance tensor contraction framework for the numerical analysis of coupled and multi-physics problems on streaming architectures. In addition to explicit SIMD instructions and smart expression templates, the framework introduces domain specific constructs for the tensor cross product and its associated algebra recently rediscovered by Bonet et. al. [1, 2] in the context of solid mechanics. The two key ingredients of the presented expression template engine are as follows. First, the capability to mathematically transform complex chains of operations to simpler equivalent expressions, while potentially avoiding routes with higher levels of computational complexity and, second, to perform a compile time depth-first search to find the optimal contraction indices of a large tensor network in order to minimise the number of floating point operations. For optimisations of tensor contraction such as loop transformation, loop fusion and data locality optimisations, the framework relies heavily on compile time technologies rather than source-to-source translation or JIT techniques. Every aspect of the framework is examined through relevant performance benchmarks, including the impact of data parallelism on the performance of isomorphic and nonisomorphic tensor products, the FLOP and memory I/O optimality in the evaluation of tensor networks, the compilation cost and memory footprint of the framework and the performance of tensor cross product kernels. The framework is then applied to finite element analysis of coupled electromechanical problems to assess the speed-ups achieved in kernel-based numerical integration of complex electroelastic energy functionals. In this context, domain-aware expression templates are shown to provide a significant speed-up over the classical low-level style programming techniques.

Keywords: Tensor contraction, data parallelism, domain-aware expression templates, nonlinear coupled electromechanics

1. Introduction

In the field of multilinear algebra, tensor contraction refers to operations involving natural pairing of tensors in finite dimensional vector spaces. Such operations are archetypal of quantum

[1]Corresponding Author: r.poya@swansea.ac.uk
[2]Corresponding Author: a.j.gil@swansea.ac.uk

and classical mechanics. It is well known that, efficient implementation of tensor contraction of tensor networks involving sum of multiple indices is a NP-hard problem [3–8]. Current adopted methodologies typically rely either on graph optimisation techniques to find optimal contraction indices, such as depth-first [3, 6], breadth-first [9, 10] and cheapest-first constructive approaches [7, 8] or dynamic programming with memoisation [7], all well established in the field of quantum many-body physics and quantum chemistry. On the other hand, in the field of mechanics of continua, tensor contractions arise naturally, in the variational forms of the governing equations [11–13] and hence in their consistent linearisation. Finite element discretisation of these forms, then heavily rely on tensorial operations between the gradient of the chosen functional spaces and the work conjugates and Hessian of the internal energies [11, 14, 15].

A myriad of strategies can be applied to optimise tensor contractions that emerge from the discretisation of an underlying variational formulation. A noteworthy approach which is typically utilised by domain specific languages (DSLs) designed for automated finite element code generation, is the exploitation of the structure and topology of the tensors either by a careful study of the bilinear operator and the chosen functional spaces or by performing similar graph optimisation techniques in order to minimise the number of floating point operations [11, 16]. Such optimisation techniques have been applied successfuly for instance in [16–21] for various discretisation schemes such as continuous Galerkin, discontinuous Galerkin and various functional spaces such as H^1, $H(\text{div})$ and $H(\text{curl})$ spaces, for elliptic as well as hyperbolic PDEs. As an automated finite element code generator, these approaches typically abstract away the numerical implementation from the mathematical formulation and have the potential to optimise the entire finite element assembly procedure. The framework described in this manuscript is not desgined to be an automated code generator for variational forms (form compiler). Instead, it is rather designed to serve as a generic tensor algebra library that facilitates an explicit mechanism for declaring tensorial operations, while potentially employing analogous optimisation techniques, where applicable. As a result, the implementation specificities of a given problem is left to the developer and not automated. However, the framework provides a high level API, to bring forth low level optimisations at the disposal of the user (say for explicit finite element programming) and as a result could be used as a standalone frontend software or an optimising backend for a form compiler.

Akin to the current framework are the specifically tailored numerical tensor algebra frameworks developed in C++. The foundation for implementation of a high performance tensor algebra framework was laid by the works of Veldhuizen et. al. [22, 23] and Landry [13] on `Blitz++` and `FTensor` libraries, respectively. Other notable examples of tensor algebra frameworks include `nDarray` [24], `LTensor` [25], `libtensor` [26] and `Eigen`'s third party tensor package [27]. Barring `Eigen` which is based on C++11 *variadic templates*, all of the aforementioned frameworks are C++03 compliant, which implies they are not truly multi-dimenional tensor libraries. In other words, these frameworks have support for tensors with up to a few spatial dimensions. Furthermore, none of the aforementioned frameworks implement domain-aware expression templates and optimisation algorithms, since they are designed as generic numerical tensor algebra libraries. On the other hand, such optimisation techniques are more suited to domain specific tensor contraction frameworks, examples of which include for instance, the `Tensor Contraction Engine` [28–30], `TiledArray` [31] and `Cyclops Tensor Framework` [32, 33], which are designed for distributed and thread-parallel tensor operations for quantam mechanical computations. A few noteworthy differences between the current framework and

the aformentioned tensor contraction libraries, specifically the Tensor Contraction Engine are as follows. a) the utilisation of compile time technologies for ahead-of-time evaluations using C++11 metaprogramming rather than source-to-source translation or JIT techniques, b) implementation of low-level optimisation techniques such as data parallelism and loop transformation (in C++) instead of relying on a low-level language optimising compiler (Fortran) c) the focus on small tensor networks rather than big, out-of-core tensors and finally d) the focus on continuum multi-physics simulations rather than quantum chemistry applications.

The fundamental design principle that all tensor frameworks rely on is the concept of expression templates in C++ [13, 34, 35], which provides a powerful means for lazy or on-demand evaluation of arbitrary chained operators as well as delaying the evaluation of certain tensor algebraic operations. In contrast to the classical operator overloading technique, expression templates completely avoid the need for creation of intermediate temporaries. In [36, 37] novel expression templates are presented which go beyond the level 1 BLAS overloads whilst exploring other optimisation opportunities such as loop-tiling and data parallelism.

Recently, data parallel and stream computing have become a requisite for large scale simulations of scientific problems. Recent generations of CPUs and GPUs, require data-parallel codes for full efficiency. Data parallelism essentially implies that the same sequence of operations should be applied to mutliple data sets synchronously. This reduces the need for instruction scheduling in favour of more arithmetic and logic units [37, 38]. On CPU architectures data parallelism is implemented via SIMD (Single Instruction Multiple Data) registers and instructions, wherein a single SIMD register can store multiple values and a single SIMD instruction can execute multiple operations on those values [39–41]. The FMA (Fused-Multiply-Add) instruction set is an archetypal example of data parallelism. There are typically two approaches to explicit vectorisation namely, the use of frameworks which are built as an extension to the language such as OpenMP [42, 43], Cilk Plus [44] and OpenCL [45] and the use of SIMD vector types [27, 38, 46]. In this work, an explicit vectorisation approach using vector types is adopted.

In this manuscript, the implementation aspects of a modern C++ based open-source data parallel tensor contraction framework named **Fastor** are presented. The framework released under the MIT licence and is accessible through `https://github.com/romeric/Fastor`. Fastor follows a different design philosophy compared to most of the available tensor algebra frameworks. It is based on statically sized arrays with a powerful in-built metaprogramming engine that allows it to perform sophisticated optimisations at compile time. In particular, its domain-aware or so-called smart expression template engine facilitates, a) transformation of chained operations to mathematically equivalent but highly efficient expressions, potentially avoiding the call to many level 3 type BLAS subroutines, b) compile time depth-first search to find the optimal contraction indices of complex tensor networks and, c) generation of customised kernels for operations on small tensors (of different order, size and data type) where typically the call to external libraries such as BLAS can be inefficient [38, 40]. The paper is organised as follows. In section 2, the governing equations of electroelasticity and the convex multi-variable constitutive equations introduced in [47–50] are presented, to showcase the focus of the current tensor contraction framework. In section 3, the interface design of the framework is discussed, by starting from the explicit SIMD vector types, the tensor class, the smart expression template engine and finally a convenient interface for tensor contraction operations using indicial notation. This is followed by a discussion on data alignment and compile time loop transformation optimisations in subsection 3.6. In section 4, a series of numerical examples are

provided. These include fundamental performance benchmarks for data parallel isomorphic and nonisomophic tensor products of tensor pairs in single and double precision floating points, the memory vs FLOP tradeoff in evaluation of large tensor networks, and their eventual compilation costs. Finally, finite element examples pertaining to the numerical integration of complex constitutive models are presented in section 5 illustrating the importance of data parallelism and domain aware expression templates.

2. Nonlinear continuum electromechanics

In this section, essential concepts of electroelasticity are discussed as a key application area for the current tensor contraction framework.

2.1. Kinematics

Let us consider the motion of an electromechanical body which in its initial configuration is defined by a domain V of boundary ∂V with outward unit normal $\boldsymbol{N}$. After the motion, the body occupies a final configuration defined by a domain v of boundary ∂v with outward unit normal $\boldsymbol{n}$, as shown in Figure 1. The pseudo-time (t) dependent mapping field $\boldsymbol{\phi}$ links a material particle from initial configuration $\boldsymbol{X}$ to final configuration $\boldsymbol{x}$ according to $\boldsymbol{x} = \boldsymbol{\phi}(\boldsymbol{X}, t)$. The deformation gradient tensor $\boldsymbol{F}$ is defined as

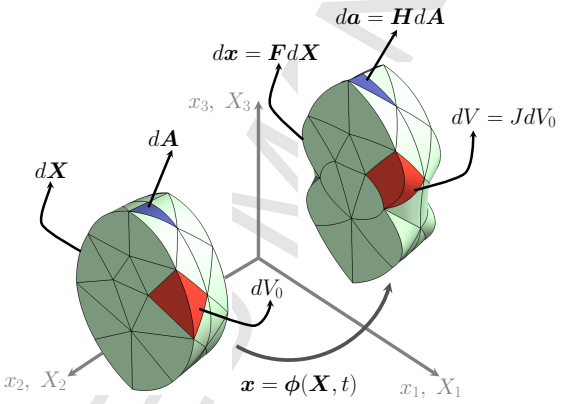


Figure 1: Motion map of a body V and the kinematic measures $\{\boldsymbol{F}, \boldsymbol{H}, J\}$.

$$\boldsymbol{F} = \boldsymbol{\nabla}_0 \boldsymbol{x} = \frac{\partial \boldsymbol{\phi}(\boldsymbol{X}, t)}{\partial \boldsymbol{X}}, \tag{1}$$

where $\boldsymbol{\nabla}_0(\bullet)$ is the Lagrangian (initial configuration) gradient operator. In addition, with the help of the tensor cross product operations, the cofactor and Jacobian ($\boldsymbol{H} = \mathrm{Cof}\boldsymbol{F}$ and $J = \det\boldsymbol{F}$) of the deformation are defined as, [1, 2, 51] [3]

$$\boldsymbol{H} = \frac{1}{2}\boldsymbol{F} \boldsymbol{\times} \boldsymbol{F}; \qquad H_{iI} = \frac{1}{2}\mathcal{E}_{ijk}\mathcal{E}_{IJK}F_{jJ}F_{kK}; \tag{2a}$$

$$J = \frac{1}{3}\boldsymbol{H} : \boldsymbol{F}; \qquad J = \frac{1}{3}H_{iI}F_{iI}. \tag{2b}$$

[3] Throughout the paper, the symbol $(\cdot)$ indicates the scalar product $\boldsymbol{a} \cdot \boldsymbol{b} = a_i b_i$, the symbol $(:)$, the double contraction operation $\boldsymbol{A} : \boldsymbol{B} = A_{ij}B_{ij}$, the symbol $(\times)$, the cross product between vectors $[\boldsymbol{a} \times \boldsymbol{b}]_i = \mathcal{E}_{ijk}a_j b_k$ and the symbol $(\otimes)$, the outer or dyadic product $[\boldsymbol{a} \otimes \boldsymbol{b}]_{ij} = a_i b_j$. The Einstein summation convention is followed throughout, wherever indices appear.

As shown in Figure 1, $\{\boldsymbol{F}, \boldsymbol{H}, J\}$ are the kinematic measures relating the differential fibre, area and volume elements from initial $\{d\boldsymbol{X}, d\boldsymbol{A}, dV\}$ to final $\{d\boldsymbol{x}, d\boldsymbol{a}, dv\}$ configuration.

2.2. Governing equations of electroelasticity

The set of equations governing the physics of an electromechanical system, can be summarised in the Lagrangian setting as follows [47, 52–57].

$$\begin{aligned} \mathrm{DIV}\boldsymbol{P} + \boldsymbol{f}_0 &= \boldsymbol{0} && \text{in} \quad V; && \text{(3a)} \\ \boldsymbol{P}\boldsymbol{N} &= \boldsymbol{t}_0 && \text{on} \quad \partial_t V && \text{(3b)} \\ \mathrm{DIV}\boldsymbol{D}_0 - \rho_0 &= 0 && \text{in} \quad V; && \text{(3c)} \\ \boldsymbol{D}_0 \cdot \boldsymbol{N} &= -\omega_0 && \text{on} \quad \partial_\omega V && \text{(3d)} \end{aligned}$$

where $\boldsymbol{P}$ is the first Piola-Kirchhoff stress tensor, $\boldsymbol{f}_0$ and $\boldsymbol{t}_0$, the Lagrangian body force and traction force vectors, respectively, $\boldsymbol{D}_0$, the Lagrangian electric displacement field, ρ_0 and ω_0, the electric charge per unit initial volume and area, respectively. Equation (3)(a,b) represents the balance of linear momentum and (3)(c,d), the Gauss's law. The rotational equilibrium dictates that $\boldsymbol{F}^T\boldsymbol{P} = \boldsymbol{P}\boldsymbol{F}^T$ and the Faraday's law in Lagrangian form can be written as $\boldsymbol{E}_0 = -\boldsymbol{\nabla}_0\phi$, with $\boldsymbol{E}_0$ the Lagrangian electric field. The described set of governing equations is subjected to suitable Dirichlet boundary condition(s).

2.3. Convex multi-variable electroelasticity

The internal energy density e, encapsulating the constitutive information necessary to close the system of governing equations, is defined as $e = e(\boldsymbol{F}, \boldsymbol{D}_0)$. Recently, Gil and Ortigosa [47, 48, 58, 59] have introduced the concept of multi-variable convexity, which satisfies the well-posedness of the governing equations described in subsection 2.2, and postulated as

$$e(\boldsymbol{F}, \boldsymbol{D}_0) = W(\boldsymbol{F}, \boldsymbol{H}, J, \boldsymbol{D}_0, \boldsymbol{d}); \qquad \boldsymbol{d} = \boldsymbol{F}\boldsymbol{D}_0, \tag{4}$$

where W represents a convex multi-variable functional in terms of the extended set of arguments $\mathcal{V} = \{\boldsymbol{F}, \boldsymbol{H}, J, \boldsymbol{D}_0, \boldsymbol{d}\}$. As presented in [47], the first Piola-Kirchoff stress tensor and the electric field vector can be expressed as

$$\boldsymbol{P} = \boldsymbol{\Sigma}_{\boldsymbol{F}} + \boldsymbol{\Sigma}_{\boldsymbol{H}} \times \boldsymbol{F} + \Sigma_J \boldsymbol{H}; \qquad P_{iI} = \Sigma_{F_{iI}} + \mathcal{E}_{ijk}\mathcal{E}_{IJK}\Sigma_{H_{jJ}}F_{kK} + \Sigma_J H_{iI}; \tag{5}$$

$$\boldsymbol{E}_0 = \boldsymbol{\Sigma}_{\boldsymbol{D}_0} + \boldsymbol{F}^T\boldsymbol{\Sigma}_{\boldsymbol{d}}; \qquad E_{0_I} = \Sigma_{D_{0_I}} + F_{Ii}\Sigma_{d_i}, \tag{6}$$

where $\Sigma_{\boldsymbol{A}} = \frac{\partial W}{\partial \boldsymbol{A}}$, where $\boldsymbol{A}$ can represent any element from the set $\mathcal{V}$. Furthermore, consistent linearisation of the governing equations leads to the constitutive tensors of the material namely, the fourth order elasticity tensor $\boldsymbol{\mathcal{C}}$, the third order piezoelectric or coupling tensor $\boldsymbol{\mathcal{Q}}$ and the second order dielectric tensor $\boldsymbol{\theta}$, defined as

$$\boldsymbol{\mathcal{C}} = \left.\frac{\partial^2 e(\boldsymbol{F}, \boldsymbol{D}_0)}{\partial \boldsymbol{F}\partial \boldsymbol{F}}\right|_{\boldsymbol{F}=\boldsymbol{\nabla}_0\boldsymbol{x}}; \quad \boldsymbol{\mathcal{Q}} = \left.\frac{\partial^2 e(\boldsymbol{F}, \boldsymbol{D}_0)}{\partial \boldsymbol{D}_0\partial \boldsymbol{F}}\right|_{\boldsymbol{F}=\boldsymbol{\nabla}_0\boldsymbol{x}}; \quad \boldsymbol{\theta} = \left.\frac{\partial^2 e(\boldsymbol{F}, \boldsymbol{D}_0)}{\partial \boldsymbol{D}_0\partial \boldsymbol{D}_0}\right|_{\boldsymbol{F}=\boldsymbol{\nabla}_0\boldsymbol{x}}. \tag{7}$$

Alternatively, following [47], the components $\boldsymbol{\mathcal{C}}$, $\boldsymbol{\mathcal{Q}}$ and $\boldsymbol{\theta}$ can be defined in terms of the set of work-conjugates $\mathcal{V}$ using Table 1, where $W_{\boldsymbol{AB}} = \frac{\partial^2 W}{\partial \boldsymbol{A}\partial \boldsymbol{B}}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ can represent any two elements from the set $\mathcal{V}$. In the context of finite elements, equations (5-7) need to be evaluated

$\boldsymbol{\mathcal{C}}$	$W_{\boldsymbol{FF}} + \boldsymbol{F}\bar{\times}(W_{\boldsymbol{HH}}\bar{\times}\boldsymbol{F}) + W_{JJ}\boldsymbol{H}\otimes\boldsymbol{H} + \boldsymbol{\mathcal{C}}_1 + 2(W_{\boldsymbol{FH}}\bar{\times}\boldsymbol{F})^{\text{sym}} + 2(W_{\boldsymbol{FJ}}\otimes\boldsymbol{H})^{\text{sym}} + 2(W_{\boldsymbol{Fd}}\otimes\boldsymbol{D}_0)^{\text{sym}} + 2((\boldsymbol{F}\bar{\times}W_{\boldsymbol{HJ}})\otimes\boldsymbol{H})^{\text{sym}} + 2((\boldsymbol{F}\bar{\times}W_{\boldsymbol{Hd}})\otimes\boldsymbol{D}_0)^{\text{sym}} + 2(\boldsymbol{H}\otimes(W_{J\boldsymbol{d}}\otimes\boldsymbol{D}_0))^{\text{sym}} + \boldsymbol{\mathcal{A}}$ where $\mathcal{A}_{iIjJ} = \mathcal{E}_{ijp}\mathcal{E}_{IJP}\left(\boldsymbol{\Sigma}_{\boldsymbol{H}} + \Sigma_J\boldsymbol{\Sigma}_{\boldsymbol{H}}\right)_{pP}$; $\mathcal{C}_{1_{iIjJ}} = (W_{\boldsymbol{dd}})_{ij}\,D_{0I}D_{0J}$ For any fourth order tensor $\boldsymbol{\mathcal{T}}$, $\mathcal{T}^{\text{sym}}_{iIjJ} = \frac{1}{2}(\mathcal{T}_{iIjJ} + \mathcal{T}_{jJiI})$
$\boldsymbol{\mathcal{Q}}^T$	$W_{\boldsymbol{FD}_0} + \boldsymbol{F}\bar{\times}W_{\boldsymbol{HD}_0} + \boldsymbol{H}\otimes W_{J\boldsymbol{D}_0} + \boldsymbol{\mathcal{Q}}_1^T + \boldsymbol{\mathcal{Q}}_2^T + \boldsymbol{\mathcal{Q}}_3^T + \boldsymbol{\mathcal{Q}}_4^T + \boldsymbol{\mathcal{Q}}_5^T$ where $[\boldsymbol{\mathcal{Q}}_1^T]_{iIJ} = [W_{\boldsymbol{dD}_0}]_{iJ}D_{0I}$; $[\boldsymbol{\mathcal{Q}}_2^T]_{iIJ} = [W_{\boldsymbol{Fd}}]_{iIj}F_{jJ}$; $[\boldsymbol{\mathcal{Q}}_3^T]_{iIJ} = [\boldsymbol{F}\bar{\times}W_{\boldsymbol{HD}_0}]_{iIj}F_{jJ}$; $[\boldsymbol{\mathcal{Q}}_4^T]_{iIJ} = [\boldsymbol{H}\otimes W_{J\boldsymbol{d}}]_{iIj}F_{jJ}$; $[\boldsymbol{\mathcal{Q}}_5^T]_{iIJ} = [W_{\boldsymbol{dd}}]_{ij}F_{jJ}D_{0_I}$
$\boldsymbol{\theta}$	$W_{\boldsymbol{D}_0\boldsymbol{D}_0} + \left(W_{\boldsymbol{D}_0\boldsymbol{d}}\boldsymbol{F} + \boldsymbol{F}^T W_{\boldsymbol{dD}_0}\right) + \boldsymbol{F}^T W_{\boldsymbol{dd}}\boldsymbol{F}$

Table 1: Elasticity tensor $\boldsymbol{\mathcal{C}}$, piezoelectric tensor $\boldsymbol{\mathcal{Q}}$ and dielectric $\boldsymbol{\theta}$ tensor re-expressed in terms of the components of the Hessian operator.

at every quadrature point. Hence, the computational cost of numerical integration would be dictated primarily by the evaluation of the work-conjugates and the Hessian of the internal energy [60].

2.4. A simple multi-variable constitutive model

A simple internal energy functional which complies with the definition of multi-variable convexity in (4), can be defined as

$$W = \mu_1 II_{\boldsymbol{F}} + \mu_2 II_{\boldsymbol{H}} \underbrace{-2(\mu_1+2\mu_2)\ln J + \frac{\kappa}{2}(J-1)^2}_{f(J)} + \tfrac{1}{2\varepsilon_1} II_{\boldsymbol{D}_0} + \tfrac{1}{2\varepsilon_2 J} II_{\boldsymbol{d}}, \quad (8)$$

where $II_{(\bullet)}$ denotes the squared of the L^2 norm of the entity $(\bullet)$ and $\{\mu_1, \mu_2, \varepsilon_1, \varepsilon_2, \kappa\}$, positive material constants. For this model, $\boldsymbol{P}$ (5) and $\boldsymbol{E}_0$ (6) are

$$\boldsymbol{P} = 2\mu_1\boldsymbol{F} + 2\mu_2\boldsymbol{H}\bar{\times}\boldsymbol{F} + \left(f'(J) - \frac{1}{2\varepsilon_2 J^2} II_{\boldsymbol{d}}\right)\boldsymbol{H}; \quad \boldsymbol{E}_0 = \frac{1}{\varepsilon_1}\boldsymbol{D}_0 + \frac{1}{\varepsilon_2 J}\boldsymbol{F}^T\boldsymbol{d},$$

and the non-zero components of the Hessian operator $\mathbb{H}_{W_1}$ are defined as

$$W_{\boldsymbol{FF}} = 2\mu_1\boldsymbol{\mathcal{I}}; \qquad W_{\boldsymbol{HH}} = 2\mu_2\boldsymbol{\mathcal{I}}; \qquad W_{JJ} = \left(f''(J) + \frac{1}{4\varepsilon_2 J^3} II_{\boldsymbol{d}}\right);$$
$$W_{J\boldsymbol{d}} = -\frac{1}{\varepsilon_2 J^2}\boldsymbol{d}; \qquad W_{\boldsymbol{D}_0\boldsymbol{D}_0} = \frac{1}{\varepsilon_1}\boldsymbol{I}; \qquad W_{\boldsymbol{dd}} = \frac{1}{\varepsilon_2 J}\boldsymbol{I}. \quad (9)$$

The tensors $\boldsymbol{\mathcal{C}}$, $\boldsymbol{\mathcal{Q}}$ and $\boldsymbol{\theta}$ can now be obtained from (9) using Table 1.

3. Interface design principle for tensorial operations

In the next subsections, the multiple stages of designing a tensor contraction framework using modern C++ features are presented, with the point of departure being the explicit SIMD vector types. It is assumed that the reader is familiar with the fundamental concepts of generic and generative programming in the context of scientific computing [23, 27, 36, 37, 61, 62]. For computatinal terminologies used in the manuscript, the reader can refer to Appendix A and for the actual implementation details, to Fastor's official repository, (available under MIT license) https://github.com/romeric/Fastor.

3.1. Data parallelism through SIMD vector types

To facilitate vector based instruction scheduling for tensorial operations, the first step is to implement explicit SIMD vector types. In the current setting, using C++ polymorphism, a set of SIMD vector types are implemented which encompass vector-enabled X86 CPU architectures from SSE to SSE4.2 and AVX to AVX2. While this is an in-built extension to the current framework, further CPU architectures such as AVX-512, MIC, Neon, AltiVec, MSA and potentially GPU support can be included by relying on the Vc library [38]. The API for the in-built extensions are kept close to that of Vc, so that in the eventual case of changing backends, a simple change of namespace should suffice. Nevertheless, in the current iteration, the framework is capable of performing vector operations on SSE2-SSE4.2/AVX-AVX2 architectures. For a full implementation of SIMD vector types and further ABI considerations, the reader can refer to [38, 46, 63].

3.2. The abstract tensor class

The point of departure for the implementation of a tensor framework is a base or an abstract tensor type. Utilising the Curiously Recurring Template Pattern (CRTP) idiom [27, 34], a straightforward implementation of the `AbstractTensor` is shown in Listing 1.

Listing 1: A canonical implementation of abstract tensor type

```
template<class Derived, FASTOR_INDEX Rank>
class AbstractTensor {
public:
    static constexpr FASTOR_INDEX Dimension = Rank;
    AbstractTensor() = default;
    FASTOR_INLINE const Derived& self() const {
        return *static_cast<const Derived*>(this);
    }
};
```

In this context, the `AbstractTensor` faciliates static binding of all derived classes to the base class, avoiding the late binding mechanism [22, 23, 62, 64], which is a key step for a successful implementation of the expression templates (notice the presence of member function `self`) [36, 37]. Additionally, note that the seemingly unnecessary template parameter, the rank of the tensor is also passed for instantiation of the `AbstractTensor`.

3.3. The tensor class

The implementation of the tensor class then follows a rather classical approach. A canonical implementation of the tensor class, removing the bounds checking and further trivial details is shown in Listing 2, where the key ingredients of the class can be summarised as follows. First, a set of data members are defined, namely the order or spatial dimension of the tensor

(`Dimension`), the total size of the tensor to be allocated in the memory (`Size`) and the stride necessary for vector instructions (`Stride`).[4]

Listing 2: A canonical implementation of the tensor class

```
template<typename T, size_t ... Rest>
class Tensor: public AbstractTensor<Tensor<T,Rest...>,sizeof...(Rest)> {
private:
    typedef T scalar_type;
    T FASTOR_ALIGN _data[product<Rest...>::value];
public:
    static constexpr FASTOR_INDEX Dimension = sizeof...(Rest);
    static constexpr FASTOR_INDEX Size = product<Rest...>::value;
    static constexpr FASTOR_INDEX Stride = stride_finder<T>::value;

    template<typename Derived, size_t Rank>
    FASTOR_INLINE Tensor<T,Rest...>& operator=(const AbstractTensor<Derived,Rank>& expr) {
        const Derived &src = expr.self();
        for (FASTOR_INDEX i = 0; i < Size; i+=Stride) {
            src.evaluate(i).store(&_data[i]);
        }
        return *this;
    }

    FASTOR_INLINE SIMDVector<T> evaluate(T i) const {
        SIMDVector<T> out;
        out.load(&_data[i]);
        return out;
    }

    template<size_t I, size_t J, size_t K>
    FASTOR_INLINE Tensor(const UnaryTraceOp<BinaryMatMulOp<Tensor<T,I,J>,UnaryTransposeOp<
        Tensor<T,K,J>>>> &a) {
        _doublecontract<T,I,K>(a.expr.lhs.expr.data(),a.expr.rhs.data());
    }
}
```

In this context, the first member function (the copy assignment operator) is responsible for static binding of any complex expression that performs element-wise operations on the tensors, to the tensor class. Note that in contrast to the classical expression templates, in the current implementation shown in Listing 2, the expression is evaluated for one SIMD vector, instead of one scalar at a time, as can be seen in the implementation of the member function `evaluate`. It is also necessary for all the expressions (such as `UnaryOps` and `BinaryOps`) to provide an `evaluate` member function. Listing 2 represents a simple example of blending expression templates and SIMD instructions (also refer to [37], for a similar implementation) for a truly multi-dimensional tensor algebra framework.

3.4. Smart expression templates: Operation minimisation through mathematical transformation

Once the tensor class is defined, what remains is the implementation of a high level interface for tensor algebraic operations. However, efficient execution of these operations does not only depend on how each individual operation/subroutine is implemented, but also on the pattern these operations are evaluated (in situations when they operate jointly on tensors). The fundamental idea of expression templates is to treat complex chain of operations as a single expression and the decision to when (how soon or late) this expression should be evaluated is called the *evaluation policy*, which typically depends on the nature and complexity of the

[4]Note that in Listing 2, it is assumed that data is appropriately aligned by the vector (register) size, the size of the tensor is a multiple of the vector stride and that no padding is necessary.

operators involved in the expression [27]. For instance, in Listing 2 the evaluation of an expression is bound to the assignment operator (`operator=`). The evaluation policy can be also overwritten at any given time by invoking the `evaluate` function explicitly.

As a common practice in implementing expression templates, all element-wise and level 1 BLAS expressions inherit from the `AbstractTensor` to facilitate delayed evaluation of chained operations, as can be seen in the copy assignment operator in Listing 2.

Fastor implements operator chaining for beyond level 1 BLAS routines through template specialiation(s) of the copy and move constructors. The idea behind chaining certain operators of level 2/3 BLAS type routines is essentially to be able to mathematically transform them to simpler expressions before evaluating them. This leads to the interesting notion of smart expression templates that facilitate exploitation of complexity reducing algorithms through mathematical equivalence and that can be viewed as more of domain specific semantics, built on top of a generic tensor algebra library.

For instance, the last member function (copy constructor) in Listing 2, provides an example of smart expression template in Fastor, where the tensor expression $\mathrm{tr}(\boldsymbol{A}\boldsymbol{B}^T)$ is dispatched to $\boldsymbol{A} : \boldsymbol{B} = A_{ij}B_{ij}$. This indeed reduces the computational complexity of the problem from $O(n^3)$ for matrix matrix multiplication, $O(n^2)$ for transpose and $O(n)$ for trace, to $O(n^2)$ for double contraction. Furthermore, vector implementation of double contraction is trivial. In the context of tensor algebra, such operations frequently occur and Fastor implements copy and move constructors for a series of such type of operations. Once again, note that expressions such as `BinaryMatMulOp`, `UnaryTransposeOp` and `UnaryTraceOp` are evaluated immediately if they act individually on a tensor. A canonical implementation of a smart expression (`UnaryTransposeOp`) is shown in Listing 3 (notice the second overload of the member function `evaluate`).

Listing 3: A canonical implementation of transpose operator

```
template<typename Expr>
struct UnaryTransposeOp {

    UnaryTransposeOp(const Expr& expr) : expr(expr) {}

    template<typename U>
    FASTOR_INLINE U evaluate(U i, U j) const {
        return expr(j,i);
    }

    const Expr &expr;
};
template<typename Expr>
FASTOR_INLINE UnaryTransposeOp<Expr> transpose(const Expr &expr) {
  return UnaryTransposeOp<Expr>(expr);
}
```

3.5. Smart expression templates: Operation minimisation through compile time depth-first constructive approach

While exploiting low-flop algorithms through mathematical transformation may be sufficient for named operators (such as `UnaryTraceOp, UnaryDetOp` etc), when the tensorial operations are expressed in indicial notation (i.e. when no named operators are present), a generalisation of the approach presented in subsection 3.4, is to perform a graph optimisation technique to find the optimal contraction indices of the tensor network. This leads to the more generic

operation minimisation technique implemented in Fastor, the so-called depth-first constructive approach, defined in Appendix A.

For the purpose of illustrating specific features and a possible function signature for tensor contraction through indicial notation, a prototypical implementation of the (`einsum`) function, between three arbitrary order tensors is presented in Listing 4, without the use of expression templates.[5]

Listing 4: Overloaded implementation of Einstein summation for three tensor singletons of arbitrary order

```
template<typename Index_I, typename Index_J, typename Index_K,
         template<typename,FASTOR_INDEX...Rest0> class Tensor0,
         template<typename,FASTOR_INDEX...Rest1> class Tensor1,
         template<typename,FASTOR_INDEX...Rest2> class Tensor2,
         typename T, FASTOR_INDEX...Rest0, FASTOR_INDEX...Rest1, FASTOR_INDEX... Rest2,
         typename std::enable_if<sizeof...(Rest0)==Index_I::NoIndices &&
                                 sizeof...(Rest1)==Index_J::NoIndices &&
                                 sizeof...(Rest2)==Index_K::NoIndices,bool>::type=0 >
FASTOR_INLINE typename ContractionType<Index_I,Index_J,Index_K,Tensor<T,Rest0...>,Tensor<T,
    Rest1...>,Tensor<T,Rest2...>>::type einsum(const Tensor0<T,Rest0...> &a, const Tensor1<T,
    Rest1...> &b, const Tensor2<T,Rest2...> &c) {
    // perform compile time depth-first search
    // if necessary call the by-pair (two tensor) overload
    // if not, set up the contraction loop nest (Cartesian product)
    // perform loop transformation and SIMD vectorisation
    // perform isomorphic/noisomorphic tensor product
}
```

Notice how the static nature of the tensors facilitates resolving the indices of the `einsum` function at compile time. With an optimising compiler this leads to no extra register allocation for the indices. This is in contrast with most of the existing tensor algebra frameworks that allocate tensors dynamically. In Listing 4, `Index` is simply a template structure (`struct`) of integral constants with one compile time (`constexpr`) data member, the `NoIndices` which accounts for the number of template parameters passed to it. Furthermore, note that the return type of `einsum` is computed at compile time through the `ContractionType` meta-function and the right amount of stack memory is allocated beforehand, which is also in contrast with the C/Fortran static arrays. A few representative examples of how the tensor objects are called and the above `einsum` function can be applied, are presented in Listing 5.

Listing 5: An example of `Tensor` instantiation and contraction of $\boldsymbol{A}_{ijk}\boldsymbol{B}_{ijlm}\boldsymbol{C}_{klmnpq}$

```
enum {I,J,K,L,M,N,P,Q};
Tensor<double,2,3,4> A; Tensor<double,2,3,5,6> B; Tensor<double,4,5,6,8,4,3> C;
// fill/populate A,B and C explicitly
A.random(); B.range(2); C.fill(42.42);
// perform tensor contraction
auto d = einsum<Index<I,J,K>,Index<I,J,L,M>,Index<K,L,M,N,P,Q>>(A,B,C);
// d is deduced as Tensor<double,8,4,3> and 96*sizeof(double)=768B is statically allocated.
```

3.6. Data alignment and compile time construction of contraction loop nests

From a computational point of view, vector or data parallel implementation of tensor contraction requires careful attention to memory alignment. In a generic tensor contraction procedure, since arbitrary indices are allowed to contract, computations along strides leading to

[5] In the presence of expression templates, overloading the `einsum` function to account for more (or less) than three tensors is not necessary, as a single tensor expression can take care of all possible overloads. For instance, the `ContractionType` in Listing 4 could as well be an expression instead of the resulting tensor and then its evaluation could be bound to a copy assignment operator similar to the one presented in Listing 2.

non-contiguous and unaligned memory access patterns and cache misses become the fundamental bottleneck. To solve this issue, some libraries designed for large dynamically allocated arrays, allow computations on general strides [65], while there are alternative frameworks that work with strict alignment and further data padding for the purpose of vectorisation [36]. Due to the strong focus of this framework on data parallelism and further due to stack allocation, the tensors are always aligned in the memory by the largest SIMD vector size that the processor is capable of i.e. `16B` alignment for SSE and `32B` alignment for AVX and so on, as shown in Listing 2 (i.e. `FASTOR_ALIGN`). However, strided data access is intrinsic to the nature of tensor contraction [26, 32], and as will be discussed shortly, Fastor employs cost-effective broadcasting vectorisation in such cases. With this decision on data alignment, Fastor classifies tensor operations into three categories, namely isomorphic tensor products (outer products), nonisomorphic tensor products (tensor contraction), and tensor permutation, as defined in Appendix A.

Once the decision on memory alignment is fixed, a variable number of nested `for` loops need to be set up depending on the contraction indices of the tensors appearing in the network. Since the `Tensor` objects presented in Listing 2 are static, Fastor uses this information to set up the contraction loop nest at compile time by generating the Cartesian product of iteration spaces of tensors. This can be typically achieved using recusrive template instantiation. To illustrate this, consider the isomorphic tensor product of singleton $[\boldsymbol{D}]_{ijk} = [\boldsymbol{a}]_i[\boldsymbol{B}]_{jk}$, where $\boldsymbol{a}$ is a first order tensor of size `2` and $\boldsymbol{B}$ is a second order tensor of `2×3`. The loop transformation procedure for this singleton is shown in Figure 2. Figure 2 is a simple example

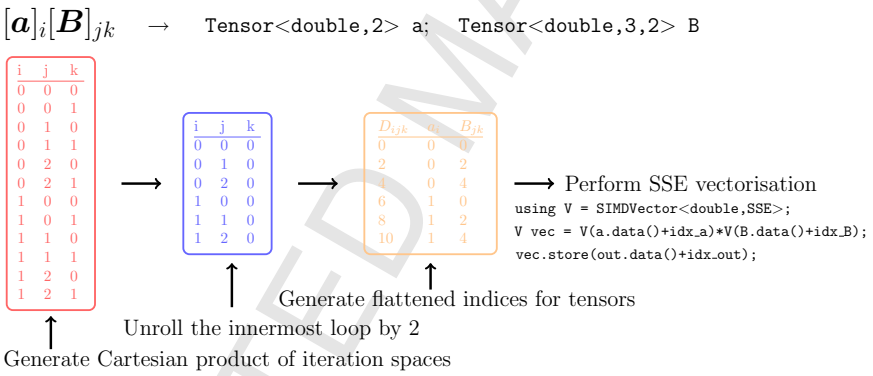


Figure 2: Loop transformation optimisation for the contraction loop nest of singleton $[\boldsymbol{D}]_{ijk} = [\boldsymbol{a}]_i[\boldsymbol{B}]_{jk}$

of loop transformation through compile time code generation. Note that, this approach is a generalisation of an efficient matrix-matrix multiplication implementation using SIMD vector types proposed as a language extension to the C++ standard committee [63]. Once performed, this type of loop transformation can facilitate other compile time optimisation opportunities, such as distinguishing loop-invariant code, subsequent hoisting and more importantly studying the vectorisability nature of the nest. In the context of vectorisability, a Cartesian product (contraction loop nest) can be classified as fully vectorisable, partially vectorisable or broadcast-vectorisable (refer to Appendix A for definitions). While partial vectorisibility implies that the final remainder operations that do not fit in the vectorised loop nest during tensor contraction procedures should be treated in a scalar fashion, broadcast-vectorisable means employing multiple cost-effective broadcast vector instructions, for computing tensor contraction on strides. This definition of vectorisability allows for an implementation strategy that enables explicit vectorisation in floating point as well as in memory load and store operations. To elaborate

this vectorisation procedure, a schematic representation of a broadcast-vectorisable tensor pair is shown in Figure 3 for the nonisomorphic tensor product $[\boldsymbol{C}]_{im} = [\boldsymbol{A}]_{ijkl}[\boldsymbol{B}]_{mjkl}$.

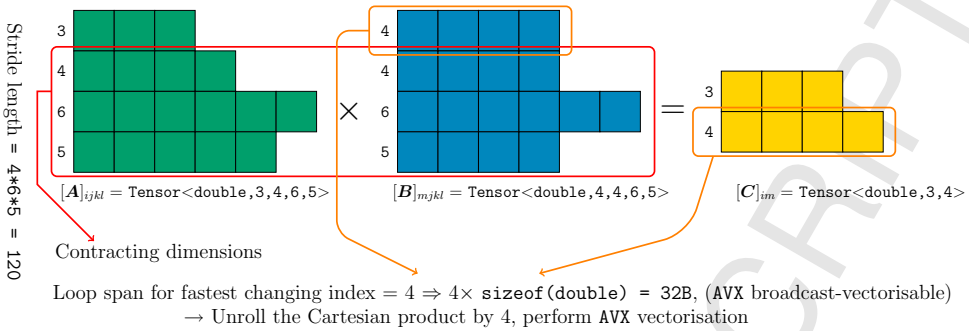


Figure 3: AVX vectorisation of the nonisomorphic tensor product of singleton $[\boldsymbol{C}]_{im} = [\boldsymbol{A}]_{ijkl}[\boldsymbol{B}]_{mjkl}$ on strides (cells represent tensor's dimensions not the register width).

Notice that, in Figure 3 m is the fastest changing index of the nest $ijklm$, and the memory access for tensor $\boldsymbol{B}$ requires broadcasting intrinsics [66] of the tensor into AVX registers (as opposed to aligned loading) by an `offset=120`, while the floating point and memory I/O operations on tensors $\boldsymbol{A}$ and $\boldsymbol{C}$ remain fully AVX vectorisable.

Finally, having performed loop transformation (Figure 2) and SIMD vectoristation (Figure 3), the third optimisation step is to perform loop fusion. Loop fusion is a direct consequence of operator chaining applied on tensor networks. At its current iteration, Fastor tries to obey the ISO C standard on strict aliasing rules. This implies that, if a network comprising of many singletons is to be evaluated, the contraction of each singleton is evaluated individually into temporaries and the loop fusion is then applied at the top level, in order to avoid chaining of singletons of different complexity and hence memory aliasing. To elaborate this, consider the evaluation of the tensor network $[\boldsymbol{G}]_{jkl} = [\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{il} + \rho\mathrm{tr}(\boldsymbol{I})[\boldsymbol{c}]_k[\boldsymbol{D}]_{jl} + \sqrt{[\boldsymbol{E}]_{jkl}}$, where ρ is a constant coefficient, $\boldsymbol{I}$ is the second order identity tensor in $\mathbb{R}^{3\times3}$ and $\boldsymbol{c}$, $\boldsymbol{D}$ and $\{\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{E}, \boldsymbol{G}\}$ are first, second and third order tensors of arbitrary size, respectively. The loop fusion procedure applied on this network is shown in Figure 4. In Figure 4, ρ being a constant and

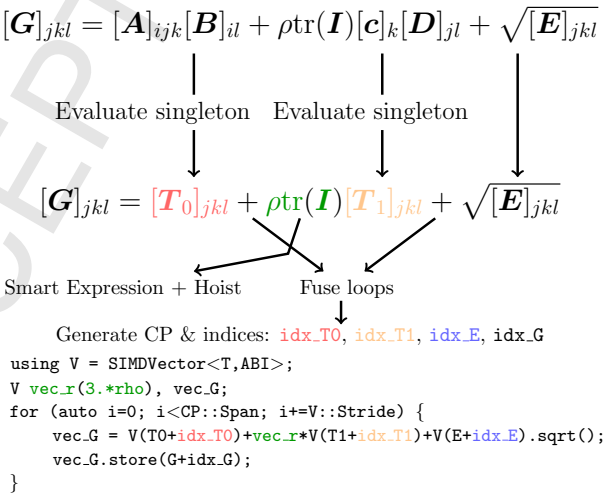


Figure 4: Loop fusion for the evaluation of the network $[\boldsymbol{G}]_{jkl} = [\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{il} + \rho\mathrm{tr}(\boldsymbol{I})[\boldsymbol{c}]_k[\boldsymbol{D}]_{jl} + \sqrt{[\boldsymbol{E}]_{jkl}}$

tr($\boldsymbol{I}$) a Fastor smart expression, are hoisted out of the loop during the evaluation of singleton $[\boldsymbol{T}_1]_{jkl} = [\boldsymbol{c}]_k[\boldsymbol{D}]_{jl}$. After evaluation of all singletons, the Cartesian product (CP) is computed and a single vectorised loop is set up to compute $\boldsymbol{G}$. The cost of creating these temporaries is evaluated in subsection 4.2 in the context of compile time operation minimisation. The future endeavour for extending Fastor is to explore batched evaluations of intermediate singletons in a multi-threaded environment as also recently presented in the context of GPGPU in [67].

It is worth mentioning that, generating the Cartesian product of the iteration space and the indices of the tensor network metaprogrammatically can lead to dramatic increase in compilation time. Depending on the level of optimisation required, each individual step described above can be performed either at compile time or run time, by issuing the `-DCONTRACT_OPT` to the compiler. These optimisation steps and further compilation aspects of tensor contraction in Fastor are studied in more detail in subsection 4.3.

4. Benchmark examples

In this section, a series of fundamental benchmarks are presented to highlight further aspects of the tensor contraction framework. In particular, the benchmarks presented in the next few subsections examine the following aspects of the framework:

1. Impact of SIMD vectorisation on the performance of tensor contraction of arbitrary order tensors
2. Impact of operation minimisation on tensor contraction and the associated memory vs FLOPs tradeoff for various cache heirarchies
3. Compilation aspects of the framework and the impact of aggressive loop transformation

All the numerical examples are run on Intel(R) Xeon(R) CPU E5-2650 v2 @2.60GHz processor running Ubuntu 16.04. The processor has three cache levels namely, an 8 way associative private `32KB` L1 data cache (and `32KB` L1 instruction cache), an 8 way associative private `256KB` L2 cache and a 20 way associative shared `20MB` L3 cache, in addition to a `32GB` DDR3 (1866 MHz) RAM. Furthermore, it supports SSE to SSE 4.2 and AVX, but not AVX2 or FMA instructions. The following three compilers are used for the benchmarks namely, GCC 6.2.0, Clang 3.9.0 and ICC 17.0.1. If the compiler and Fastor's optimisation level are not specified, GCC and `-CONTRACT_OPT` should be assumed, respectively. All the reported CPU run-times are measured by taking the average of one million calls, unless the benchmark took a considerably long time finish, in which case the number of calls where lowered by a factor of 10. Furthermore, in the SIMD tensor contraction benchmark presented in the next subsection, the performance was measured by turning off the turbo mode (although similar performance traits were observed under the turbo mode). The raw data, pre-built binaries and Python scripts for visualisation, for all the numerical examples presented in the manuscript are accessible through https://github.com/romeric/LogfilesFastor.

4.1. Impact of SIMD vectorisation on the performance of tensor contraction of arbitrary order tensors

As a starting point, it is important to verify that the tensor contraction framework achieves predicted speed-ups from explicit SIMD vectorisation and that no significant overhead is introduced by the underlying layers of abstractions. To this end, the present benchmark attempts to analyse the speed-ups achieved in FLOP and memory I/O over the scalar code (in the sense of [38], it is a combination of arithmetic and memory I/O benchmark, but rather in the context

of tensor contraction). Here, scalar code refers to a variant of the implemenation where no explicit vectorisation has been performed and the compiler auto-vectoriser is purposely turned off. This is important in order to assess if the framework achieves the theoretical maximum of SSE/AVX FLOPs and (read/write) bytes per cycle.

In the present benchmark, the interest is in the run-time performance of SIMD vectorised tensor contraction of tensor pairs, hence, operation minimisation is not performed. Furthermore, both vector and scalar variants are based on the same contraction loop nest and are both compiled with "`-O3 -mavx`" options. Furthermore, for the purpose of benchmarking, it was necessary to ensure that the scalar code was not vectorised by passing `-fno-tree-vectorize` to GCC, `-fno-vectorize` to Clang and `-no-vec` to ICC and carefully examining the generated assembly codes. The internal level of optimisation utilised for these benchmarks correspond to the default option `-DCONTRACT_OPT=0`. This optimisation level is indeed equivalent to writing the contraction loop nest explicitly as multiple nested for loops and relying on the compiler for further optimisations. This is indeed also important in order to completely isolate and measure the performance of SIMD vectorisation. Fastor's further internal optimisation levels are studied in subsection 4.3.

Figure 5 presents the speed-ups gained from SIMD vectorisation over the scalar code, for isomorphic tensor products of arbitrary order tensors. These include outer products of the following pairs $[\boldsymbol{A}]_{ij}[\boldsymbol{B}]_{kl}$, $[\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{lmn}$, $[\boldsymbol{A}]_{ijkl}[\boldsymbol{B}]_{mnpq}$, $[\boldsymbol{A}]_{ijklm}[\boldsymbol{B}]_{npqrs}$, $[\boldsymbol{A}]_{ijklmn}[\boldsymbol{B}]_{pqrstu}$ and $[\boldsymbol{A}]_{ijklmnpq}[\boldsymbol{B}]_{rstuvwxy}$, where the size of tensors $\boldsymbol{A}$ and $\boldsymbol{B}$ are kept identical. Furthermore, to assess the performance of SIMD vectorisation, the last dimension of the tensors are chosen to be a multiple of SSE (4 for single precision and 2 for double precision) and/or AVX (8 for single precision and 4 for double precision) registers. It should be emphasised that, Figure 5 benchmarks are relative to the scalar code and essentially show the speed-up in load and store (bytes transferred per cycle) as well as floating point operations (only multiplication in case of outer product). While for arithmetic operations optimal speed-ups can be achieved, the load and store operations strongly depend on the size of the tensor. In the benchmarks presented in this section, the smallest tensor size resulting from the outer product is `288B = 0.0087890625`×L1 cache and the largest tensor size is `1MB = 4`×L2 cache.

Next, the performance of nonisomorphic tensor contraction of pairs of arbitrary order tensors are analysed against the scalar code, with the analysis parameters and the compiler options remaining the same as before. However, in contrast to the case of outer product of tensors, the number of loops to be set up/merged depend on the number of contracting indices (see Figure 3) and the floating point operations performed are multiplication followed by addition. Figure 6 shows the speed-up achieved in tensor contraction of pairs of tensors using SIMD vector types over the scalar code. For this benchmark all tensors fit in to `L1` cache.

As can be observed, in both isomorphic (outer product) and nonisomorphic tensor product benchmarks, optimal speed-ups are achieved with SSE (4X for SP and 2X for DP) as well as AVX (8X for SP and 4X for DP) vectorisation over the scalar variant. Certainly, for the isomorhpic tensor products where tensors do not fit in `L1` cache Figure 5(e,f), a more noticable degradation in speed-up is observed. The variations in speed-up with different tensor sizes can be explained by carefully studying the assembly code generated by the compilers. In that, since tensor objects in Fastor are static, an optimising compiler may generate different intrinsics for different tensor sizes, which may be optimal for some but not the other. As a particular example, consider the case of AVX vectorisable double precision tensor products in Figure 5

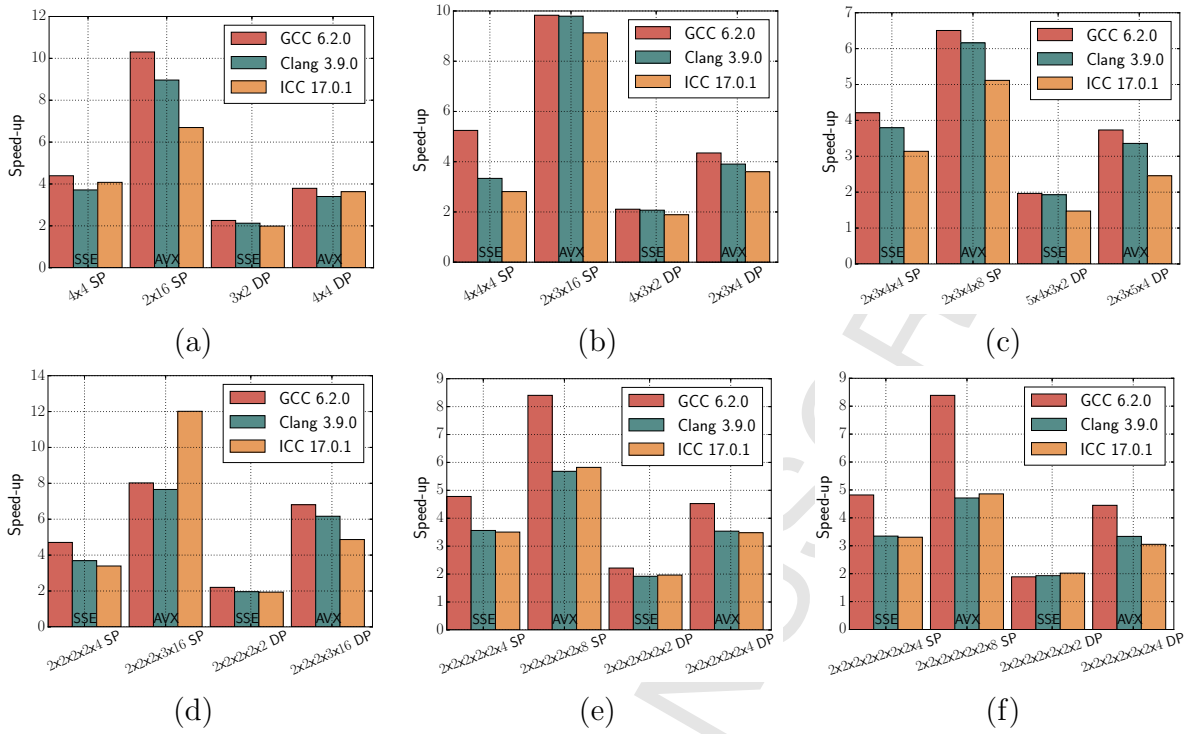


Figure 5: Speed-ups achieved by SIMD vectorisation in perfoming outer product of tensors of single precision (SP) and double precision (DP) floating point over the scalar version with tensors of order (a) 2 ($[\boldsymbol{A}]_{ij}[\boldsymbol{B}]_{kl}$), (b) 3 ($[\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{lmn}$), (c) 4 ($[\boldsymbol{A}]_{ijkl}[\boldsymbol{B}]_{mnop}$), (d) 5 ($[\boldsymbol{A}]_{ijklm}[\boldsymbol{B}]_{nopqr}$), (e) 6 ($[\boldsymbol{A}]_{ijklmn}[\boldsymbol{B}]_{opqrst}$) and, (f) 8 ($[\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{pqrstuvwxy}$). Since the order and dimension of the tensors in outer product are kept the same, a 4×4 for instance, essentially implies $[\boldsymbol{A}]_{4\times4} \otimes [\boldsymbol{B}]_{4\times4}$.

and Figure 6, for whom the three compilers used for this benchmark generate the following different intrinsics for the innermost loop:

1. **GCC 6.2.0** generates one additional move (`movslq`), one add (`addq`) and one shift and rotate instruction (`salq`) and emits aligned load and store instructions for all tensors.
2. **Clang 3.9.0** generates the most compact code with no additional instructions, but changes the aligned load instruction for tensor $\boldsymbol{A}$ to a broadcast (`vbroadcastsd`), and emits aligned store instructions.
3. **ICC 17.0.1** generates two additional move (`movslq`) and two shift and rotate instructions (`shlq`), changes the aligned load instruction for tensor $\boldsymbol{A}$ to a broadcast (`vbroadcastsd`) and emits an unaligned store instruction (`vmovupd`) for the output tensor. This explains the slight drop in speed-up observed with ICC in certain cases.

It is important to clarify this aspect through inspecting the generated intrinsics, since the vectorisation approach followed by Fastor is through explicit SIMD vector types, which at times, is known to be sensitive to compiler's mis-compilation [27, 38].

4.2. The depth-first search approach and memory vs FLOPs tradeoff

As defined in Appendix A, the depth-first search is an operation minimisation technique for tensor contraction over complex networks that performs by-pair tensor contraction in the network. This essentially leads to the creation of multiple intermediate temporaries and hence

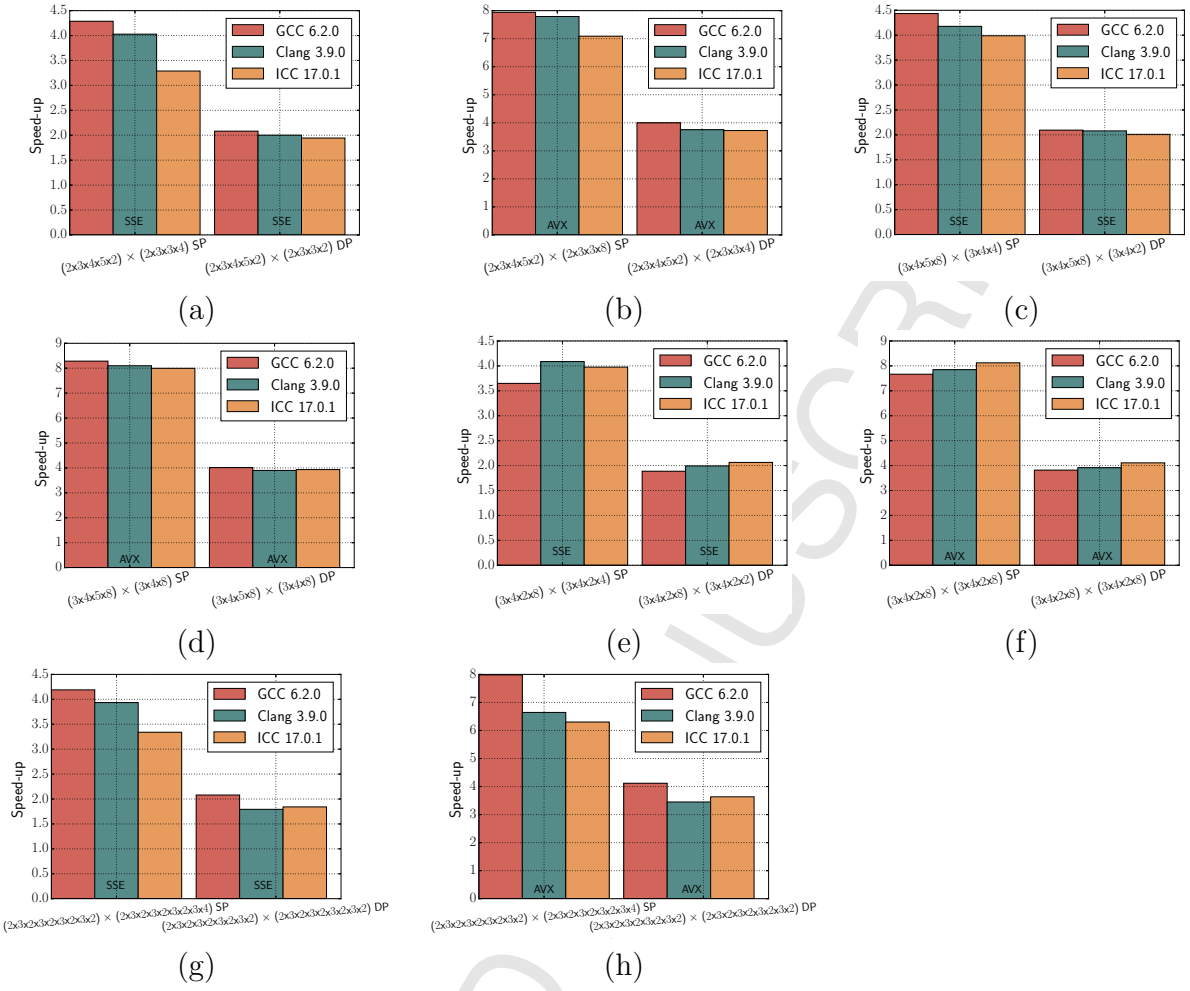


Figure 6: Speed-ups achieved by SIMD vectorisation in perfoming tensor contraction on pairs of tensors of single precision (SP) and double precision (DP) floating point over the scalar version with (a) single index contraction ($[\boldsymbol{A}]_{ijklm}[\boldsymbol{B}]_{njop}$) using SSE, (b) single index contraction ($[\boldsymbol{A}]_{ijklm}[\boldsymbol{B}]_{njop}$) using AVX, (c) two index contraction ($[\boldsymbol{A}]_{ijkl}[\boldsymbol{B}]_{ijm}$) using SSE, (d) two index contraction ($[\boldsymbol{A}]_{ijkl}[\boldsymbol{B}]_{ijm}$) using AVX, (e) three index contraction ($[\boldsymbol{A}]_{ijkl}[\boldsymbol{B}]_{ijkm}$) using SSE, (f) three index contraction ($[\boldsymbol{A}]_{ijkl}[\boldsymbol{B}]_{ijkm}$) using AVX, (g) 8 index contraction ($[\boldsymbol{A}]_{ijklmnopq}[\boldsymbol{B}]_{ijklmnopr}$) using SSE and, (h) 8 index contraction ($[\boldsymbol{A}]_{ijklmnopq}[\boldsymbol{B}]_{ijklmnopr}$) using AVX. x-labels represent the actual dimensions of the tensors.

can be perceived as a performance bottleneck. An alternative methodology to this is to evaluate the whole tensor network at once, without performing a depth-first algorithm. On the other hand, if the tensors are statically allocated (like in Fastor) and the temporaries created by the depth-first search fit in CPU cache, the operation minimistion may instead prove to be more cache optimal, by virtue of increasing the spatial locality. In this section, a fundamental benchmark is manufactured that compares the run-time perfomance of the by-pair (termed here as FLOP optimal) tensor contraction algorithm against the single expression evaluation (termed here as memory-saving) algorithm. Note that, these benchmarks do not include the performance of the depth-first search approach in finding the optimal sequence of tensor contraction. As the depth-first search algorithm itself is performed metaprogrammatically, no run-time code is generated for it. The compilation time and further performance aspects of the depth-first

search are studied in subsection 4.3.

To be able to compare the performance of the two aforementioned approaches, a three tensor singleton ($[\boldsymbol{D}]_{kmn} = [\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{ijl}[\boldsymbol{C}]_{mnl}$) is chosen first. The cost of contracting this network through a single evaluation is $3\xi_i\xi_j\xi_k\xi_l\xi_m\xi_n$, where ξ_a denotes the iteration space of index a and the 3 stands for 2 multiplications and one addition. A fixed sequence for by-pair tensor contraction is then chosen, namely $[\boldsymbol{T}]_{kl} = [\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{ijl} \Rightarrow [\boldsymbol{D}]_{kmn} = [\boldsymbol{T}]_{kl}[\boldsymbol{C}]_{mnl}$, where $\boldsymbol{T}$ represents the temporary. The total cost of the contraction over the network is now $2(\xi_i\xi_j\xi_k\xi_l + \xi_k\xi_l\xi_m\xi_n)$, where 2 stands for one multiplication and one addition.

Note that depending on the iteration space of tensors the above sequence may not always correspond to the most optimal one. Nonethless, an attempt is made to keep the above sequence optimal either by manually choosing the sizes of tensors $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ or through the compiler flag `-DFASTOR_KEEP_DP_FIXED`. The sizes of the tensors are then successively increased in such a way that the memory requirement for the intermediate temporary $\boldsymbol{D}$ ranges from fitting into `L1` cache to four times the size of `L3` cache. Certainly, as the dimensions increase, the number of floating point operations also increase. The assessment is then based on how much reduction in FLOP count is necessary to outweigh the cost of allocation of the temporary i.e. for a temporary fitting into a given cache, what should be the approximate `reduced FLOP count`, where

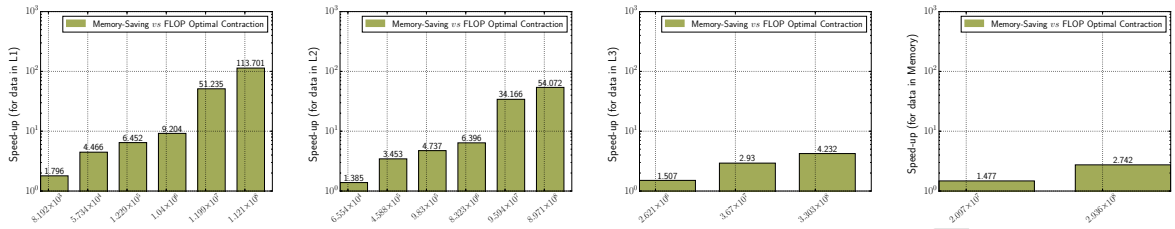
$$\begin{aligned}\texttt{reduced FLOP count} &= \texttt{cost}_{\texttt{MemOpt}} - \texttt{cost}_{\texttt{FLOPOpt}} \\ &= 3(\xi_i\xi_j\xi_k\xi_l\xi_m\xi_n) - 2(\xi_i\xi_j\xi_k\xi_l + \xi_k\xi_l\xi_m\xi_n).\end{aligned}$$

Figure 7 shows the speed-up of FLOP optimal contraction over the memory-saving contraction scheme, while keeping the temporary size to fit into a fixed cache and successively increasing the iteration space in order to increase the `reduced FLOP count`. Hence, every bar in Figure 7, compares the two schemes for a fixed data size and fixed `reduced FLOP count`. While the contraction loop nest certainly differs, it is made sure that both algorithms are equally vectorised (in particular, using AVX intrinsics), in order to fully isolate the aspect of SIMD vectorisation from operation minimisation. Hence, depending on the vectorisation, the reduction in FLOP count should be divided by the vector size (which is not done here for the purpose of clarity).

As can be observed in Figure 7, for tensor networks fitting `L1` cache, even a reduction of 100-1000 FLOPs through the depth-first scheme can be beneficial. For `L2` cache a saving of 10^3 or more in FLOPs is needed to actually outweigh the single expression evaluation scheme. Similarly, for `L3` cache, a reduction of 10^6 and for tensor networks not fitting in any cache, a reduction of (10^7) in floating point operations is required for the by-pair tensor contraction scheme to be beneficial. Certainly, as the saving in FLOP increases for a given cache size, orders of magnitude performance can be gained over the memory-saving approach.

The same benchmark is then repeated with a different tensor network, namely the four tensor singleton $[\boldsymbol{E}]_{lo} = [\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{ijl}[\boldsymbol{C}]_{mnk}[\boldsymbol{D}]_{mno}$, where the performance of contracting this network through a single expression evaluation (with cost $4\xi_i\xi_j\xi_k\xi_l\xi_m\xi_n\xi_o$) is compared with the by-pair approach for the sequence $[\boldsymbol{T}_0]_{kl} = [\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{ijl}$; $[\boldsymbol{T}_1]_{ko} = [\boldsymbol{C}]_{mnk}[\boldsymbol{D}]_{mno} \Rightarrow [\boldsymbol{E}]_{lo} = [\boldsymbol{T}_0]_{kl}[\boldsymbol{T}_1]_{ko}$ requiring the creation of two temporaries now (with a total cost of $2(\xi_i\xi_j\xi_k\xi_l+\xi_m\xi_n\xi_k\xi_o+\xi_k\xi_l\xi_0)$), hence

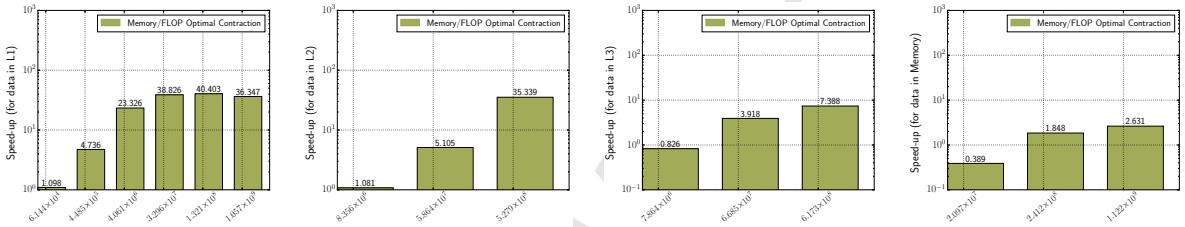
$$\begin{aligned}\texttt{reduced FLOP count} &= \texttt{cost}_{\texttt{MemOpt}} - \texttt{cost}_{\texttt{FLOPOpt}} \\ &= 4(\xi_i\xi_j\xi_k\xi_l\xi_m\xi_n\xi_o) - 2(\xi_i\xi_j\xi_k\xi_l + \xi_m\xi_n\xi_k\xi_o + \xi_k\xi_l\xi_0).\end{aligned}$$



(a) (b) (c) (d)

Figure 7: Speed-up achieved for contraction of three tensor singletons ($[\boldsymbol{A}]_{ijk}[\boldsymbol{B}_{ijl}[\boldsymbol{C}]_{mnl}$) using by-pair contraction over single expression evaluation for tensor sizes that fit a) `L1` cache (size of temporary created `16KB=0.5×L1` cache), b) `L2` cache (size of temporary created `128KB=0.5×L2` cache), c) `L3` cache (size of temporary created `10MB=0.5×L3` cache) and, d) main memory (size of temporary created `80MB=4×L3` cache). x-labels indicate the number of FLOPs saved/reduced by utilising by-pair (FLOP optimal) contraction and numbers on top of bars show the corresponding speed-up.

Figure 8 shows the speed-up of FLOP optimal contraction over memory-saving contraction



(a) (b) (c) (d)

Figure 8: Speed-up achieved for contraction of four tensor singletons ($[\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{ijl}[\boldsymbol{C}]_{mnk}[\boldsymbol{D}]_{mno}$) using by-pair contraction over single expression evaluation for tensor sizes that fit a) `L1` cache (size of temporary created `16KB=0.5×L1` cache), b) `L2` cache (size of temporary created `128KB=0.5×L2` cache), c) `L3` cache (size of temporary created `10MB=0.5×L3` cache) and, d) main memory (size of temporary created `80MB=4×L3` cache). x-labels indicate the number of FLOPs saved/reduced by utilising by-pair (FLOP optimal) contraction and numbers on top of bars show the corresponding speed-up.

scheme, while keeping the sizes of the temporaries to fit into fixed cache and successively increasing the iteration space in order to increase the `reduced FLOP count`.

Certainly, the performance is now affected by the creation of two temporaries. For tensor networks fitting `L1` cache a reduction of 10^4 in FLOPs, for tensor networks fitting `L2` cache a reduction of 10^6 in FLOPs, for tensor networks fitting `L3` cache a reduction of 10^7 in FLOPs and for tensor networks not fitting in any cache a reduction of $> 10^8$ in FLOP count is required for the by-pair tensor contraction scheme to outperform the single expression evaluation scheme.

One can correlate these numbers to the latency of data fetch (from different caches) of the architecture. For the `L1` cache of the tested Intel Xeon processor, the ratio of bytes read/writen (4 cycle latency) to floating point operation (5 cycles `mul` + 3 cycles `add`) per each iteration of contraction loop nest is going to be small enough (refer to [66] for various cache latencies). Hence, for tensors fitting in `L1` cache, even minimal savings in FLOPs can be beneficial.

Note that, due to the design of the current tensor contraction framework, all temporaries are allocated on the stack. Different (and perhaps more in favour of the memory-saving approach) performance traits should be expected for dynamic and heap allocated tensors. Since the reduction in FLOP count is correlated to the size of tensors, it may not always be apparent to choose one scheme over the other, and that leads to the idea of a heuristic cost model [14, 41].

It is worth mentioning that, these benchmarks essentially relate the correlation of fixed memory allocations to the sequence (evalution pattern) of operation minimisation. In cases where the sequence is not fixed, the operation minimisation may prove to be even more beneficial [68].

4.3. Compilation aspects of operation minimisation and further compile time tensor contraction optimisations

As described in subsection 3.6, generating the Cartesian product of iteration space and further the indices of tensors metaprogrammatically can lead to an increase in compilation time. In this subsection, compilation aspects of Fastor are studied under two settings. First, the different optimisation levels available in Fastor for tensor contraction (supplied as compiler flag `-DCONTRACT_OPT`) are studied. These optimisation levels essentially relate to the construction of contraction loop nests (i.e. the Cartesian products) and loop transformation optimisations described in subsection 3.6 which are in fact not related to operation minimisation. Second, the compilation aspects of operation minimisation is studied and compared to the compilation aspects of single expression evaluation scheme. The latter study is indeed related to the benchmarks described in previous subsection on runtime perfomance of the two schemes.

As Fastor can compute the Cartesian product of contraction loop nest either at compile time or at runtime, this leads to the following three optimisation levels

1. `-DCONTRACT_OPT=0` [default]: The Cartesian product is computed at runtime. This is equivalent to explicitly writing the contraction loop nest and relying on the compiler to optimise it.
2. `-DCONTRACT_OPT=1`: The Cartesian product is computed at compile time and stored in variadic template containers, but the indices of tensors are computed at runtime. In that, the cost of indexing and memory access is still present at run time.
3. `-DCONTRACT_OPT=2`: The Cartesian product and the indices of tensors are computed at compile time and stored in variadic template containers. This is an extreme level of optimisation which completely eliminates dynamic memory I/O (at least in theory) and mandates that only floating point arithmetics should be performed at run time.

Note that, irrespective of the optimisation levels described above, Fastor will always have enough information to perform loop enrolling and vectorisation, as the information regarding the iteration space of tensors is available at compile time.

To study the various aspects of the above optimisation levels, a singleton comprising of one 7th order tensor $\boldsymbol{A}$ and one 8th order tensor $\boldsymbol{B}$ is considered. The singleton is then contracted such that the nonisomorphic vector space is successively increased. These correspond to 7 index contraction $[\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijklmnop}$, 6 index contraction $[\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijklmnpr}$, 5 index contraction $[\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijklmprs}$, 4 index contraction $[\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijklprst}$, 3 index contraction $[\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijkprstu}$, 2 index contraction $[\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijprstuv}$ and finally one index contraction $[\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{iprstuvw}$, respectively. The dimensions of the tensors are chosen such that each of the aformentioned multi-index contractions correspond to a Cartesian product with dimesnions $2^8 \times 8$, $2^9 \times 9$, $2^{10} \times 10$, $2^{11} \times 11$, $2^{12} \times 12$, $2^{13} \times 13$ and $2^{14} \times 14$, respectively. In fact, these dimensions range from small to large, in order to assess the compilation times for all kinds of feasible applications. Four aspects of these optimisation levels are then studied namely, the compilation time, memory footprint, generated binary size and the eventual execution time of each. All the benchmarks are run with double precision AVX vectorisable nests with compiler flags as "`-O3 -mavx`". Although various compiler flags could be used to optimise for generated

binary sizes and sanitise memory footprint, they would lead to extensive parametric studies, which is not the purpose here. The goal here is, to study Fastor's internal optimisation schemes with realistic compiler flags (also in order to be consistent with the other benchmarks). The only two additional flags used are `-Wstack-usage` for GCC and `-fconstexpr-steps=16000000` for Clang.

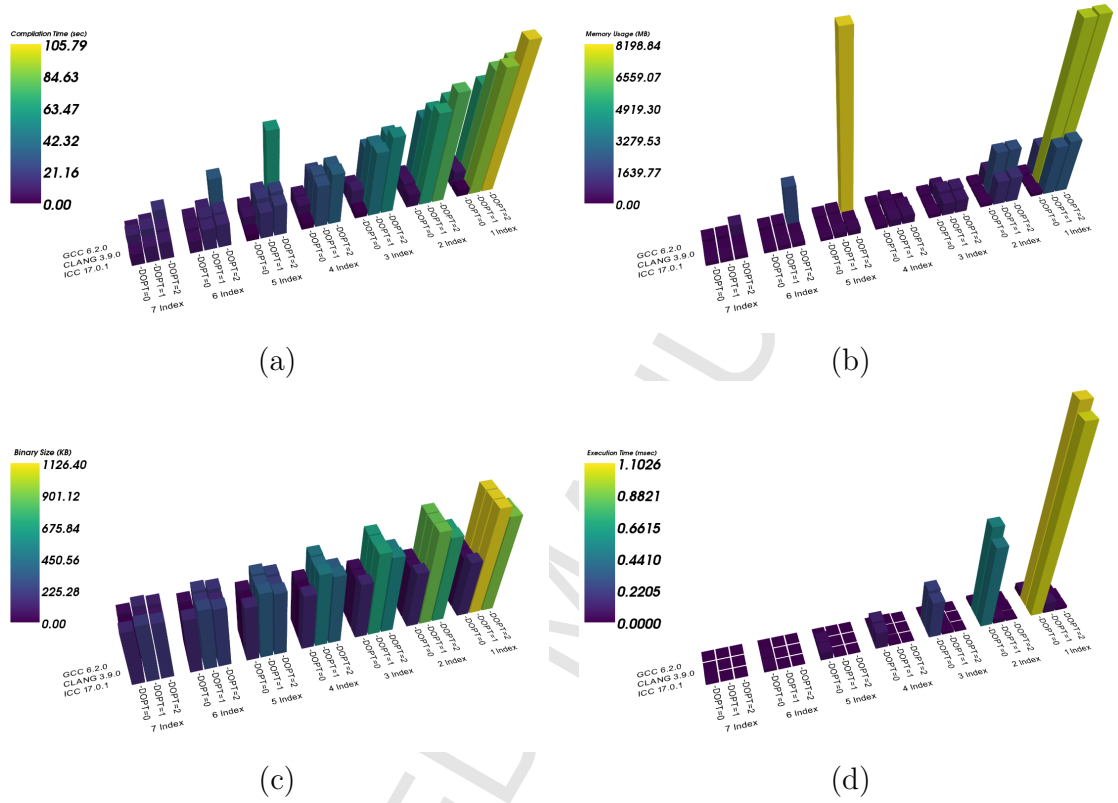


Figure 9: Compilation aspects of different optimisation levels for multi-index tensor contraction of singletons (Lower is better). a) compilation time (wall time), b) memory footprint, c) size of binaries generated, and d) eventual execution time (wall time). Contraction indices correspond to: (7 index)$\rightarrow \boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijklmnop}$ (span $2^8 \times 8$), (6 index) $\rightarrow [\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijklmnpr}$ (span $2^9 \times 9$), (5 index) $\rightarrow [\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijklmprs}$ (span $2^{10} \times 10$), (4 index) $\rightarrow [\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijklprst}$ (span $2^{11} \times 11$), (3 index) $\rightarrow [\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijkprstu}$ (span $2^{12} \times 12$), (2 index) $\rightarrow [\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{ijprstuv}$ (span $2^{13} \times 13$)and (1 index) $\rightarrow [\boldsymbol{A}]_{ijklmno}[\boldsymbol{B}]_{iprstuvw}$ (span $2^{14} \times 14$), respectively. Note that data for GCC 6.2.0 for 4 index contraction and lower is not available for optimisation level `-DOPT=2`, due to stall and excessive memory footprint. `-DOPT` is used as a shorter name alias for `-DCONTRACT_OPT`.

From Figure 9 (shown as raw data), the first observation is that `-DCONTRACT_OPT=0` has the compilation time and memory footprint of a typical application in C++, with all the compilers. The second observation is that for `-DCONTRACT_OPT=2`, GCC compilation time and memory usage (8.2GB memory footprint) increases exponentially for 5 index contraction (i.e. for the span $2^{10} \times 10$) and eventually consumes all the available memory and stalls for 4 index contraction (i.e. for the span $2^{11} \times 11$). Further build profiling reveals that unlike ICC and Clang, GCC stores up all large variadic templates and static arrays on the stack in order to perform global optimisation for fixed indices, but does not optimise the memory I/O. A deeper insight can be gained through a comparison of different optimisation levels presented in Table 2

and Table 3. As can be observed, the memory usage and compile time increases quadratically for Clang starting from 4 index contraction, under the two latter optimisation levels. ICC shows the least memory footprint (up to ≈2GB) for both higher optimisation levels and GCC and Clang show the shortest compilation time for `-DCONTRACT_OPT=1` and `-DCONTRACT_OPT=2`, respectively. The size of generated binaries are all comparable for all compiler for a fixed optimisation level. Clang generates a slightly more compact code compared to the other two compilers for `-DCONTRACT_OPT=0` and ICC generates the most compact code for the other two optimisation levels.

These results impact the run time accordingly. GCC compiled codes, show no signifcant improvement in run time since the stack size is increased but memory I/O is still present. However, at the cost of high memory usage and compilation time, ICC and Clang completely optimise away the run time memory I/O and generate codes with `-DCONTRACT_OPT=2` which is more than 80X faster than that of `-DCONTRACT_OPT=0` and nearly up to 5X faster than `-DCONTRACT_OPT=1` (see Figure 9d). Note that, these performance gains come on top of the benchmarks presented in subsection 4.1 for vectorisation (as here all the generated codes are AVX vectorised). However, as can be seen from the raw data in Figure 9, for `-DCONTRACT_OPT=0`, GCC emits fully aligned memory load and store instructions which perform slightly faster than their counterparts from the other compilers (as also explained in subsection 4.1).

It should be mentioned that `-DCONTRACT_OPT=1,2` correspond to optimisation levels that neither a compiler would be willing to perform nor are they available in any vendor specific libraries, such as BLAS. In fact, their implementation, require building a compile time virtual engine to perform numerical analysis on template parameters (analogous to a numerical analysis software that performs computations on arrays at runtime). Certainly, the complexity involved in developing a metaprogramming engine of such sort is tremendous and the benchmarks in Figure 9 reflect that. For instance, at `-DCONTRACT_OPT=2`, Fastor performs sorting, concatenation, reshaping and many more operation of $(n \log(n))$ on `std::integer_sequence`s (C++14) with length n more than 1000. This shows that the metaprogramming engine in Fastor extends much beyond expression templates and operator chaining.

The motivation behind implementing these optimisations, is due to the domain specific nature of Fastor to primarily optimise tensor contractions in finite element computations where extremely large tensors rarely occur, and for which the compilation time could remain in seconds and the memory footprint would be in the range of a few 100 megabytes at most. Note that, a more in-depth analysis of template instantiation and compile time profiling is beyond the scope of this study. For such diagnostic studies (carried out using the LLVM based Templight [69, 70]), the interested reader can refer to `https://github.com/romeric/LogfilesFastor` for more details.

↓ \→	Compilation Time			Memory Usage			Binary Size			Execution Time		
	GCC	Clang	ICC	GCC	Clang	ICC	GCC	Clang	ICC	GCC	Clang	ICC
7 Index	1.127	1.112	1.189	1.084	1.116	1.103	0.274	0.353	0.315	0.7	0.669	7.388
6 Index	1.253	1.34	1.395	1.18	1.29	1.241	0.347	0.511	0.383	1.038	17.352	12.565
5 Index	1.564	1.745	2.016	1.384	1.751	1.539	0.495	0.746	0.516	1.239	22.809	17.058
4 Index	2.082	2.64	3.423	1.836	3.064	2.195	0.813	1.253	0.832	1.004	27.694	16.102
3 Index	3.34	4.214	7.033	2.686	7.261	3.638	1.542	2.428	1.501	1.012	28.171	15.837
2 Index	6.212	10.086	13.435	4.748	21.449	6.767	3.111	4.88	2.905	0.938	30.133	17.053
1 Index	12.78	23.506	44.008	9.296	72.99	13.52	6.935	11.006	6.277	0.778	22.376	15.942

Table 2: Compilation aspects & run time performance of `-DCONTRRACT_OPT=1` **normalised** with respect to `-DCONTRRACT_OPT=0`

Next, the compilation aspect of operation minimisation is studied. For this benchmark, the cost of compilation over the single expression evaluation scheme (when depth-first search is not perfomed) is studied. However, unlike the benchmarks conducted in subsection 4.2, the

↓ \→	Compilation Time			Memory Usage			Binary Size			Execution Time		
	GCC	Clang	ICC	GCC	Clang	ICC	GCC	Clang	ICC	GCC	Clang	ICC
7 Index	1.639	0.995	1.243	3.688	1.122	1.117	0.249	0.353	0.293	0.763	0.765	11.984
6 Index	3.247	1.444	1.487	12.618	1.304	1.269	0.322	0.432	0.338	1.458	25.055	25.361
5 Index	10.583	1.981	2.363	50.688	1.778	1.593	0.396	0.589	0.427	2.329	44.861	46.86
4 Index	-	3.236	4.61	-	3.13	2.307	-	0.9	0.607	-	66.444	67.764
3 Index	-	5.316	10.439	-	7.378	3.847	-	1.528	0.986	-	82.122	80.098
2 Index	-	12.54	22.455	-	21.7	7.185	-	2.811	1.721	-	73.247	73.02
1 Index	-	28.31	83.96	-	73.481	14.369	-	5.589	3.277	-	65.373	62.965

Table 3: Compilation aspects & run time performance of `-DCONTRRACT_OPT=2` **normalised** with respect to `-DCONTRRACT_OPT=0`

interest here is not in the runtime performance of the operation minimisation, hence, studying cache heirarchies and creation of intermediate temporaries are not pursued. Instead the two compilation aspect namely, compilation time and compiler's memory footprint is studied as a function of the Cartesian product (iteration space of nests) and reported in Table 4-Table 6 as the number of FLOPS saved/reduced, since these two parameters (iteration span and number of FLOPS saved) are correlated. All the benchmarks are run by resorting back to the default optimisation level i.e. `-DCONTRACT_OPT=0`, as operation minimisation is an orthogonal matter to loop transformation optimisations. Analogous to the benchmarks presented in subsection 4.2, a three tensor singleton $[\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{ijl}[\boldsymbol{C}]_{mnl}$, a four tensor singleton $[\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{ijl}[\boldsymbol{C}]_{mnl}[\boldsymbol{D}]_{no}$ and a five tensor singleton $[\boldsymbol{A}]_{ijk}[\boldsymbol{B}]_{ijl}[\boldsymbol{C}]_{mnl}[\boldsymbol{D}]_{no}[\boldsymbol{E}]_{p}$ are chosen and their sizes are successively increased. Table 4, Table 5 and Table 6 show the compilation time and memory usage operation

Saved FLOPs	Compilation Time			Memory Usage		
	GCC	Clang	ICC	GCC	Clang	ICC
44544	1.307	1.307	1.325	1.07	1.07	1.076
89088	1.331	1.255	1.32	1.063	1.249	1.075
178176	1.288	1.712	1.289	1.062	1.726	1.077
356352	1.29	1.331	1.385	1.074	1.063	1.078
712704	1.34	1.252	1.283	1.062	1.242	1.078
1425408	1.295	1.705	1.323	1.07	1.729	1.078
2850816	1.294	1.288	1.364	1.067	1.062	1.079

Table 4: Compilation cost of operation minimisation **normalised** with respect to single expression evalution for 3 tensor singleton

minimisation scheme over the single expression scheme for each of the aforementioned singleton. As can be observed from the results, the compilation time is not related to the sizes of the tensors but rather to the number of tensors (or operators) appearing in the whole tensor network. This is a fairly certain issue as the depth-first search recursive in nature[6]. In that, the compilation time and memory footprint of four and five tensor singleons are around 30% and 80-100% more than those of single expression evaluations. However, in terms of raw timings, the highest compilation time for operation minimisation has been 2.25 seconds (with ICC) which corresponds to a FLOP reduction of 41566208. The conclusion drawn from these results is that, a compile time depth-first search is fundamentally low-cost if the number of tensors in the network are small. At the cost increasing the compile time by merely a few seconds million to billions of runtime operations can be saved.

5. Applications & Real-world experimentation: Kernel-based numerical integration of nonlinear materials

In this section, numerical examples pertaining to the application of the current tensor contraction library in an embedded finite element framework are presented, specifically, in

[6]The cost of depth-first search through Fastor's meta-engine is at most $n!/2$, where n is the number of tensors appearing in the network.

Saved FLOPs	Compilation Time			Memory Usage		
	GCC	Clang	ICC	GCC	Clang	ICC
256512	1.255	1.281	1.78	1.27	1.271	1.294
513024	1.26	1.758	1.584	1.261	1.731	1.292
1026048	1.256	1.282	1.359	1.26	1.054	1.296
2052096	1.31	1.318	1.459	1.262	1.259	1.295
4104192	1.268	1.716	1.512	1.258	1.731	1.295
8208384	1.263	1.314	1.427	1.265	1.047	1.296
16416768	1.281	1.244	1.438	1.268	1.249	1.299

Table 5: Compilation cost of operation minimisation **normalised** with respect to single expression evalution for 4 tensor singleton

Saved FLOPs	Compilation Time			Memory Usage		
	GCC	Clang	ICC	GCC	Clang	ICC
649472	1.745	1.778	2.293	1.758	1.759	1.822
1298944	1.727	1.32	2.288	1.753	1.052	1.823
2597888	1.724	1.263	2.188	1.745	1.242	1.827
5195776	1.716	1.747	2.311	1.763	1.749	1.829
10391552	1.778	1.277	2.205	1.756	1.046	1.828
20783104	1.724	1.256	2.274	1.752	1.246	1.827
41566208	1.745	1.712	2.314	1.756	1.738	1.828

Table 6: Compilation cost of operation minimisation **normalised** with respect to single expression evalution for 5 tensor singleton

numerical integration of work-conjugates and Hessian of some polyconvex hyperelastic and multi-variable convex electroelastic energy functionals presented in section 2. The objective of these examples are to examine the speed-ups gained in numerical integration (or the so-called local assembly) of complex energy functionals using the framework's data parallelism, smart expression templates and additional domain specific features (benchmarked in the previous sections). In order to gain insights into each of the aformationed optimisation steps, the Fastor's implementation is benchmarked against three different individual implementations, namely

1. **Explicitly-vectorised Implementation** [*Variant 1*]: This implementation benefits from Fastor's explicit vectorisation but uses classical operator overloading. Comparison against this implementation will measure solely the benefit of operator chaining of expression templates for finite element local assembly procedures.
2. **Auto-vectorised Implementation** [*Variant 2*]: This implementation does not benefit from Fastor's explicit vectorisation and this is left to the compiler's auto-vectoriser. The implementation also uses classical operator overloading. Comparison against this implementation will measure the impact of explicit SIMD vectorisation and operator chaining for finite element local assembly procedures.
3. **Classical Implementation** [*Variant 3*]: This implementation does not benefit from Fastor's explicit vectorisation and operator chaining. Furthermore, unlike variants 2 and 3, this implementation uses a classical implementation of the tensor cross product. Comparison against this implementation will measure the impact of explicit SIMD vectorisation, operator chaining and optimised tensor cross product kernels for finite element local assembly procedures. Note that this implementation only holds for three-dimensional problems, as the tensor cross product is a three-dimensional operator.

Note that while the backend implementaions for each implementation differs, they are all similar in terms of API, functions' signatures, data structures used and the local assembly contraction loop nest (in fact, all implementations are fundamentally based on Listing 6). Furthermore, all variants are compiled with identical compiler flags. An attempt is made to keep the other implementations as close to a realistic implementation as possible, in that

the calls to functions such as determinant, inverse, transpose and cofactor used within the quadrature loop nest (as shown in Listing 6) are kept the same for all the implementations i.e. they are all call optimised in-built routines. This is true for most implementations, where such calls are dispatched to either vendor BLAS or optimised in-built subroutines. The only exception is that for the third variant of the implementation, the `cross` function in Listing 6 uses a classical implementation as its implementation is not available in say, BLAS. It is worth mentioning that, in the current tensor contraction framework, the optimised implementation of the tensor cross products involves complete manual loop unrolling, explicit AVX vectorisation, zero elimination and restructuring of the data for super-scalar execution. The technique of eliminating zeros from the computation is a rather common practice in generating domain specific kernels [11, 40].

To this effect, three energy functionals are chosen, one purely mechanical (the Mooney-Rivlin model) and two electromechanical, including the one presented in subsection 2.4, all listed in Table B.9 in Appendix B. The assessment is then to perform finite element analyses based on displacement-based formulation for mechanical problems and displacement-potential based formulation for electromechanical problems with high order triangles and tetrahedral elements and monitor the speed-ups achieved for local assembly. The quadrature loop nest is set up in the most classical fashion, in that it includes iteration over the quadrature points and test and trial spaces [11, 14, 15, 18, 41]. However, the last two loops (over test and trial spaces) are removed in favor of the abstraction provided by Fastor's `Tensor` class.

Consequently, in line with the theme of this framework, explicit tensorial operations involving computation of the following quantities are carried out within every quadrature point: a) the Jacobian of the isoparametric mapping ($\boldsymbol{\nabla}_{\boldsymbol{X}}\boldsymbol{\xi}$) (as described in [71] b) the material gradient of the displacements ($\boldsymbol{\nabla}_0\boldsymbol{u}$), c) the variables in the extended kinematic set $\mathcal{V}$, d) the set of work-conjugates and subsequently the first Piola-Kirchhoff stress tensor and the electric field vector and finally e) the Hessian of the internal energy.

It should be clear that, the utmost efficiency of the approach taken here for local assembly is not the objective of these benchmarks. The benchmarks rather showcase the usage of the framework in explicit finite element programming through seemingly hidden domain aware expressions, which plays a key role in kernel-based numerical integration shown in Listing 6.

Listing 6: The structure of the quadrature loop nest

```
for (auto g=0; g<ngauss; ++g) {
    // Compute Jacobian of isoparametric mapping
    auto ParentGradientX = matmul(GradBases,LagrangeElemCoords);
    // Compute material gradient
    auto MaterialGradient = matmul(inverse(ParentGradientX),GradBases);
    // Compute the deformation gradient tensor
    auto F = matmul(MaterialGradient,EulerElemCoords);
    // Compute the cofactor of deformation gradient tensor
    auto H = cofactor(F);
    // Compute the Jacobian of deformation gradient tensor
    auto J = determinant(F);
    // Compute work-conjugates
    // Sigma_F, Sigma_H, Sigma_J, Sigma_DO and Sigma_d
    // Compute the first Piola-Kirchhoff stress tensor
    auto P = Sigma_F + cross(Sigma_H,F) + Sigma_J*H;
    // Compute the electric field/displacement
    // E0, D0
    // Compute the Hessian components
    // WFF, WFH, WFJ, WFD0, WFd, WHH, WHJ,... Wdd
    // Compute the Hessian of the energy W
    // H_W ...
}
```

In the current setting, kernel-based computation is a consequence of expression templates combined with the C++11 `auto` keyword, in that, specific quantities within a quadrature loop nest can be lumped as a single expression and a single kernel can be launched for it. To illustrate this, consider the evaluation of the deformation gradient tensor `F` in Listing 6. Computing this quantity requires computation of `ParentGradientX` and `MaterialGradient` first. However, note that the automatic type deduction via `auto` does not force these quantities to bind to an object and as a result their computation is postponed and their automatic type is chained and carried over to the next line. Now, by the time the computation of `F` is requested, three `matmul` and one `inverse` functions are chained together, see Listing 6. A canonical and rather schematic representation of how the type of `F` is detected in Fastor can be represented as in Listing 7.

Listing 7: A single expression for computing the deformation gradient tensor

```
BinaryMatMulOp<BinaryMatMulOp<UnaryInvOp<BinaryMatMulOp<GradBases,LagrangeElemCoords> >,
    GradBases>,EulerElemCoords>
```

The evalutaion policy in Fastor, detects that an efficient implementation for this chained expression is available that does not require as many memory load and store operations. Hence, it statically dispatches the expression for `F` to a bespoke kernel. In particular, the evaluation of this kernel involves SIMD optimised matrix multiplications for on-cache tensors. It is worth mentioning that, chaining multiple operations of level 3 BLAS as shown in Listing 7, is a fundamentally rare feature for generic tensor algebra libraries. Fastor leverages from this by virtue of being domain specific. Similarly, for computing the first Piola-Kirchhoff stress tensor, Fastor detects the following expression, shown in Listing 8.

Listing 8: A single expression for computing the Piola-Kirchhoff stress tensor

```
BinaryAddOp<Sigma_F,BinaryAddOp<BinaryCrossOp<Sigma_H,F>, BinaryMulOp<Sigma_J,H>>>
```

Evaluation of this expression requires a single transparent loop within the quadrature loop nest, which also gives rise to a myriad of other optimisation possibilities other than vectorisation, as shown in Figure 4. The same concept is applied for the computation of other variables such as electric field and the Hessian.[7]

Based on our initial set up explained at the start of the section, synthetic finite element examples are manufactured based on two meshes, one triangular mesh and one tetrahedral mesh, respectively, as shown in Figure 11. Moreover, careful attention is paid to the assembly code generated by the compilers. However, for the purpose of brevity, only the results form the Intel compiler with `-O3 -xHost`, are presented here. All the benchmarks in this section are carried out with double precision floating point. For high order elements, nodal Lagrange basis functions with optimal nodal placements [60, 72] are chosen, to guarantee the stability and p-convergence property of the basis functions. These correspond to Fekete point nodal distribution for triangles and Warburton nodes for tetrahedra. Furthermore, the optimal quadrature scheme for triangles and tetrahedra presented in [73] is employed.

Table 7 and Table 8 show the speed-ups achieved using Fastor over the other implementations for the triangular mesh and the tetrahedral mesh, respectively. As can be observed from

[7] In fact, as a part of the smart expression template engine for a domain specific tensor contraction framework, it is possible to decesively change the evaluation policies of the expressions (as described in subsection 3.4) such that specific quantities of interest can be computed as a single kernel. This facilitates locality of reference which has a significant importance in SIMD and GPU computing.

p	Mooney-Rivlin Model		Electroelastic Model 1		Electroelastic Model 2	
	Explicit SIMD	Auto-Vectoriser	Explicit SIMD	Auto-Vectoriser	Explicit SIMD	Auto-Vectoriser
$p = 1$	1.32	1.598	1.734	1.135	2.168	1.661
$p = 2$	1.226	1.758	1.645	1.336	1.982	1.77
$p = 3$	1.177	1.924	1.497	1.528	1.856	1.912
$p = 4$	1.132	2.064	1.485	1.747	1.723	1.972
$p = 5$	1.145	2.095	1.366	1.667	1.595	1.928
$p = 6$	1.089	2.085	1.334	1.93	1.517	2.202

Table 7: Speed-ups achieved in numerical integration using Fastor over other implementations for the 2D triangular mesh

p	Mooney-Rivlin Model			Electroelastic Model 1			Electroelastic Model 2		
	Explicit SIMD	Auto-Vectoriser	Classic	Explicit SIMD	Auto-Vectoriser	Classic	Explicit SIMD	Auto-Vectoriser	Classic
$p = 1$	1.462	2.226	5.514	1.928	2.51	4.63	1.591	2.054	3.004
$p = 2$	1.964	3.52	6.711	1.957	3.358	5.181	1.7	1.978	3.208
$p = 3$	1.183	3.321	6.277	1.582	3.341	5.815	1.643	2.813	4.294
$p = 4$	1.506	4.89	7.557	1.401	3.674	5.367	1.505	3.021	4.26
$p = 5$	1.604	6.114	8.313	1.362	4.063	5.397	1.485	3.474	4.44
$p = 6$	2.009	5.96	7.382	1.247	4.171	5.034	1.267	3.214	3.971

Table 8: Speed-ups achieved in numerical integration using Fastor over other implementations for the 3D tetrahedral mesh

Table 7 (corresponding to the two-dimensional case with triangular mesh), for Mooney-Rivlin model, where the constitutive law is simpler and the number of operators to chain within the quadrature nest is small, most of the performance comes from SIMD vectorisation. However, for the two electroelastic models, where the constitutive law is significantly complex and the number of operators to chain are large, up to 85% of the performance of Fastor comes from operator chaining; see Figure 10(a,b,c) for a visual representation.

A counter-intuitive finding from Figure 10 is that, at high polynomial degrees, the effect of explicit vectorisation is more pronounced than operator chaining. However, by studying the quadrature nest shown Listing 6, one can observe that most of the tensors within the nest are small (of size $d \times d$ where the $d = 2, 3$ is the spatial dimension of the problem). In the absence of operator chaining and at low polynomial degrees, Fastor attempts to generate vectorised code for every operator individually by strictly aligning the operands pointers at `16B` or `32B` boundaries, depending on the vectorisation level. This is too strict of a requirement, that eventually forces the compiler to insert further paddings in order to avoid cache spills. This destroys the data locality and hence impacts the run time. However, as seen in Table 7, if the decision of vectorisation is left to the auto-vectoriser, the compiler tries to be much more conservative about vectorisation. Alternatively if the operators are chained and then vectorised as a single expression, the combined benefit of both paradigms (vectorisation and operator-chaining) can be harnessed. The conclusion drawn from this observation is that, operator chaining is essential for maintaining data locality in the quadrature nest.

At high polynomial degrees, most of the simulation time is consumed in computing the defromation gradient tensor `F`, which mainly involves the three `matmul` functions in Listing 6. Despite being a Fastor smart expression (as explained earlier), the cost of matrix-matrix multiplications is certainly going to dominate the computation time as the sizes of `GradBases`, `LagrangeElemCoords` and `EulerElemCoords` are much bigger compared to the rest of the variables within the quadrature loop. Operator chaining effect, while present is going to be a fraction of the cost of `matmul`s between larger tensors. Accelerating the local matrix-matrix multiplication kernels within the quadrature points is also studied in [40].

For the three-dimensinoal problems (tetrahedral mesh), similar performance traits can be observed as shown in Table 8 and Figure 10(d,e,f). However, here the effect of zero-elimination using the bespoke tensor cross product kernels comes into play. Note that, straight forward implementation of the tensor cross product is d^6 in computational complexity and d^4 in memory

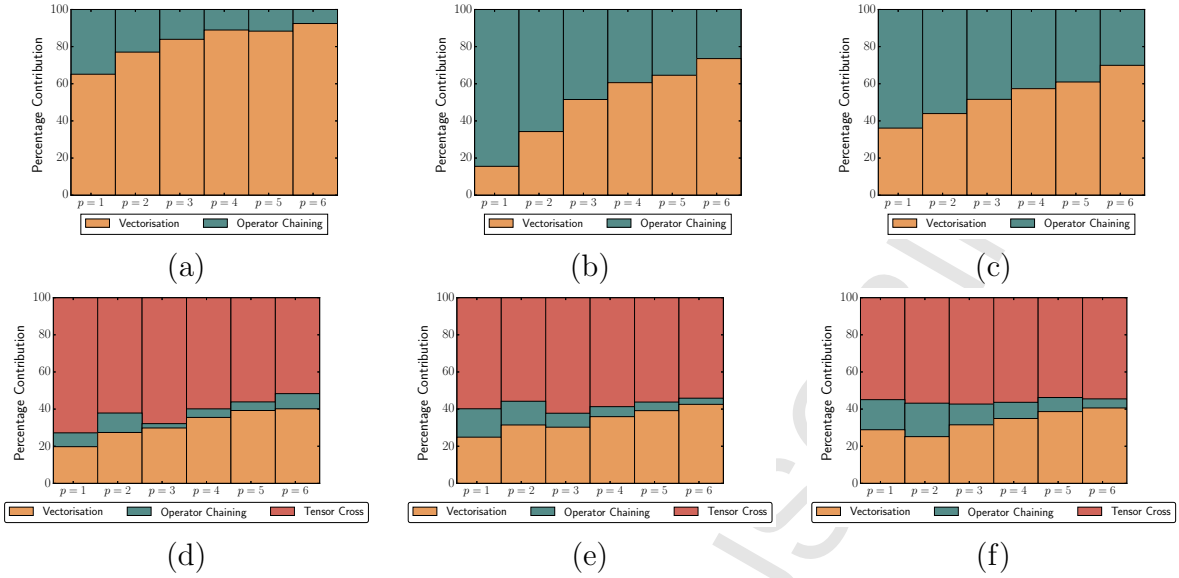


Figure 10: Contribution percentile of different optimisations for numerical integration using Fastor; (a), (b), (c) for triangular mesh and (d), (e), (f) for tetrahedral mesh.

access, implying that it is dimension dependent and independent of the polynomial degree. As a result, a rather constant speed-up is observed using the optimised tensor cross product kernels. It is worth mentioning that, while a different approach to performing numerical integration could be employed, the aforementioned results can be used as indicative numbers, of what is possible by relying on explicit tensor manipulations.

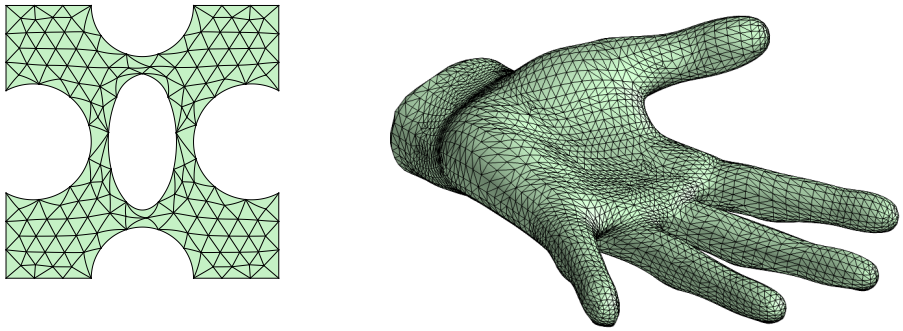


Figure 11: Meshes used for finite element benchmarks, a) A curved mechanical component [60] and b) Artificial hand used for simulating electrostriction [48].

6. Concluding remarks

A domain specific data parallel tensor contraction framework for numerical analysis of coupled and multi-physics applications is presented. The framework encompasses tensor contraction of isomorphic and nonisomorphic tensor networks by relying on explicit vectorisation using SIMD vector types. Furthermore, the in-built smart expression template engine performs compile time operation minimisation technique using mathematical transformation for named chained operators and depth-first constructive approach for un-named operators on tensor networks. The framework utilises heavy compile technologies perform aggressive loop optimisation,

which in certain cases can completely eliminate the run time memory I/O. Benchmark examples presented show optimal SIMD speed-ups for contraction of arbitrary order tensors on a recent Intel processor using three different compilers. Finally, finite element examples involving kernel-based numerical integration of complex convex multi-variable energy functional are carried out in two and three dimensions, where all the features of the current tensor contraction framework are utilised, in particular, the effect of operator chaining and launching vectorised kernels, is shown to be paramount. In this context, numerical examples presented, confirm significant speed-ups over the classical approaches.

Appendix A. Computational aspects of tensor contraction

In this section, some common computational terminologies used in the context of tensor contraction are defined.

Definition 1 *Tensor network: A complex network of tensors comprising of two or more tensors, multiplied and summed over a set of indices, for instance* $\mathcal{A}_{ijk}\mathcal{B}_{lmj} + \mathcal{C}_{ijkl}\mathcal{D}_{j}\mathcal{E}_{m} + ...$

Definition 2 *Singleton: A singleton or a single term tensor network is a single sub-expression of (Definition 1), for instance* $\mathcal{A}_{ijk}\mathcal{B}_{lmj}$.

Definition 3 *Isomorphic tensor product (outer product): Given a tensor pair* $\boldsymbol{\mathcal{A}}$ *and* $\boldsymbol{\mathcal{B}}$ *belonging to vector spaces* $\boldsymbol{\Xi}$ *and* $\boldsymbol{\aleph}$*, respectively, their product is said to be isomorphic, if and only if there are no contracting indices between the two, i.e. if isomorphism exists between the vector space of the product* $\boldsymbol{\Upsilon}$ *and the product of the vector spaces* $\boldsymbol{\Xi} \otimes \boldsymbol{\aleph}$.

Definition 4 *Nonisomorphic tensor product (tensor contraction): Given a tensor pair* $\boldsymbol{\mathcal{A}}$ *and* $\boldsymbol{\mathcal{B}}$ *belonging to vector spaces* $\boldsymbol{\Xi}$ *and* $\boldsymbol{\aleph}$*, respectively, their product is said to be nonisomorphic, if at least their is one common index between the two i.e. if no isomorphism exists between the vector space of the product* $\boldsymbol{\Upsilon}$ *and the product of the vector spaces* $\boldsymbol{\Xi} \otimes \boldsymbol{\aleph}$.

Definition 5 *Named operator: An operation performed on a tensor, a pair or a network is said to be named, if there is a specific name for the function signature, for instance,* `gemm, matmul, rotg, transpose, trace`.

Definition 6 *Un-named operator: An operation performed on a tensor, a pair or a network is said to be un-named, if it is expressed through indicial notation, for instance,* $\mathcal{A}_{iii}$ *is an un-named reduction operation.*

Definition 7 *Contraction loop nest: A variable number of nested* for *loop iterating over the space of tensor dimensions.*

Definition 8 *Fully vectorisable contraction loop nest: A loop nest is said to be fully vectorisable if and only if a) the span (iteration space) of the fastest changing index in the catersian product is a multiple of SIMD vector size and b) the index is not a contraction index.*

Definition 9 *Partially vectorisable contraction loop nest: A loop nest is said to be partially vectorisable if and only if a) the span (iteration space) of the fastest changing index in the catersian product is not a multiple of SIMD vector size but nevertheless greater than it and b) the index is not a contraction index.*

Definition 10 *Broadcast-vectorisable contraction loop nest: lerA loop nest is said to be broadcast-vectorisable if a) the span (iteration space) of the fastest changing index in the catersian product is a multiple of or greater than SIMD vector size b) the index is a contraction index. Double contraction is a special case of broadcast-vectorisable contraction loop nests.*

Definition 11 *Depth-first constructive search: A compile-time graph search to find the order in which pairs of tensors need be contracted so that the contraction over all tensor network incurs minimum floating point operations. The by-pair nature of tensor network evaluation leads to multiple intermediate temporaries which introduces a memory vs FLOP tradeoff.*

Appendix B. Material models used for finite element benchmarks

The internal energies of the materials used in section 5 are as follows

Material Model	Internal Energy
Mooney-Rivlin	$W_{mr} = \mu_1 II_{\boldsymbol{F}} + \mu_2 II_{\boldsymbol{H}} - 2(\mu_1 + 2\mu_2)\ln J + \frac{\kappa}{2}(J-1)^2$
Electroelastic Model 1	$W_{el,1} = \mu_1 II_{\boldsymbol{F}} + \mu_2 II_{\boldsymbol{H}} - 2(\mu_1 + 2\mu_2)\ln J + \frac{\kappa}{2}(J-1)^2 + \frac{1}{2\varepsilon_1} II_{\boldsymbol{D}_0} + \frac{1}{2\varepsilon_2 J} II_{\boldsymbol{d}}$
Electroelastic Model 2	$W_{el,2} = \mu_1 II_{\boldsymbol{F}} + \mu_2 II_{\boldsymbol{H}} - (2\mu_1 + 4\mu_2 + 12\mu_e)\ln J + \frac{\kappa}{2}(J-1)^2 + + \frac{1}{2\varepsilon_1} II_{\boldsymbol{D}_0} + \frac{1}{2\varepsilon_2 J} II_{\boldsymbol{d}} + \mu_e \left(II_{\boldsymbol{F}}^2 + \frac{2}{\mu_e \varepsilon_e} II_{\boldsymbol{F}} II_{\boldsymbol{d}} + \frac{2}{\mu_e^2 \varepsilon_e^2} II_{\boldsymbol{d}}^2 \right)$

Table B.9: Material models used for finite element benchmarks

Acknowledgement: The first author acknowledges the financial support received through The Erasmus Mundus SEED program. Both second and third authors acknowledge the financial support provided by the Sêr Cymru National Research Network for Advanced Engineering and Materials.

References

[1] J. Bonet, A. J. Gil, R. Ortigosa, A computational framework for polyconvex large strain elasticity, Computer Methods in Applied Mechanics and Engineering 283 (2015) 1061–1094.

[2] J. Bonet, A. J. Gil, R. Ortigosa, On a tensor cross product based formulation of large strain solid mechanics, International Journal of Solids and Structures 84 (2016) 49–63.

[3] L. Chi-Chung, P. Sadayappan, R. Wenger, On optimizing a class of multi-dimensional loops with reduction for parallel execution, Parallel Processing Letters 07 (1997) 157–168.

[4] J. H. Reif, Depth-first search is inherently sequential, Information Processing Letters 20 (1985) 229 – 234.

[5] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, R. Harrison, Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 237–248.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, 2009.

[7] R. N. C. Pfeifer, J. Haegeman, F. Verstraete, Faster identification of optimal contraction sequences for tensor networks, Phys. Rev. E 90 (2014) 033315.

[8] G. Evenbly, R. N. C. Pfeifer, Improving the efficiency of variational tensor network algorithms, Phys. Rev. B 89 (2014) 245118.

[9] A. Hartono, A. Sibiryakov, M. Nooijen, G. Baumgartner, D. E. Bernholdt, S. Hirata, C.-C. Lam, R. M. itzer, J. Ramanujam, P. Sadayappan, Automated Operation Minimization of Tensor Contraction Expressions in Electronic Structure Calculations, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 155–164.

[10] S. J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed., Prentice Hall, 2009.

[11] K. B. Ølgaard, G. N. Wells, Optimisations for quadrature representations of finite element tensors through automated code generation, ACM Transactions on Mathematical Software 37 (2010) 8:1–8:23.

[12] B. Jeremić, K. Runesson, S. Sture, Object-oriented approach to hyperelasticity, Engineering with Computers 15 (1999) 2–11.

[13] W. Landry, Implementing a high performance tensor library, Sci. Program. 11 (2003) 273–290.

[14] A. Logg, K.-A. Mardal, G. N. Wells (Eds.), Automated Solution of Differential Equations by the Finite Element Method, volume 84 of *Lecture Notes in Computational Science and Engineering*, Springer, 2012.

[15] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, G. N. Wells, Unified Form Language: A domain-specific language for weak formulations of partial differential equations, ACM Transactions on Mathematical Software 40 (2014) 9:1–9:37.

[16] R. C. Kirby, A. Logg, L. R. Scott, A. R. Terrel, Topological optimization of the evaluation of finite element matrices, SIAM Journal on Scientific Computing 28 (2006) 224–240.

[17] R. C. Kirby, M. Knepley, A. Logg, L. R. Scott, Optimizing the evaluation of finite element matrices, SIAM Journal on Scientific Computing 27 (2005) 741–758.

[18] A. Logg, G. N. Wells, DOLFIN: Automated finite element computing, ACM Transactions on Mathematical Software 37 (2010) 20:1–20:28.

[19] A. T. T. McRae, G.-T. Bercea, L. Mitchell, D. A. Ham, C. J. Cotter, Automated generation and symbolic manipulation of tensor product finite elements, SIAM Journal on Scientific Computing 38 (2016) S25–S47.

[20] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, P. H. J. Kelly, Firedrake: automating the finite element method by composing abstractions, To appear in ACM Transactions on Mathematical Software (2016).

[21] Prudhomme, Christophe, Chabannes, Vincent, Doyeux, Vincent, Ismail, Mourad, Samake, Abdoulaye, Pena, Goncalo, Feel++ : A computational framework for galerkin methods and advanced numerical methods, ESAIM: Proc. 38 (2012) 429–455.

[22] T. L. Veldhuizen, Computing in Object-Oriented Parallel Environments: Second International Symposium, ISCOPE 98 Santa Fe, NM, USA, December 8–11, 1998 Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 223–230.

[23] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, T. Veldhuizen, Generic Programming: International Seminar on Generic Programming Dagstuhl Castle, Germany, April 27–May 1, 1998 Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 25–39.

[24] B. Jeremi, S. Sture, Tensor objects in finite element programming, International Journal for Numerical Methods in Engineering 41 (1998) 113–126.

[25] A. Limachea, P. Rojas Fredini, A tensor library for scientific computing, in: Mecanica Computacional, volume XXVII, San Luis, Argentina, 2008, pp. 2907–2925.

[26] E. Epifanovsky, M. Wormit, T. Ku, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, A. I. Krylov, New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations, Journal of Computational Chemistry 34 (2013) 2293–2309.

[27] G. Guennebaud, B. Jacob, Eigen v3, http://eigen.tuxfamily.org, 2010.

[28] S. Hirata, Tensor Contraction Engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories, The Journal of Physical Chemistry A 107 (2003) 9887–9897.

[29] Q. Lu, X. Gao, S. Krishnamoorthy, G. Baumgartner, J. Ramanujam, P. Sadayappan, Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions, J. Parallel Distrib. Comput. 72 (2012) 338–352.

[30] C.-C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, P. Sadayappan, Memory-optimal evaluation of expression trees involving large objects, Computer Languages, Systems and Structures 37 (2011) 63 – 75.

[31] J. A. Calvin, E. F. Valeev, Task-based algorithm for matrix multiplication: A step towards block-sparse tensor computing (2015).

[32] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, J. Demmel, A massively parallel tensor contraction framework for coupled-cluster computations, Journal of Parallel and Distributed Computing 74 (2014) 3176 – 3190. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[33] E. Solomonik, T. Hoefler, Sparse Tensor Algebra as a Parallel Programming Model, ArXiv e-prints (2015).

[34] T. Veldhuizen, Expression templates, C++ Report 7 (1995) 26–31.

[35] K. Matsuzaki, K. Emoto, Implementation and Application of Functional Languages: 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 72–89.

[36] K. Iglberger, G. Hager, J. Treibig, U. Rde, Expression Templates Revisited: A performance analysis of current methodologies, SIAM Journal on Scientific Computing 34 (2012) C42–C69.

[37] Y. I. J. Progsch, A. Adelmann, A new vectorization technique for expression templates in C++ (2011).

[38] M. Kretz, V. Lindenstruth, Vc: A C++ library for explicit vectorization, Software: Practice and Experience 42 (2012) 1409–1430.

[39] F. Witherden, A. Farrington, P. Vincent, PyFR: An open source framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach, Computer Physics Communications 185 (2014) 3028 – 3040.

[40] B. D. Wozniak, F. D. Witherden, F. P. Russell, P. E. Vincent, P. H. Kelly, GiMMiK - Generating bespoke matrix multiplication kernels for accelerators: Application to high-order computational fluid dynamics, Computer Physics Communications 202 (2016) 12 – 22.

[41] F. Luporini, A. L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D. A. Ham, P. H. J. Kelly, Cross-loop optimization of arithmetic intensity for finite element local assembly, ACM Transactions on Architecture and Code Optimization 11 (2015) 57:1–57:25.

[42] L. Dagum, R. Menon, OpenMP: An industry standard API for shared-memory programming, Computational Science & Engineering, IEEE 5 (1998) 46–55.

[43] OpenMP Architecture Review Board, OpenMP application program interface version 4.0, 2013. URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[44] M. Frigo, P. Halpern, C. E. Leiserson, S. Lewin-Berlin, Reducers and other Cilk++ hyperobjects, in: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2009, ACM, New York, NY, USA, 2009, pp. 79–90.

[45] J. E. Stone, D. Gohara, G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, IEEE Des. Test 12 (2010) 66–73.

[46] A. Fog, C++ vector class library, 1.22 ed., 2016. URL: http://www.agner.org/optimize/vectorclass.pdf.

[47] A. J. Gil, R. Ortigosa, A new framework for large strain electromechanics based on convex multi-variable strain energies: Variational formulation and material characterisation, Computer Methods in Applied Mechanics and Engineering 302 (2016) 293 – 328.

[48] R. Ortigosa, A. J. Gil, A new framework for large strain electromechanics based on convex multi-variable strain energies: Finite element discretisation and computational implementation, Computer Methods in Applied Mechanics and Engineering 302 (2016) 329 – 360.

[49] R. Ortigosa, A. J. Gil, A new framework for large strain electromechanics based on convex multi-variable strain energies: Conservation laws, hyperbolicity and extension to electro-magneto-mechanics, Computer Methods in Applied Mechanics and Engineering 309 (2016) 202 – 242.

[50] R. Ortigosa, A. J. Gil, C. H. Lee, A computational framework for large strain nearly and truly incompressible electromechanics based on convex multi-variable strain energies, Computer Methods in Applied Mechanics and Engineering 310 (2016) 297 – 334.

[51] R. de Boer, Vektor- und Tensorrechnung für Ingenieure, Springer, 1982.

[52] A. L. Dorfmann, R. W. Ogden, Nonlinear Theory of Electroelastic and Magnetoelastic Interactions, Springer, Dordrecht, Heidelberg, London, New York, 2014.

[53] R. M. McMeeking, C. M. Landis, Electrostatic forces and stored energy for deformable dielectric materials, Journal of Applied Mechanics 72 (2005) 581–590.

[54] D. K. Vu, P. Steinmann, Nonlinear electro- and magneto-elastostatics: Material and spatial settings, International Journal of Solids and Structures 44 (2007) 7891–7905.

[55] D. K. Vu, P. Steinmann, G. Possart, Numerical modelling of non-linear electroelasticity, International Journal for Numerical Methods in Engineering 70 (2007) 685–704.

[56] R. Bustamante, D. A., R. W. Ogden, Nonlinear electroelastostatics: a variational framework, Zeitschrift fur angewandte Mathematik und Physik 60 (2009) 154–177.

[57] R. Poya, A. J. Gil, P. D. Ledger, A computational framework for the analysis of linear piezoelectric beams using *hp*-FEM, Computers and Structures 152 (2015) 155–172.

[58] R. Ortigosa, A. J. Gil, J. Bonet, C. Hesch, A computational framework for polyconvex large strain elasticity for geometrically exact beam theory, Computational Mechanics 57 (2016) 277–303.

[59] R. Ortigosa, A. J. Gil, A computational framework for incompressible electromechanics based on convex multi-variable strain energies for geometrically exact shell theory, Computer Methods in Applied Mechanics and Engineering In Print (2016) http://dx.doi.org/10.1016/j.cma.2016.12.034.

[60] R. Poya, R. Sevilla, A. J. Gil, A unified approach for a posteriori high-order curved mesh generation using solid mechanics, Computational Mechanics 58 (2016) 457–490.

[61] F. Cirak, J. C. Cummings, Generic programming techniques for parallelizing and extending procedural finite element programs, Engineering with Computers 24 (2008) 1–16.

[62] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley, 2001.

[63] M. Kretz, Data-parallel vector types & operations, ISO/IEC C++ Standards Committee Paper (2016).

[64] S. Meyers, Effective Modern C++, O'Reilly Media, 2014.

[65] F. G. Van Zee, R. A. van de Geijn, BLIS: A framework for rapidly instantiating BLAS functionality, ACM Transactions on Mathematical Software 41 (2015) 14:1–14:33.

[66] D. Levinthal, Performance Analysis Guide for Intel Core™ i7 Processor and Intel Xeon™ 5500 processors, 2009. URL: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[67] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, S. Tomov, High-performance tensor contractions for GPUs, Procedia Computer Science 80 (2016) 108 – 118. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[68] E. Acar, R. J. Harrison, F. Olken, O. Alter, M. Helal, L. Omberg, B. Bader, A. Kennedy, H. Park, Z. Bai, D. Kim, R. Plemmons, G. Beylkin, T. Kolda, S. Ragnarsson, L. Delathauwer, J. Langou, S. P. Ponnapalli, I. Dhillon, L. Lim, J. R. Ramanujam, C. Ding, M. Mahoney, J. Raynolds, L. Elden, C. Martin, P. Regalia, P. Drineas, M. Mohlenkamp, C. Faloutsos, J. Morton, B. Savas, S. Friedland, L. Mullin, C. V. Loan, Future directions in tensor-based computation and modeling, in: NSF Workshop Report, VA, USA, 2009.

[69] Z. Porkoláb, J. Mihalicza, N. Pataki, Á. Sipos, Analysis of profiling techniques for C++ template metaprograms, Ann. Univ. Sci. Budapest. Sect. Comput. 30 (2009) 97116.

[70] N. Pataki, Testing by C++ template metaprograms, Acta Univ. Sapientiae 2 (2010) 154–167.

[71] J. Bonet, A. J. Gil, R. D. Wood, Nonlinear Solid Mechanics for Finite Element Analysis: Statics, 3rd ed., Cambridge University Press, Cambridge, UK, 2016.

[72] F. D. Witherden, P. E. Vincent, An analysis of solution point coordinates for flux reconstruction schemes on triangular elements, Journal of Scientific Computing 61 (2014) 398–423.

[73] F. Witherden, P. Vincent, On the identification of symmetric quadrature rules for finite element methods, Computers & Mathematics with Applications 69 (2015) 1232 – 1241.