Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Building a Compiler for Special Matrix Multiplies

Author:
Mehedi Vin Paribartan

Supervisor:
Prof. Paul Kelly

Second Marker:
Prof. Kin Leung

May 2, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Matrix multiplication (MM) is a well studied operation due to demand for an efficient and performant implementation, arising from its heavy use in multiple areas of science and engineering. The are many libraries that conform to the Basic Linear Algebra Subprograms (BLAS) interface, which focus on how to optimally implement the BLAS routines on specific hardware architectures. The generic routines are not always optimal in cases where additional information is known before-hand. If the operator matrix is known to be sparse and relatively small (under 100 x 100), displayed in Figure 1.1, we can achieve a more efficient routine. This thesis' motivation originates from the block-by-panel case of matrix multiplication, that occurs in Flux Reconstruction implementations whose aims are to simulate fluid flows [15]. Due to the popularity of matrix multiplication, we believe advancements in generating optimal routines would help other areas of research as well.

1.1 Context

1.1.1 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) is a family of techniques that use numerical analysis to estimate the flow of fluids as well as the interaction between the fluids and surface areas. High-order (third and above) methods can potentially give more accurate results at the same computational cost as low-order methods. Traditionally, high-order methods were harder to implement and less robust, thus had a lower adoption in both industry and academia.
In 2007 H.T. Huynh [15] presented the Flux Reconstruction (FR) approach, a single mathematical framework that unifies multiple high-order schemes. The framework made it simple to create solutions for economical high-order CFD problems. FR has good element locality lending itself to be efficiently run on streaming architectures, such as GPUs and CPUs with Single Instruction Multiple Data (SIMD) instruction sets.
PyFR [9] is an open-source Python library that implements the FR approach to solve

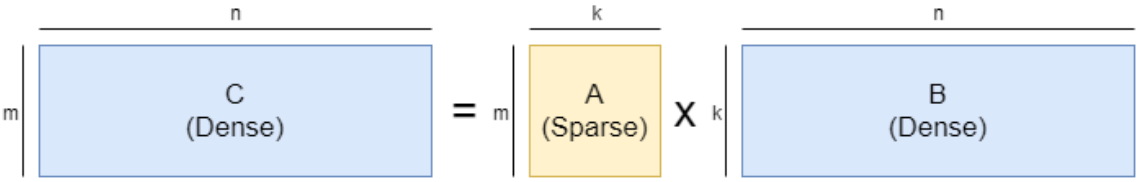


Figure 1.1: Visualisation of a block-by-panel type of matrix multiplication arising from Flux Reconstruction. The operator matrix is typically small, square and sparse. The operand and output matrices are typically fat and dense, implying $N >> M, K$ and $M \approx K$

advection-diffusion type problems and is designed to target a range of hardware architectures. In FR, partial differential equations are converted into block-by-panel style matrix multiplies where the operator matrix is sparse and heavily reused. In 2014, Wozniak [23] extended PyFR with GiMMiK to generate bespoke kernels to take advantage of the additional information known about the matrices, which provided performance speedups over generic BLAS libraries when running on GPUs. Section 2.2.3 covers GiMMiK in further detail. Other commercial libraries have more recently been released, such as cuSPARSE by NVIDIA [8] and MKL 'Inspector-Executor' routines by Intel [7], which also utilise additional information about the operator matrix. PyFR can target AVX-512 architectures via Intel's LIBXSMM [6], an open-source library for specialised sparse and dense matrix multiplies. In 2019 Price [18] presented VecReg, which also generated routines to run on AVX-512. Price compared VecReg against LIBXSMM with a suite of PyFR example matrix multiplies and obtained speedups in some cases.

1.1.2 Convolutional Neural Networks

A type of deep learning class is Convolutional Neural Networks (CNN), typically used in the image processing domain. In a CNN, there are convolutional layers, where the input is convolved with a (learned) filter. The 'lowering method' [5] is a commonly applied technique to carry out the convolutions, where they are transformed into matrix multiplications. It can be common for the operator matrix arising from the filter to be sparse and have a small amount of unique non-zeros. As the convolution is applied repeatedly to many inputs during training, improving the speed of this operation can be beneficial.
In 2017 Park et. al. [17] designed Direct Sparse Convolutions (DSC) as an alternative approach to the lowering method. DSC leads to greater arithmetic intensity by reducing the amount of repeated information stored from the input. DSC can be considered as a 'virtual sparse-matrix-dense-matrix multiplication' [17] and so would benefit from more performant sparse-dense matrix multiplication routines.
Whilst not a direct motivator for this thesis, CNNs and other machine learning areas are an example of other scientific applications that could benefit from the work in this thesis.

1.2 Objectives

The current limitation with LIBXSMM's sparse matrix multiply register-based routine, is that it supports up to 31 unique constants in the operator matrix. If there are more, LIBXSMM defaults to a less specialised routine. Price found speedups over LIBXSMM by being able to support more unique constants by packing them within the vector registers [18]. The first objective of this thesis will be to explore how we can adapt LIBXSMM to support more unique constants via packing for AVX-512 based CPUs.
If the number of unique constants in the operator matrix is less than the total number of slots within the vector register file, then packing can result in free, unused registers. Price used these free registers for column-based Common Sub-expression Elimination (CSE), but noted that were many other CSE opportunities. As well exploring how to implement these CSE techniques within LIBXSMM's sparse routine, we will attempt to create an algorithm that can consider the various CSE options and then select the most cost effective combination of options. This thesis will primarily target AVX-512 on the Intel Skylake-X architecture. The proposed techniques in conjunction with the register packing could be adapted for libraries other than LIBXSMM, to target other computer architectures.
In Sections 2 and 3, we cover some background information and a range of prior and related work. Section 4 outlines the testing environment, the test-suite and covers the methodology used for evaluation.

Chapter 2

Background

In this chapter we provide a brief overview of CFD and Flux Reconstruction. Later, in Section 2.2 the Python library PyFR is covered in more depth, including the characteristics of the operator matrices it uses as well as its approaches to heterogeneous computing. Lastly, we cover the Intel AVX-512 SIMD architecture and how the open source library LIBXSMM targets this architecture for optimised matrix multiplication.

2.1 Flux Reconstruction Overview

In CFD there are three basic, physical laws that state that mass, momentum and energy are conserved within a closed system. The Navier-Stokes equations are commonly used to describe the conservation of momentum in a fluid flow. The need to conserve other physical properties leads to continuity equations being solved in tandem. There are variants of the equation, such as if the flow is compressible or not. Regardless of the variation, they are given as a Partial Differential Equation. Analytical solutions, if possible in theory, are often too complex and so, computationally expensive to solve, so numerical methods are used to obtain estimated solutions.
In the numerical solutions, time and space are discretized. Three of the most popular spatial discretizations are; Finite Volume (FD) 'where the governing system is discretized onto a structured grid of points', Finite Difference (FV) 'where the domain is decomposed into cells and an integral form of the problem is solved within each cell', Finite Element (FE) 'where the domain is decomposed into elements inside of which sits a polynomial that is required to satisfy a variational form of the governing system' [22]. These methods are often implemented with first-order or second-order accuracy. The accuracy order determines how the error in the solution will respond to a change in the resolution of the grid [22]. It is possible to implement the schemes with higher-orders, but the computational cost increases rapidly. Spectral Difference (SD) is more recent class of high-order methods [21], which involve decomposing within the frequency space. However, they have issues with geometrical flexibility.
Flux Reconstruction provides the superior accuracy of high-order spectral or finite difference methods with the geometrical flexibility of low-order finite volume or finite element schemes [9]. Detailed derivations of how the FR framework obtains numerical solutions to the governing equations, via a seven-stage process, are provided by Castonguay et. al. in [4] and Huynh in [15]. The remainder of this thesis should be accessible without having read the additional material.

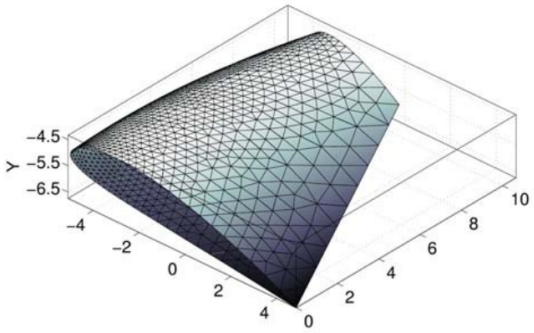


Figure 2.1: NACA 4415 wing represented via a 3-d unstructured triangular surface mesh [3]

2.2 PyFR

2.2.1 Overview

PyFR is an open-source Python library, first presented by Witherden et al. [9] that implements the FR approach to solve advection-diffusion type problems and is designed to target a range of hardware architectures [10]. PyFR is a high-order accurate solver. Traditionally, commercial solvers have been low-order accurate and can provide stable results relatively quickly. The downfall with low-order solvers is that they perform poorly when a higher accuracy is desired. PyFR satiates an increasing desire from both industry and academia for a higher accuracy solver to CFD problems, by implementing the FR scheme. FR has good element locality and so PyFR is able to run the scheme efficiently on modern streaming architectures. Whilst most of the scheme is implemented in Python, PyFR uses external libraries to accelerate the matrix multiplication part of FR. PyFR uses different libraries when targeting different hardware [10].
In FR, partial differential equations can be converted into block-by-panel style matrix multiplies where the operator matrix $\mathbf{A}$ can be sparse, and is heavily reused. The resulting matrix is accumulated with a scalar β (usually either 0 or 1 - i.e. added or not).

$$\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C} \tag{2.1}$$

This gives an opportunity to generate bespoke kernels by analysing the operator matrix. The kernel is then used on multiple, fat matrices ($\mathbf{B}$) over the FR process.
In an unstructured grid, the Euclidean space is tessellated using a simple type of shape, where each instance has an irregular pattern. The collection of the shapes in the space is often called the mesh. Figure 2.1 shows an example of an unstructured grid using triangles to represent the wing of a plane, produced by Becker et. al. [3].

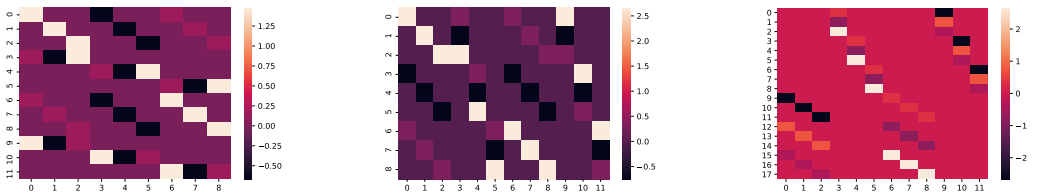
2.2.2 Operator Matrices

PyFR supports 2D and 3D grids and multiple types of shapes. The operator matrices that are repeatedly used originate from a process of storing *flux* and *solution* points into a file. This is an arbitrary process and could be changed to lead to different patterns in the matrices, but the process used is static within PyFR.
Throughout this thesis, there will be 5 operator matrices for a given combination of shape, quadrature and order of solution. Specifically, each set will have an **m0**, **m3**, **m6**, **m132** and **m460**. The characteristics of the operator matrices highly depend on the shape used, as well as the quadrature (numerical integration) method and also the order of the solution. Generally, as the order of the solution increases, so do the dimensions of the

operator matrix.
The following sections showcase some properties of example operator matrices from PyFR, (Appendix A provides more detail on every operator matrix example). The matrices are directly plotted as heatmaps, where each cell represents an element of the matrix and the row and column numbers are labelled on the axes. Note: the operator matrices are not a representation of the meshes, but represent operations carried out in FR.

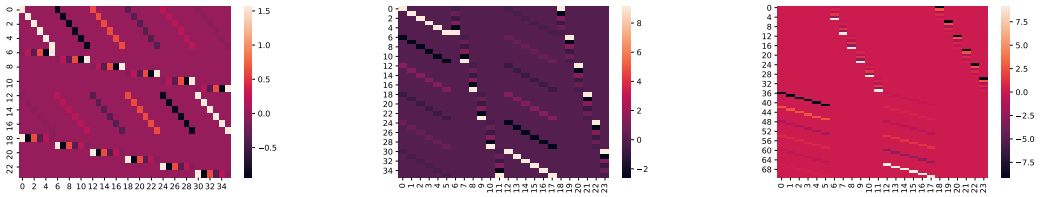
Quadrilaterals

Figures 2.2 and 2.3 show the operator matrices for Quadrilateral shapes using the Gauss-Legendre quadrature. The operator matrices increase in size as the order of the solution increases. Large sections of pink/purple are sections where the elements are all 0, highlighting that the the matrices are sparse. The pattern of the respective operator matrix for a given FR step (**m**_), holds as the order of the solution varies. Blocks of patterns can be seen better at the higher-orders, like in Figure 2.3 where interleaved diagonals span the matrix. The patterns arise from the storage layout of the points in memory and the neighbouring point being considered. For quadrilaterals, two sets of neighbours are considered, those within a stride of 1 and those within a stride of n. The stride of 1 leads to the sparser sections - for example the left half of Figure 2.4a.
Figure 2.4 shows the **m132** operator matrix for Quadrilateral fourth-order with slightly differing quadratures being used. The different stride lengths and their respective sections can be clearly seen. Tables A.1 and A.2 in Appendix A show that the dimensions stay the same, but the density and number of unique non-zeros varies when the quadrature used was varied. Figure 2.5 shows another example of how the operator matrix grows as the order increases. The top half of the matrices have a 'blocky' diagonal, associated with the neighbours within a stride of 1. The bottom half has a different diagonal pattern, associated with the neighbours within a stride of n.



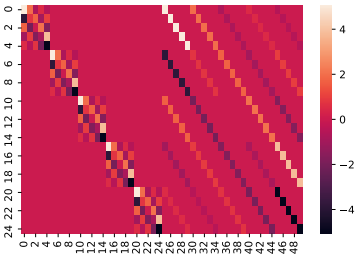
(a) **m0** (b) **m3** (c) **m6**

Figure 2.2: Quadrilateral Gauss-Legendre Second-Order

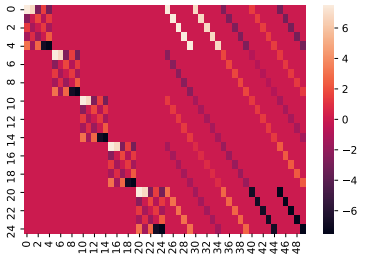


(a) **m0** (b) **m3** (c) **m6**

Figure 2.3: Quadrilateral Gauss-Legendre Fifth-Order

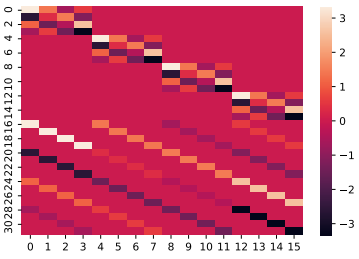


(a) **m132** Gauss-Legendre

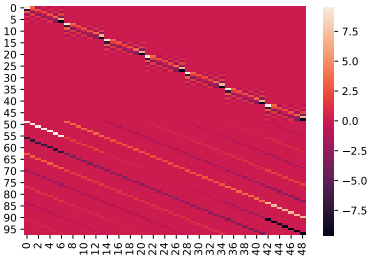


(b) **m132** Gauss-Legendre-Lobatto

Figure 2.4: Quadrilateral Fourth-Order different quadratures



(a) **m460** Third-Order

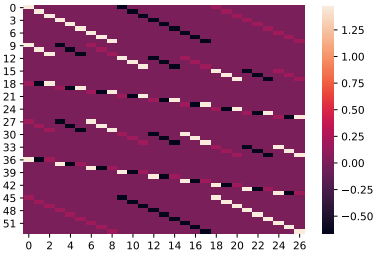


(b) **m460** Sixth-Order

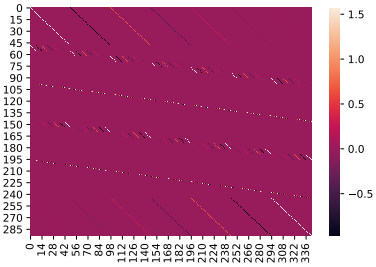
Figure 2.5: Quadrilateral Third-Order vs Sixth Order

Hexahedra

Hexahedra meshes lead to even more sparse operator matrices as points within stride of n^2 in memory must also be considered, due to the increase in dimensionality over quadrilaterals. This results in the blocks connecting to points within a stride of 1 and n becoming sparser as the dimensions of the operator matrix increase, but those patterns for the blocks remain the same. Figure 2.6 illustrates how the size of the operator matrix increases greatly as the order of the solution increased from second to sixth,, leading to a sparser matrix. The sections for strides of 1, n and n^2 are distinctly shown, from left to right respectively, in both matrices in Figure 2.7.



(a) **m0** Second-Order



(b) **m0** Sixth-Order

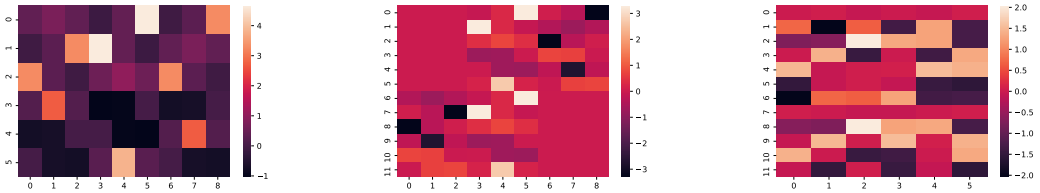
Figure 2.6: Hexahedra Second-Order vs Sixth Order

(a) **m132** Second-Order (b) **m132** Fourth-Order

Figure 2.7: Hexahedra Second-Order vs Fourth Order

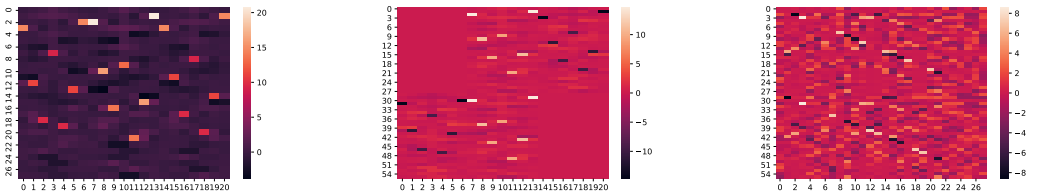
Triangles

Unlike the shapes above, triangles lead to dense operator matrices due to how the points are stored in memory. Figures 2.8 and 2.9 show the operator matrices for a triangular mesh using the Williams-Shunn quadrature with second-order and sixth-order respectively. Again, as the order increases, the size of the matrices increases. However, the size of the matrices at sixth-order are relatively very small, compared to quadrilaterals and hexahedra. The patterns, whilst not very sparse, still hold for the respective operators as the order increases. Figures 2.8b and 2.9b show that operator **m6** has a density of just above 0.5, whilst the other operators are shown to be more dense.



(a) **m3** (b) **m6** (c) **m460**

Figure 2.8: Triangles Williams-Shunn Second-Order



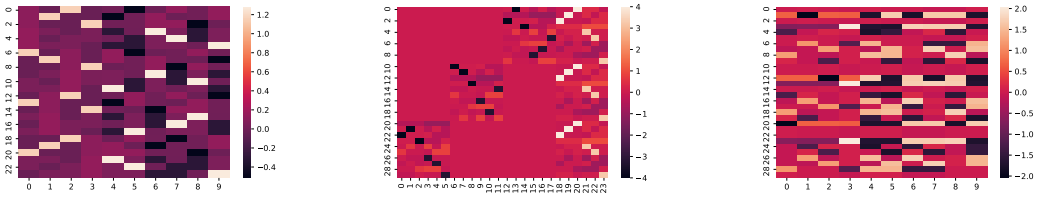
(a) **m3** (b) **m6** (c) **m460**

Figure 2.9: Triangles Williams-Shunn Sixth-Order

Tetrahedra

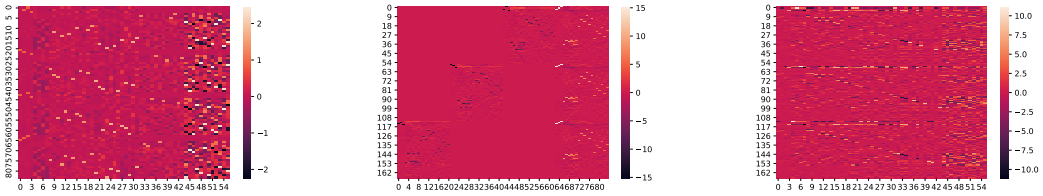
Figures 2.10 and 2.11 show some of the operator matrices for Tetrahedra meshes using the Shunn-Ham quadrature. The operator **m6** for both second-order and fifth-order show distinct sparse regions, one for each different stride length. The other operators are shown to be denser than **m6**. The operator matrix sizes increase at a greater rate compared to

triangular meshes, as the order increases, but are still small compared to the operators for the other 3D shape, hexahedra.
In summary, the operator matrices are denser but smaller for triangles and tetrahedra, compared to sparser and potentially much larger matrices for quadrilaterals and hexahedra. The matrices for triangles and tetrahedra also have more number of unique non-zeros at higher orders, which is detailed in Appendix A.



(a) **m0** (b) **m6** (c) **m460**

Figure 2.10: Tetrahedra Shunn-Ham Second-Order



(a) **m0** (b) **m6** (c) **m460**

Figure 2.11: Tetrahedra Shunn-Ham Fifth-Order

2.2.3 GiMMiK

In 2014, Wozniak [23] extended PyFR with GiMMiK to generate bespoke kernels to take advantage of the additional information known about the matrices, which provided performance speedups over generic libraries when running on GPUs. The main optimisations made were to fully unroll the loops, allowing multiplications by 0 to be removed, and to embed the constants of the operator matrix in the GPU kernel code. Roth [20] and Park [16] extended GiMMiK to generate kernels targeting CPU hardware.
In 2016, GiMMiK v2.0 was released. The bespoke kernels targeting CPUs used OpenMP SIMD pragmas to take advantage of SIMD ISA extensions on CPUs. The use of the OpenMP library gives less control to GiMMiK over the eventual assembly code, but provides a simple kernel code. The compiler is then required to support the pragma and then generate the final kernel assembly code.

2.3 Single Instruction Multiple Data Architectures

Single Instruction Multiple Data (SIMD) architectures operate on multiple sets of data with a single instruction. SIMD differs from vector-processing by operating on all elements of the vector simultaneously, as opposed to operating on the vector elements in a pipelined fashion. This means that multiple results are obtained from a single instruction operating identically, on different source elements. Figure 2.12 illustrates how a parallel addition is performed with SIMD. Each element is stored in a 'lane' of the vector and the right most

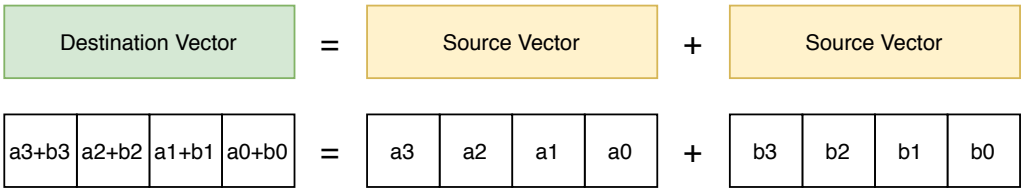


Figure 2.12: SIMD - Single Addition Instruction

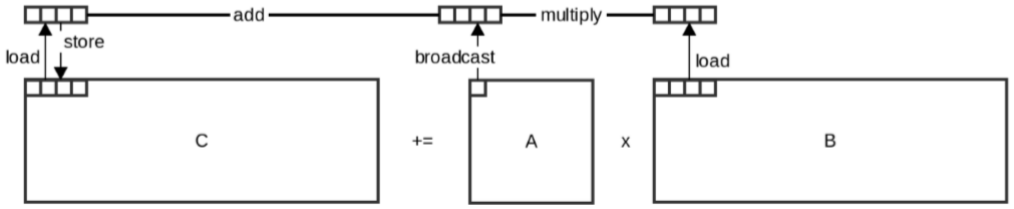


Figure 2.13: Roth illustrated the vectorisation of the matrix multiply routine[20]

lane is numbered lane zero. The vectors in Figure 2.12 have 4 lanes and the left most lane in each vector is the third lane of that vector. The operation takes values from the same lane in each source vector, and places the result of the addition into the respective lane in the destination vector.

Historically SIMD was used for massively-parallel supercomputers. Modern SIMD is now more commonly found as Instruction Set Architecture (ISA) extensions. The Advanced Vector Extension (AVX) class of extensions have been heavily adopted among x86 class CPUs and Arm also has the Neon and Scalable Vector Extension (SVE) extensions. SVE is unique in that it does not specify the vector width, only a range of valid widths that the CPU vendor can implement.

2.3.1 AVX-512: A Single Instruction Multiple Data ISA

Intel AVX-512 is a SIMD ISA extension for x86 CPUs, and is a evolution of its predecessor AVX-2, which it is backwards compatible with. AVX-512 registers are 512-bit wide, double that of AVX-2. The registers can therefore store up to 8 double precision (DP) or 16 single precision (SP) floating point numbers. In the case of DP, each register has 8 vector lanes. AVX-512 also support signed and unsigned integers of differing lengths. The ISA provides the ability to perform arithmetic operations, bit manipulation, compression, shuffling and more between vectors.

The Intel microarchitecture, Skylake Server Configuration, has a sub-memory system that supports 2x64B loads and 1x64B stores in each cycle. This allows an entire 512-bit (64B) register to be loaded or stored during each cycle, using only one memory port. AVX-512 is included in Skylake-SP server CPUs; the improved sub-memory system provides the memory bandwidth required by the doubling of vector width.

The main instructions used during a matrix multiply are the Fused-Multiply-Add (FMA) and memory loads/stores. The FMA instruction multiplies two source values and adds them to the destination register. AVX-512 has a single instruction that can FMA 8 DP values. In Skylake-SP, hardware support provides this instruction with the same cycle latency as an 8-wide DP vector addition.

2.3.2 Sparse MM: Vectorisation

As shown by Roth [20] in Figure 2.13, the vectorised matrix multiply is performed by the following procedure:

- For each row:
  - Loading a stride of C equal to the width of the vector register. (only if adding αAB to βC where $\beta \neq 0$)
  - For each element in A on this row, broadcast the element into a vector register and multiply with the corresponding stride from B.
  - Store the accumulated value for the C stride.
- Repeat this for all 'columns' of strides.

With AVX-512 and DP, the strides of B and C are 8 values wide. A really nice observation arises from this which is that the strides fit exactly into a single 64 byte cache-line. Since Skylake-SP supports 64 bytes loads and stores per cycle, each B-stride can be loaded in each cycle, even when storing the C-stride. Additionally, a column of B-strides takes up $\frac{8*K*64}{8}$ bytes for DP with dimension K. So for A with $K \leq 100$, a column of B-strides takes a max of 6.25 KiB, which would fit within the L1 data cache of a Skylake-SP core. This means repeated access to B-strides for different rows should be reading from L1-cache after the first access for a previous C-stride calculation.
A couple of trivial optimisations arise from the above routine when A is sparse:

- If the element in A is 0, skip the load of the corresponding B stride and continue to the next element of A.
- After eliminating the the above, fully unroll the loop to potentially improve performance.

2.4 LIBXSMM

2.4.1 The Library

LIBXSMM is a library for specialized dense and sparse matrix multiplies as well as other related deep learning operations [6]. The library focuses on small matrix multiplies, which is approximated by problems with dimensions $MNK : (MNK)^{1/3} \leq 64$ [6]. The library is designed to target various Intel architectures such as Intel SSE, AVX, AV2 and AVX-512. The library acts as a Just-In-Time (JIT) compiler, by generating code after analysing values discovered during run-time. The resulting byte-code is then placed into an executable buffer in memory. LIBXSMM is designed to be both compiler agnostic and threading-runtime agnostic. This means the same performance should be expected regardless of the compiler or threading library utilised by the user application.
Each specialized operation that LIBXSMM supports has its own dedicated function, that uses LIBXSMM's instruction API to 'write' assembly instructions into the executable buffer. LIBXSMM then casts this buffer to a suitable function pointer, allowing the user application to call the generated code. This process is visualized by Figure 2.14.

2.4.2 Use within PyFR

PyFR can optionally incorporate LIBXSMM as a matrix multiplication implementation whilst using the OpenMP as its threading runtime. This LIBXSMM is not required to provide a threading runtime, and is threading runtime agnostic. As well as running on CPU only clusters, it can use the library when heterogeneously computing [10]. In contrast to GiMMiK 2.0 where the compiler writes the resulting kernel, LIBXSMM directly writes every line of assembly for the bespoke kernel, giving a finer grain of control, with the additional complexity that comes with this control.

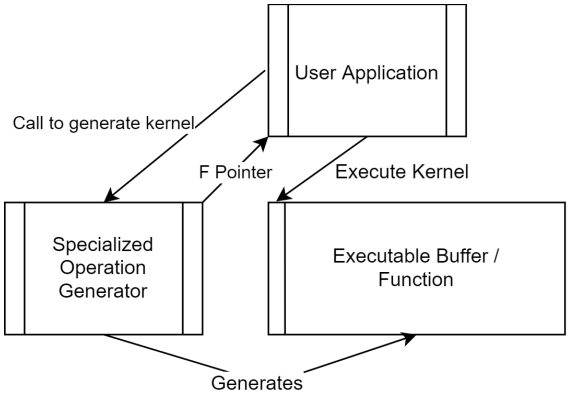


Figure 2.14: High-level view of Just-In-Time code generation in LIBXSMM

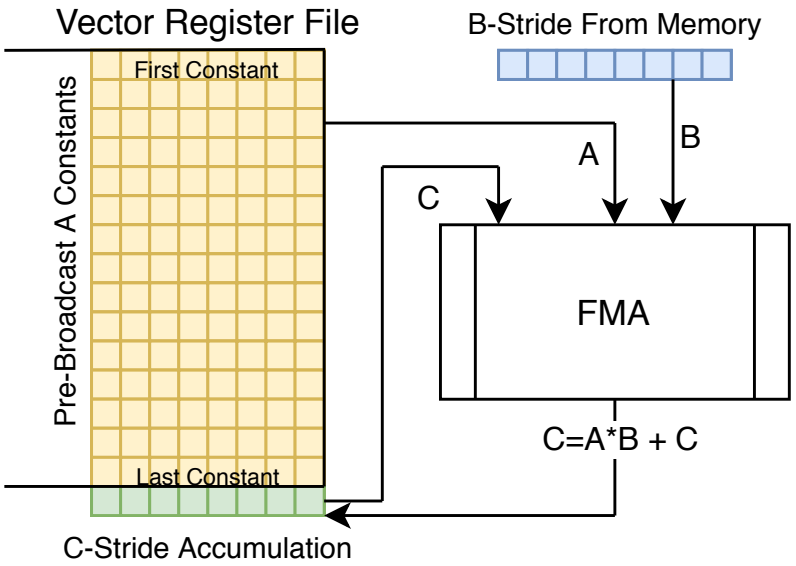


Figure 2.15: Pre-broadcasting constants from the operator matrix for use with the FMA operation - DP with AVX-512

2.4.3 Sparse MM: Storing A in the register file

LIBXSMM has a specific generator for when A has less than or equal to 31 unique non-zeros. As part of the JIT process, the number of unique non-zeros is counted. If above 31, then a slower alternative fallback strategy is used. As described in section 2.3.2, the elements of A have to be broadcast to fill a vector. If one vector register is used to accumulate the C-stride, then there are 31 free vector registers to store constant from A. The FMA operations can use memory locations as source operands as well as register sources. This means B-strides do not have to be stored in registers.
This generator pre-broadcasts each unique element of A into separate vector registers. During the FMA, the the broadcasted element of A is obtained from the corresponding register, as shown in Figure 2.15. This removes every load instruction for the operator matrix, leading to performance gains. The routine follows the strategy outlined in section 2.3.2.

Chapter 3

Related Work

This chapter will explore the work previously done to speedup PyFR's execution on CPUs and will also analyse CSE techniques from multiple research areas that have a common goal of flop reduction. The main purpose of this chapter is to review directly related work and to cover possible approaches to CSE that could be used in our implementation.

3.1 VecReg

Overview

In 2019, Price presented VecReg [18], a Just-In-Time (JIT) code generator targeting AVX-512 hardware to perform sparse matrix multiplication. Price used the Mako templating library to generate C code that makes use of AVX-512 intrinsics. The C code would then be compiled using Intel's compiler, ICC.

The main difference between VecReg and LIBXSMM's sparse register routine is how the constants from the operator matrix are stored in the register file. As mentioned in chapter 2, LIBXSMM pre-broadcasts each of the unique constants into a vector register. VecReg took a different approach, by compactly packing up to 8 (double precision) constants into a 512-bit array, as displayed in Figure 3.1. VecReg followed the same higher-level routine of matrix multiplication that LIBXSMM uses, and so still required the constants to be repeated across an entire, single vector register. During run-time, VecReg kernels broadcast the constants into a vector register, by extracting them from the compact format. VecReg relies on ICC to perform this broadcasting.

Evaluation

The compact storage of constants lead to some spare registers that could be used for CSE to reduce the flop count of the kernel. Price found up to 1.6x speedups over LIBXSMM, which was attributed to being able to perform CSE, and in general found speedups over LIBXSMM for operator matrices with more than 31 unique constants (when LIBXSMM defaults to a less specialised sparse kernel).

Price notes that LIBXSMM was loading C when not required, which has been corrected in a newer version of the library. Price continues his evaluation when A is dense and compares against LIBXSMM, to see how the ICC, therefore how VecReg, handles operator matrices with more than the storage limit of 240 unique constants (Price approximated 256, but one register is always required for storing the broadcast version of a constant and another to store a cumulative sum for matrix C).

Although Price does not explicitly state the following, we believe the assumption that the run-time broadcasting would not be on the critical-path was essential to performance not

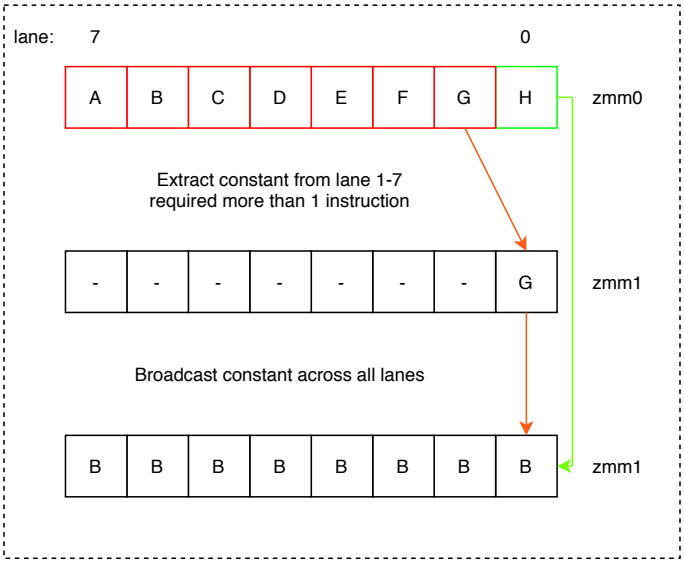


Figure 3.1: VecReg run-time constant unpacking/broadcasting from lane 0 is 'cheaper' in terms of instructions required than unpacking from the other lanes in the register [18]

degrading for operator matrices that do fit within LIBXSMM sparse A unique constant limit.

Investigating VecReg in practice

There does not exist a single instruction in AVX-512 that can broadcast any vector lane that isn't lane zero, from a source register, across the entirety of a destination register. If the constant is stored in lane zero, then ICC can use the 'VBROADCAST' instruction to achieve the desired broadcast with one instruction, which is represented by the green step in Figure 3.1. However, for other lanes, Price's assumption was that ICC would generate code that is longer than a single assembly instruction to broadcast the constant, represented by the red step in Figure 3.1.
A couple of sample kernels generated by VecReg are provided by Price. We investigated the assembly output of ICC, whilst using the same compiler options as Price (importantly, '-O3' used). ICC did not store any of the 512-bit arrays directly into the AVX-512 registers. Instead, packed values of A were loaded from memory into lane zero of a register. This was then broadcast using 'VBROADCAST'. By not utilising the register file to store A, the ICC compiler misses out on potential performance improvements. This investigation reveals VecReg does not implement the theoretical method proposed by Price.
Figure 3.2 shows the speedups of VecReg over LIBXSMM versus the number of unique non-zeros in A. The points for $\beta = 0$ should be ignored due to the issue in LIBXSMM previously mentioned being fixed. For $\beta = 1$ and the number of unique constants ≤ 31, most of the data points show that VecReg was noticeably slower. This is explained by VecReg/ICC kernels loading A from memory (most likely L1 cache). Therefore results of VecReg cannot be used to say that run-time broadcasting does not add to the critical-path, as we believe the performance penalties arise from loading A from memory.

Key Takeaways

We believe any run-time broadcasting operations should be tuned to use minimal instructions to avoid reaching the critical-path. To achieve this, alternative storage layouts would have to be explored as no single-instruction supports *any-source-lane* broadcasting. It should be noted that although there are a couple of issues with Price's work, the idea

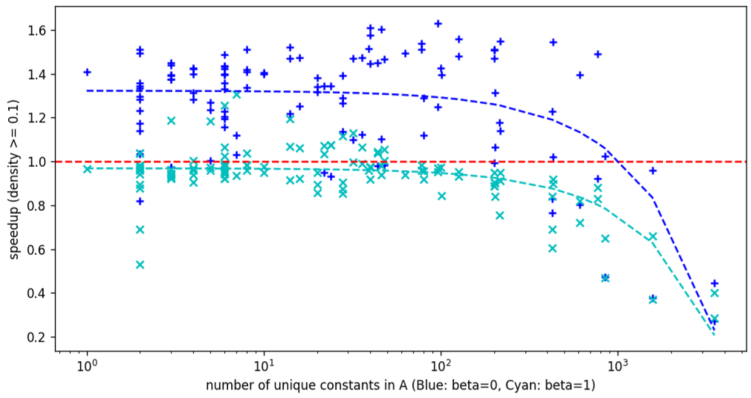


Figure 3.2: VecReg Speedup vs unique constants in A (density of A ≥ 0.1) [18]

of packing constants seems to be an excellent approach to being able to support more operator matrices and to allow for CSE.

3.2 Topological Optimization of the Evaluation of Finite Element Matrices

Kirby et al. [19] presented a framework in 2006 that aimed to reduce flop count over the evaluation of finite element matrices. The matrix being evaluated is known as the 'stiffness' matrix, which represents the system of linear equations to be solved in order to obtain an approximation for the original differential equation.
Kirby et al. make the abstraction to only consider the vector products when multiplying a R^{nm} matrix with a vector in R^m in the the optimization problem. The objective was to minimize the number of multiply-add pairs, or FMA operations. The total number of FMAs is bounded by $n * m$ when calculating

$$\left[(y^i)^t g\right]_{i=1}^{n} \quad y, g \in R^m \tag{3.1}$$

A discrete distance was calculated between $y, z \in R^m$, and if they are close, then the product of $y^t g$ should be easy to compute once $z^t g$ is known.
Kirby et al. formed a graph where a node represents a vector product $y^t g$. This node $i \in m$ was connected to every other node $j \in m, i \neq j$. The edges were weighted by how close, by the chosen distance metric, y^i was to y^j. A lower distance represented a lower amount of FMAs that were required to go from the vector product of the node i to the vector product of node j. Kirby et al. further expand on the distance metric, which they define as a complexity-reducing relation [19]. An example metric that satisfies the formal definition is the Hamming distance.
By calculating a directed minimum spanning tree (MST), and calculating the products in the traversal order, the minimum number of FMAs should have been used to calculate all the vector products.
As noted by Kirby et al. there were a couple of downfalls with the approach. The true optimal number of FMAs was not guaranteed to be found, as the complexity-reducing relation only considered two vectors at a time. By considering more than two, it is possible to find routines that would use even fever FMAs.
After removing multiplies by zero, the additional optimizations only provided a 'modest' speedup [19]. This can be attributed to the routine now being more memory bounded. A modification to the modelling of the cost, to include memory access could have potentially improved the speedup provided by the additional optimizations.

By forming a fully connected graph, a trivial algorithm to find the MST would have quadratic complexity. Kirby et al. propose various methods to reduce the search space, including only considering edges that have a minimum FMA reduction payoff.
We can utilize the notion of using a weighted-graph to decide which CSE optimizations decisions to make if some clash, on a per operator-matrix basis. An important takeaway from this related piece of work, is that we should consider the memory access patterns and costs when weighting the edges.

3.3 Optimized Code Generation for Finite Element Local Assembly Using Symbolic Manipulation

In 2013, Russel and Kelly [12] presented EXCAFE, a code generator that used symbolic integration to perform finite element local assembly, as opposed to the then popular approaches; quadrature and tensor contractions. This allowed the authors to perform a CSE pass, which was crucial to achieving at least similar performance levels as quadrature and tensor contraction implementations.
Russel and Kelly extended the factorizer from the work of Hosangadi et al. [2]. They had to adapt it to work with the representations they used, but also improved the work to be able to detect more types of factorizations.
Russel and Kelly achieved a reduction in operation count by over a factor of 4 in some cases, across their selected benchmark. Our work will require a factorizer and so the work of Russel and Paul would be a good starting point. However, as noted by the authors, the code-generator does not scale well with large problems. While the symbolic integration part of EXCAFE suffers with large sizes, more worryingly the factorization is both computationally and memory intensive.

3.4 Compiler-Level Matrix Multiplication Optimization for Deep Learning

In 2019, Zhang et al. [14] proposed two algorithms, a greedy best first search and a novel, Neighborhood Actor Advantage Critic (N-A2C) algorithm, designed to optimise the GEMM routine by choosing the optimal tiling configurations for the target hardware.
The nodes in the graphs were the tiling configurations. Tiling configurations that are close, are likely to have similar performance. Both algorithms dynamically searched new nodes by exploring configurations that were likely to not have similar performance as the current configuration. For the edges in the graphs used in both algorithms, the weights are calculated by measuring run-time differences between the nodes.
Since LIBXSMM is a JIT compiler, we can make the assumption that AVX-512 would be available to collect measurements, if we are targeting AVX-512 hardware during the JIT process. This has the potential to simplify the weight calculation on any graph model, as ideally our model would consider memory access, and not just FMA uses. The major disadvantage of this approach, as acknowledged by Zhang et al, is that the measurement is subject to large noise. This greatly impacts the decisions taken by the greedy BFS algorithm [14].
It is important to highlight that this paper looked into GEMM, and not sparse-dense MM. So the random, dynamic searching of configurations could be justified, as there was no prior information to take advantage of, other than the dimensions.

Chapter 4

Methodology

4.1 Environment

Software Version

All testing will be run on Ubuntu 18.04 LTS. The benchmark will be compiled using GCC version 7.5.0, but the exact compiler used should not effect performance gains as the LIBXSMM kernels are compiler agnostic. The updates to LIBXSMM made were made to the parent commit '51e64904fc53c19de79ec8e66414233f4fc4130e', which itself builds upon version 1.14. The maximum code buffer size was doubled in LIBXSMM for both the version with new additions and the reference/baseline version. Appendix B details which matrices required this and why.

Benchmark Controls

To minimise the gap from the peak kernel performance and the best run-time obtained, we can control a few things. Firstly, we can pin the benchmark to a single core. This prevents any repeat compulsory-cache misses from the process being moved to a different core. The benchmark should be run with real-time priority, which can be easily set within the Linux environment. This important measure prevents the OS from scheduling other processes to run on the core the benchmark is running on. Compared to the run-times of the kernels, the scheduling overhead may not be that significant, but the CPU time taken by other process can be very significant. Since we will be timing the time from when the kernel is called to when it returns, any time taken by other processes will effect the measured time.

Hardware

The benchmarks will be run on Amazon Web Services (AWS) Elastic Compute (EC2) nodes, that have Intel CPUs with AVX-512. The exact instance type is 'm5.xlarge' which uses a single core from the Intel Xeon Platinum 8175M, a Skylake-SP based CPU, along with 8GB of RAM. AWS provides exclusive access to the physical core, even though the entire CPU is shared. However, as the benchmark aims to keep reused data within L2, sharing L3 cache should not be detrimental. Disabling Hyper-Threading should not be necessary due to using a single core and setting CPU affinity.
AWS does not allow us to control the CPU frequency unless we rent enough cores or the entire CPU. This means any timing measurements will be affected by the turbo frequency of the CPU, which is likely to not remain constant due to varying CPU temperatures and other uncontrollable factors when using an AWS VM. After the first round of testing, a

reliable, stable average recording of 2.4GHz was found for the frequency, which is one of the AVX-512 heavy-use boost frequencies of the CPU.
Another pitfall of this strategy is having to share memory. In PyFR simulations, kernels are ran on all available cores with memory bandwidth being heavily used by all cores. In contract, on AWS it is highly unlikely all users will simultaneously used a lot of memory bandwidth, and so our single core VM achieves close to peak single core memory bandwidth found in [13]. AnandTech also showed that as more cores utilise memory bandwidth, the bandwidth available per core decreases. What this means is that any results that are found to be compute bound, could instead be memory bound in a typical PyFR simulation. Therefore, this will have to be considered when evaluating the following results.

4.2 Benchmark Suite

We will be testing the kernel performance for different operator matrices ($\mathbf{A}$) in the matrix multiplication from equation 4.1 with DP. $\alpha = 1$ and $\beta = 0$ will be the only versions tested, $\beta = 1$ would have the same effect on performance for the reference and updated versions of LIBXSMM. This is because all versions tested would load the stride of $\mathbf{C}$ in the same way, before continuing with their strategies to carry out the matrix multiplication.

$$\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C} \tag{4.1}$$

Emulating PyFR's use of LIBXSMM

The strategy found to be optimal with LIBXSMM for PyFR splits the $\mathbf{B}$ matrix every 48 columns. This takes advantage of the massive multi-threading available in a typical PyFR simulation setup. The setup effectively calls a 'hot' kernel with new data. The kernel is very likely to remain in L1, and only spill to L2 cache if the B matrix has a very, very large amount of rows (A has a very large amount of columns). The C program PyFR uses to call the kernels, is linked to the LIBXSMM library. If dynamic linking was to be used, once the kernel is 'hot', runtime linker look-ups should be resolved and cached, so function call expenses should be minimal over the entire simulation.
To provide performance metrics that translate well into PyFR's real-world use case, we propose a method that aims measure the performance of the kernel for $\mathbf{B}$ with 48 columns. For each operator matrix kernel, we pass to it a random $\mathbf{B}$ with $192,000$ columns of data. The time measured is just *before* calling the kernel to when the kernel *returns*, using the C '*gettimeofday*' function to measure.
Figure 4.1 shows the results of an experiment to determine if using X columns in B with a single call to the kernel, can be used to extrapolate the best performance when operating on $\mathbf{B}$ with 48 columns. The test operator matrix was taken from the PyFR examples (quad-p4-gauss-legendre-m132). If X is too low, then $\mathbf{B}$ remains in the last-level cache (LLC) and so the best performance measured would not be realistic for unseen $\mathbf{B}$. $X = 192,000$ was chosen as it was large enough to not fit in LLC, leading to the performance within the flat range in Figure 4.1 for the average execution time per 48 columns of $\mathbf{B}$.

Repeating Experiments

During a single benchmark run, each kernel is called 60 times with the $192,000$ column $\mathbf{B}$. The *best* time is recorded. For the evaluation, the entire benchmark is run 3 times and the average of the best times is used. The benchmark is statically linked to LIBXSMM, eliminating runtime linking overheads.
The benchmark consists of two separate sets of operator matrices, the first a collection of example operator matrices in PyFR, and the second a set of synthetic matrices.

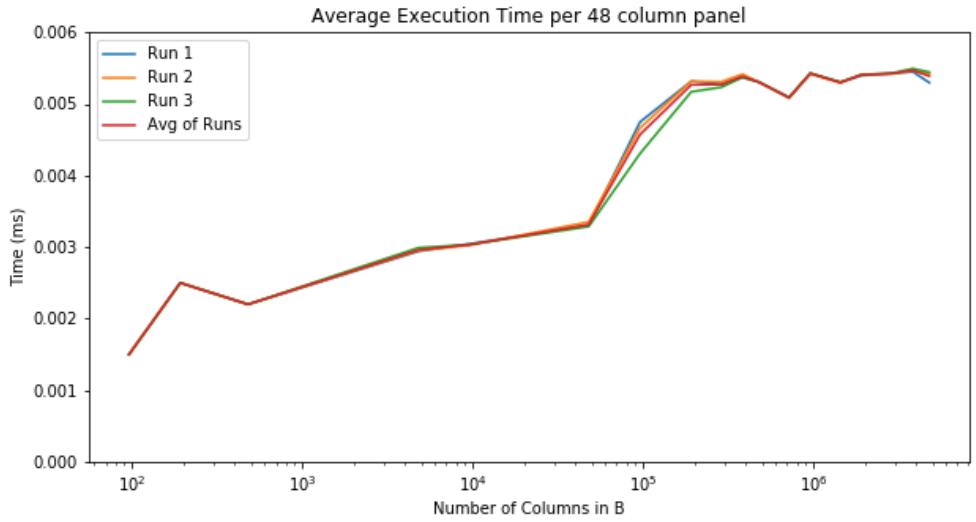


Figure 4.1: Investigate Number of B columns - Average Best Time per 48 columns

4.2.1 PyFR Example Operator Matrices

A subset of examples were taken from a set provided by PyFR. From the whole set (detailed in Appendix A), any matrix with less than or equal to **120** unique non-zeros were used. Matrices with over 120 unique non-zeros would have ended up using the same kernel generation strategy in our updated LIBXSMM and the reference version. In Table 4.1 'GL' is Gauss-Legendre quadrature, 'GLL' is Gauss-Legendre-Lobatto, 'WS' is Williams-Shunn and 'SH' is Shunn-Ham. 'all' is equivalent to having $m0, m3, m6, m132, m460$ operator matrices.

Order	1	2	3	4	5	6
Quadrilateral - GL	*all*	*all*	*all*	*all*	*all*	*all*
Quadrilateral - GLL	-	*all*	*all*	*all*	*all*	*all*
Hexahedra - GL	*all*	*all*	*all*	*all*	*all*	*all*
Hexahedra - GLL	-	*all*	*all*	*all*	*all*	*all*
Triangle - WS	*all*	*all*	*all*	$m0, m3, m6$	$m0$	$m0$
Tetrahedra - SH	*all*	*all*	$m0, m3, m6$	$m0$	-	-

Table 4.1: PyFR Example Operator Matrices Used in Benchmark

4.2.2 Synthetic Operator Matrices

As the operator matrices vary in more than one characteristic between them in the PyFR examples, a suite was synthesised to only vary one characteristic at a time. The base configuration was 128 rows and columns, a density of 0.05 and either 16 or 64 number of unique non-zeros (U). The data was uniformly randomly sampled from the available unique constants. The data was also placed uniformly randomly placed. Two versions of the base configuration were made to be able to compare against two routines LIBXSMM used, one for $U < 32$ and one for $U \geq 32$.

- Vary number of rows in A $\in [32, 64, 128, 256, 512, 1024]$. $U = 16$
- Vary number of rows in A $\in [32, 64, 128, 256, 512, 1024]$. $U = 64$

- Vary number of columns in A $\in [32, 64, 128, 256, 512, 1024]$. $U = 16$
- Vary number of columns in A $\in [32, 64, 128, 256, 512, 1024]$. $U = 64$
- Vary density $\in [0.01, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5]$. $U = 16$
- Vary density $\in [0.01, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5]$. $U = 64$
- Vary number of unique non-zeros in A $\in [8, 16, 24, 31, 40, 48, 56, 64, 78, 96, 112, 120, 128]$

4.3 Performance Metric

An alternative performance metric to time will be used for the evaluation. The aim is to provide insight into how well the hardware capabilities are utilised and to also better showcase performance speedups.
We call the metric used ***pseudo-FLOP/s***. For a given operator matrix, the pseudo-FLOPS (Floating Points Operations) is first counted. The algorithm to count the pseudo-FLOPS is a basic 3 loop matrix multiply, with the exception that when an element is 0 in **A**, the FLOPS are not counted. The algorithm assumes an FMA is available, and each FMA counts as two FLOPS. The count is then divided by the measured runtime of the kernel from the benchmark, resulting in a performance rate metric of 'pseudo-FLOP/s'.
The 'pseudo' prefix is used as the same FLOP count will be used across the different kernels/strategies implemented, even though the actual FLOPS required by an optimised kernel might be less. This leads to a different 'pseudo-FLOP/s' as the optimised kernel should have lower run-times, thus displaying the performance improvement in a hopefully more contextual way. Figure 4.2 transforms the time measurements from Figure 4.1, using the new 'pseudo-FLOP/s' measurement. The data being displayed is the same, but shown in a different context.

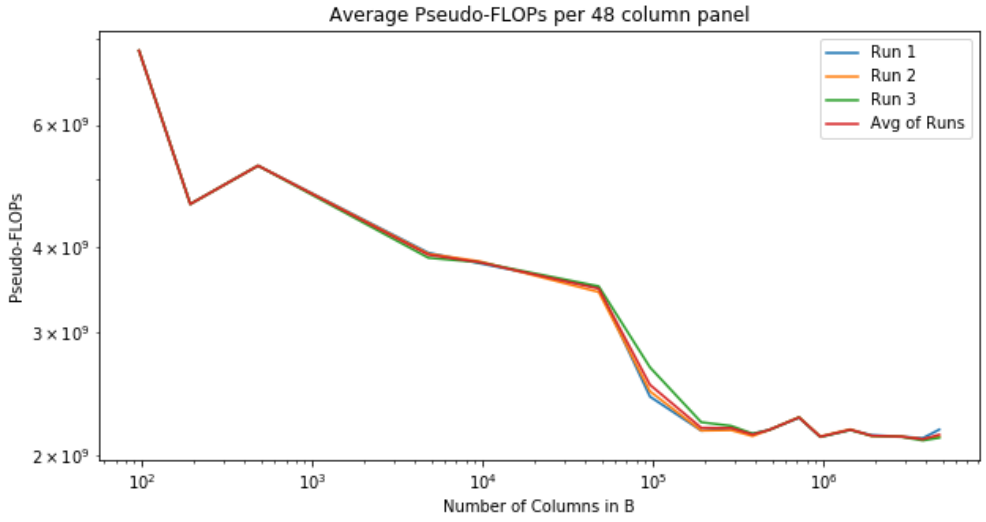


Figure 4.2: Investigate Number of B columns - Average pseudo-FLOP/s per 48 columns

Chapter 5

Register Packing

In this chapter we introduce a novel AVX-512 in-register file storage layout that enables single instruction broadcasting, regardless of the vector lane the value is stored in. Both the packing scheme and the method for broadcasting are detailed in Section 5.1. The strategy is evaluated against the reference LIBXSMM on the PyFR suite and synthetic suite. Finally, Intel VTune profiling results are used to further discuss the results.

5.1 Solution

5.1.1 AVX-512 Shuffle Instruction

The storage layout was designed to be able to use the *VSHUFF64X2* AVX-512 instruction. On Skylake-SP, it has a 3 cycle latency [11]. The instruction has two source vector register operands, an 8-bit integer operand and one destination vector operand. The 8-bit integer is encoded at compilation time.
The vector registers are split into sections of 128-bits, so each source and destination register has four sections. The *VSHUFF64X2* instruction copies 128-bit sections from the source registers to the destination.
The 8-bit integer, called the *selector*, selects which source sections to copy for a destination register. The selector is split into four groups of 2-bits. The lowest 2-bits select for the lowest 128-bits in the destination, and the highest 2-bits select for the highest 128-bits in the destination.
The lowest 4-bits of the selector, select for the first two 128-bit sections in the destination register, which are chosen from the first source register. A 2-bit selector can choose any of the four 128-bit sections from the source register.
So the lowest half (256-bits) of the destination register is given two 128-bit sections from the *first* source register. The 128-bit sections chosen can be the same or different. Similarly, the highest 4-bits of the selector choose 128-bit sections from the second source register. The highest half (256-bits) of the destination register is given two 128-bit sections from the *second* source register. The entire operation is illustrated in Figure 5.1 [1].

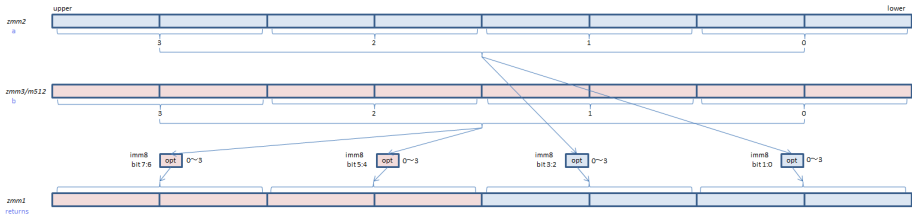
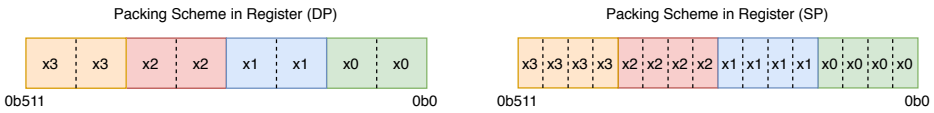


Figure 5.1: Illustration of VSHUFF64X2 [1]

5.1.2 Single Instruction Broadcasting

To make use of the *VSHUFF64X2* instruction for a single-instruction broadcast, the 128-bit sections have to contain *only* the desired value to be broadcast. This means that the value must be repeated for 128-bits. For double precision, this means storing the value twice. For single precision, the value must be stored four times. The logical register layout is shown in Figure 5.2, which shows that the are four 128-bits sections in a register. A trade-off is made by repeating values to obtain a single-instruction broadcast, that uses only one temporary register to hold the broadcasted value. Alternative methods could pack more value in the registers, but would require more instructions and temporary registers to broadcast the values.



(a) Double Precision (b) Single Precision

Figure 5.2: Layout within Register for DP and SP values

Figure 5.3 shows how the *VSHUFF64X2* instruction can be used with the logical storage layout from Figure 5.2 to achieve a single-instruction broadcast, regardless of which lanes (groups of sequential lanes due to repetition) the value is stored in. The same register is used as **both** source operands, which can be seen in Figure 5.3 by both the source registers being *zmm*0. This allows the selector to choose the same 128-bit source section, for all four 128-bit sections in the destination register. The end result is that the selected source section is broadcast to all four sections - a $1-4$ broadcast operation. The same process would apply to SP, and would also be a $1-4$ broadcast, as the SP value must be repeated four times to fill the 128-bit section.

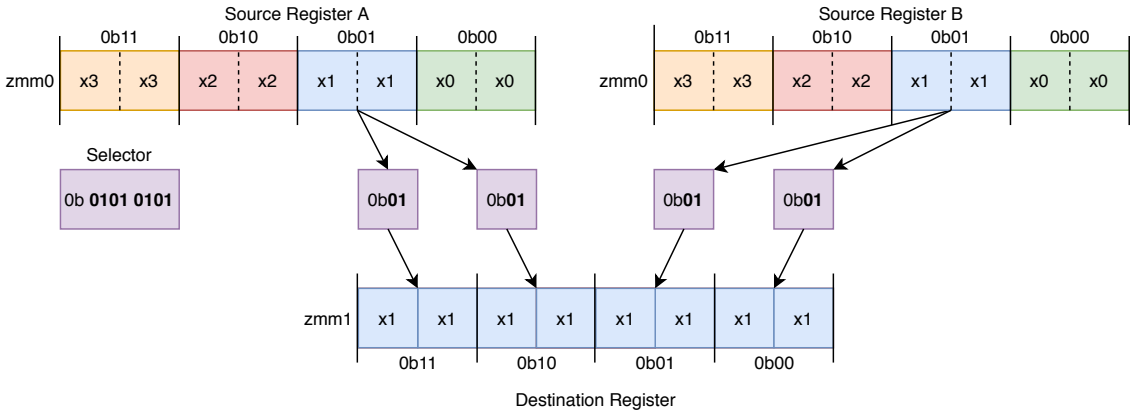


Figure 5.3: Using VSHUFF64X2 to Broadcast

Figure 5.4 showcases the four possible broadcasts and the corresponding selector values required to achieve them. The binary values show that a 2-bit value is repeated, which is due to the same source 128-bit section being chosen for all four destination sections.

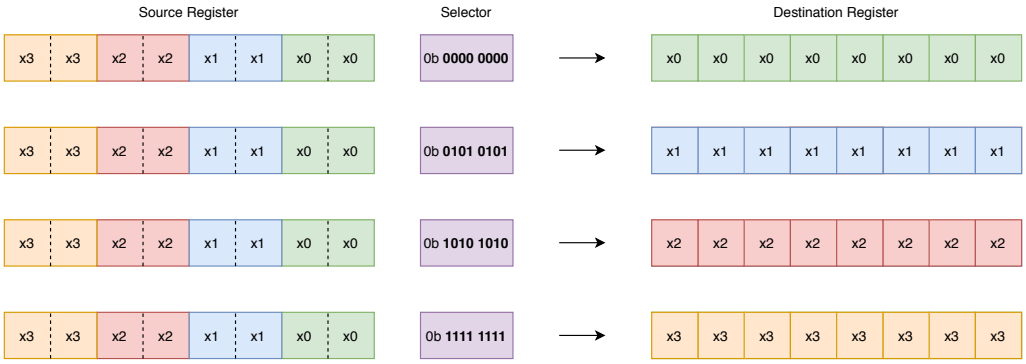


Figure 5.4: The four possible effective broadcasts

5.1.3 Register Packing for the MM routine

By fully storing **A** in the register file with the packing scheme, no memory access is required for **A** during the MM routine. The adaptation made to the routine from Section 2.3.2 is to broadcast the required value of **A** from the *register file* using the *VSHUFF64X2* instruction. Figure 5.5 shows that the 31^{st} register is used as the temporary register to hold the broadcasted value. This is then multiplied with the stride of **B** (loaded from memory) in the FMA to calculate (part of the accumulation of) the stride of **C**.

Hypothesis

1: Kernels for operator matrices where the number of unique non-zeros (U), is bound by $U \leq 31$, should not have decreased performance when using register packing compared to other strategies deployed by LIBXSMM. The run-time broadcasting should *not* add to the critical path of execution.
2: Kernels for operator matrices where the number of unique non-zeros (U), is bound by $31 < U \leq 120$, should have increased performance compared to other strategies deployed by LIBXSMM.

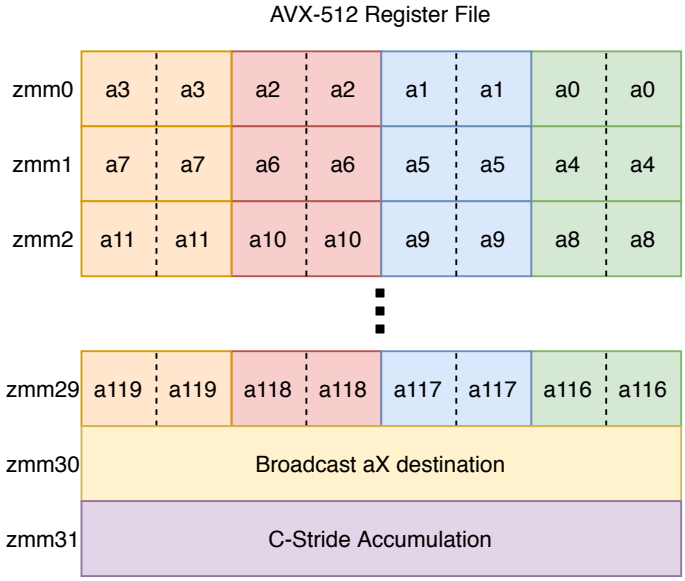
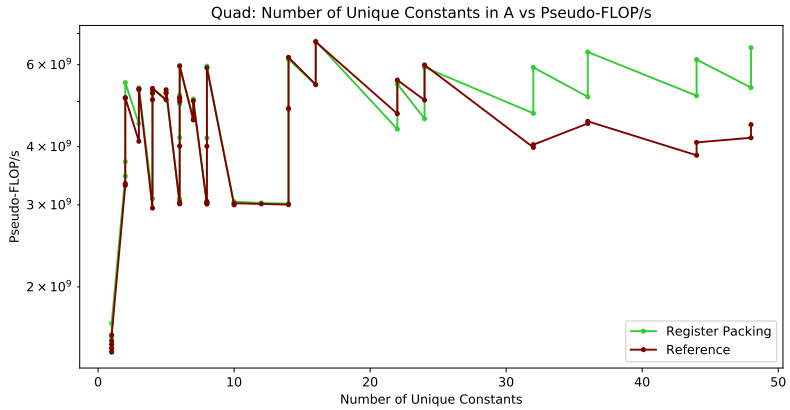


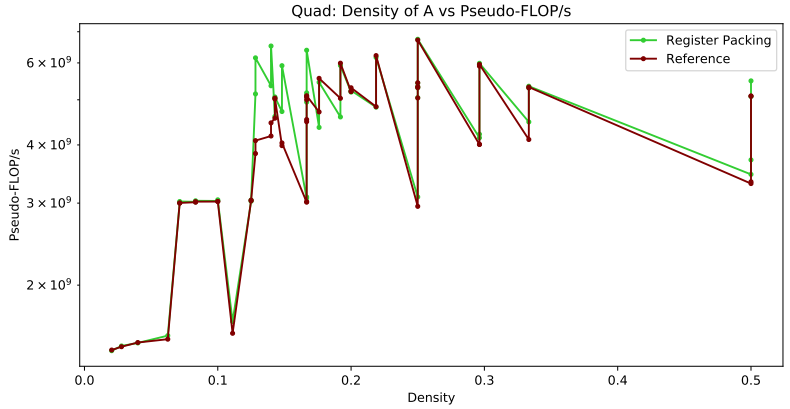
Figure 5.5: The Vector Register File logical layout when using register packing for the SpMM routine

5.2 Evaluation on PyFR Suite

5.2.1 Quadrilaterals



(a) Plotting against Number of Unique Non-Zeros



(b) Plotting against Density

Figure 5.6: Register Packing vs Reference Performance - PyFR Quadrilateral Examples

5.2.2 Hexahedra

5.2.3 Triangles

5.2.4 Tetrahedra

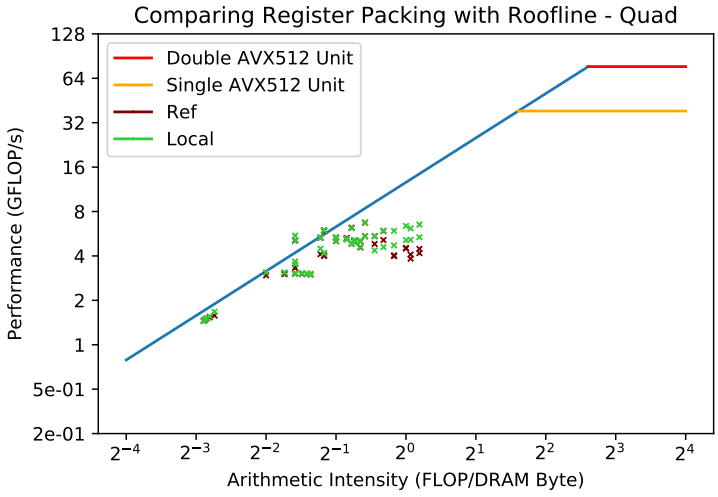
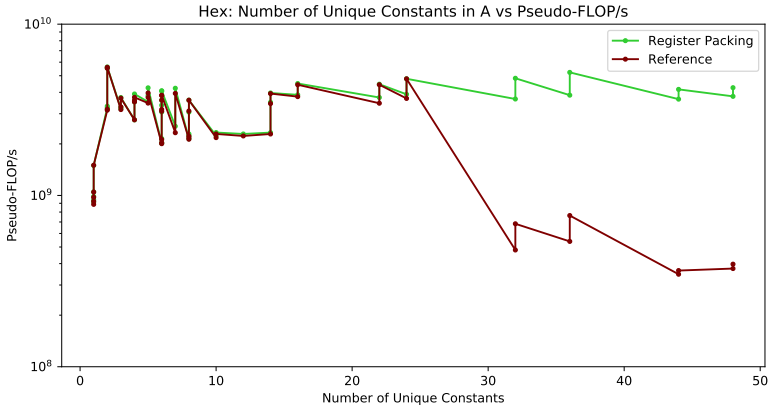
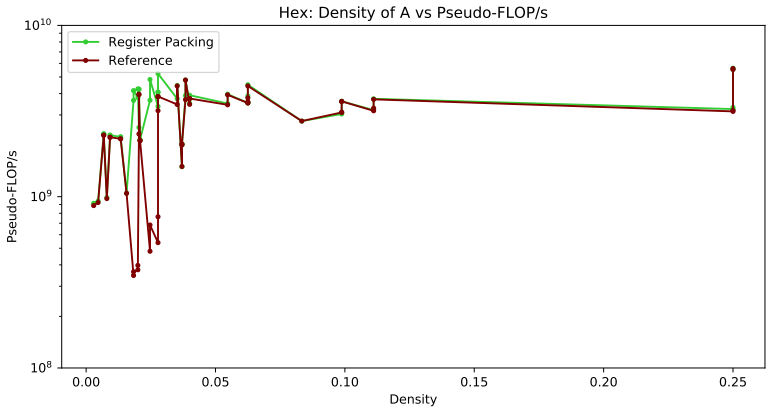


Figure 5.7: Register Packing vs Reference: Roofline Plot - PyFR Quadrilateral Examples



(a) Plotting against Number of Unique Non-Zeros



(b) Plotting against Density

Figure 5.8: Register Packing vs Reference Performance - PyFR Hexahedra Examples

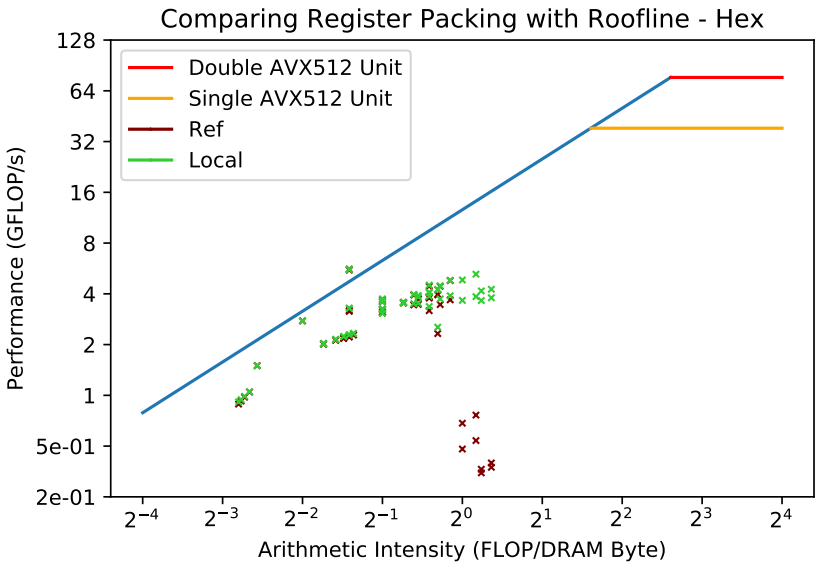
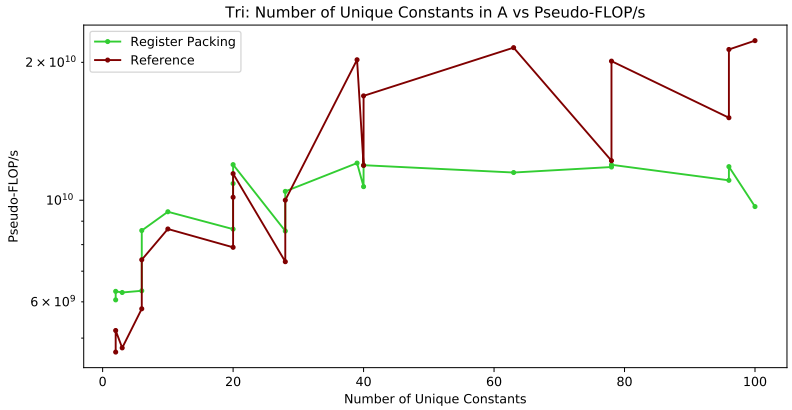
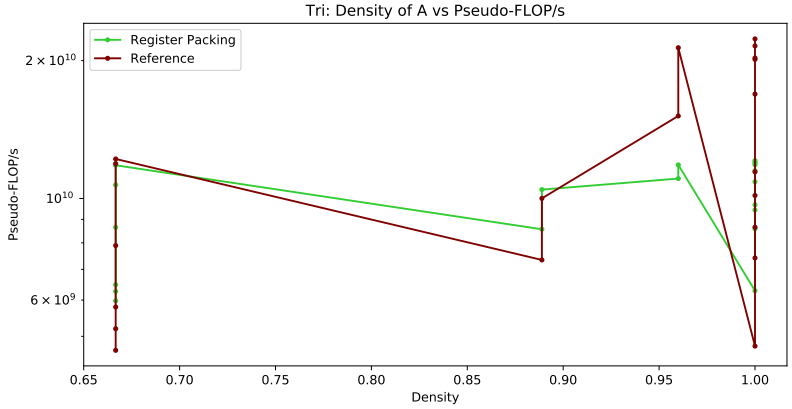


Figure 5.9: Register Packing vs Reference: Roofline Plot - PyFR Hexahedra Examples



(a) Plotting against Number of Unique Non-Zeros



(b) Plotting against Density

Figure 5.10: Register Packing vs Reference Performance - PyFR Triangles Examples

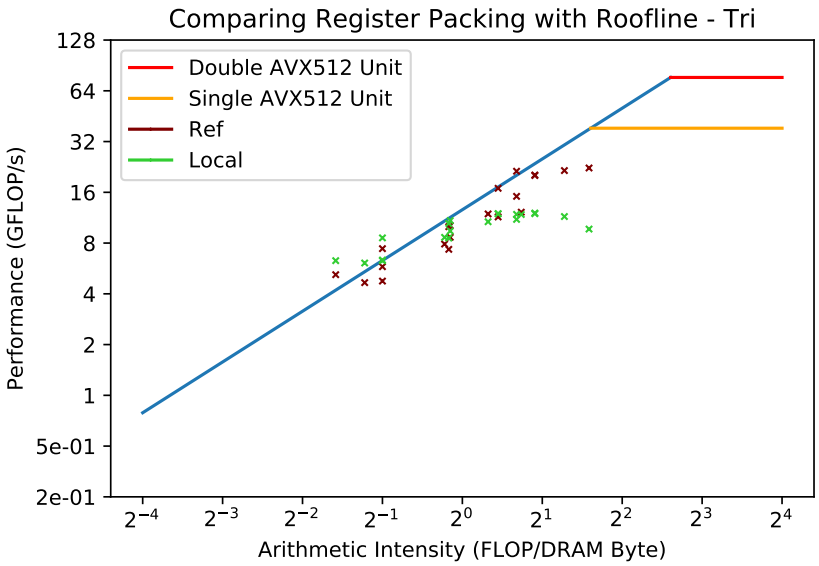
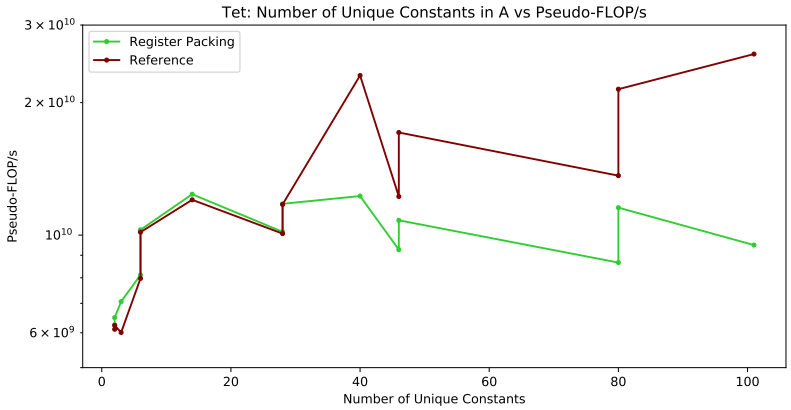
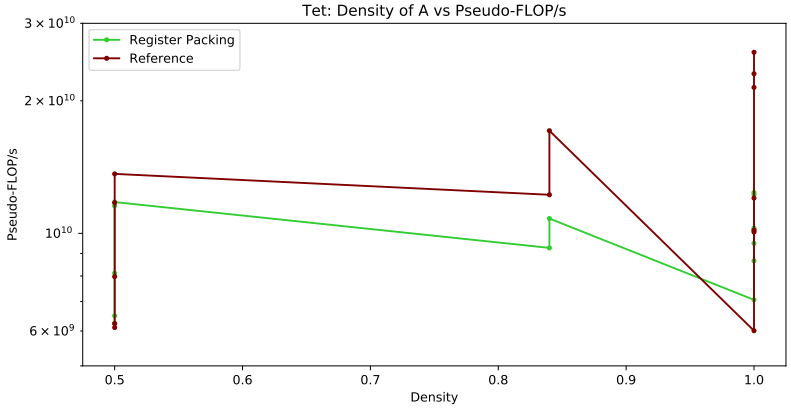


Figure 5.11: Register Packing vs Reference: Roofline Plot - PyFR Triangles Examples



(a) Plotting against Number of Unique Non-Zeros



(b) Plotting against Density

Figure 5.12: Register Packing vs Reference Performance - PyFR Tetrahedra Examples

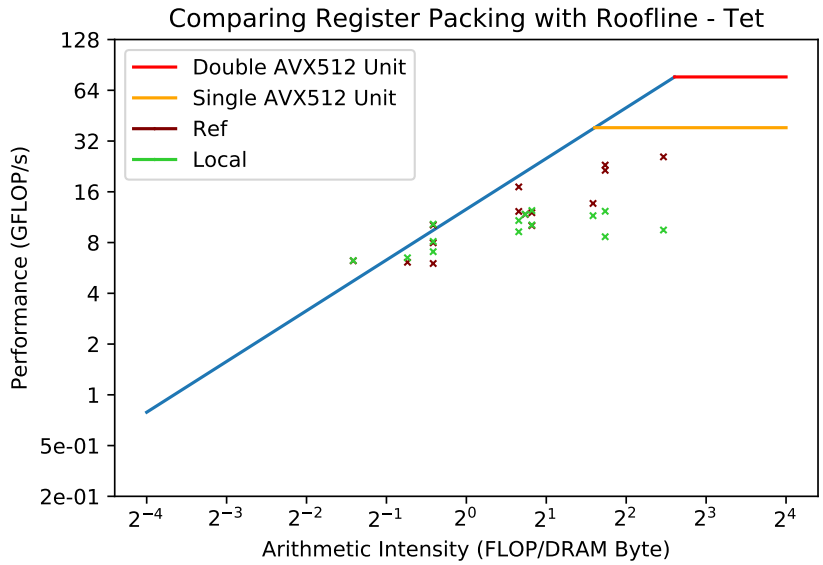


Figure 5.13: Register Packing vs Reference: Roofline Plot - PyFR Tetrahedra Examples

5.3 Evaluation on Synthetic Suite

5.3.1 Vary Number of Rows

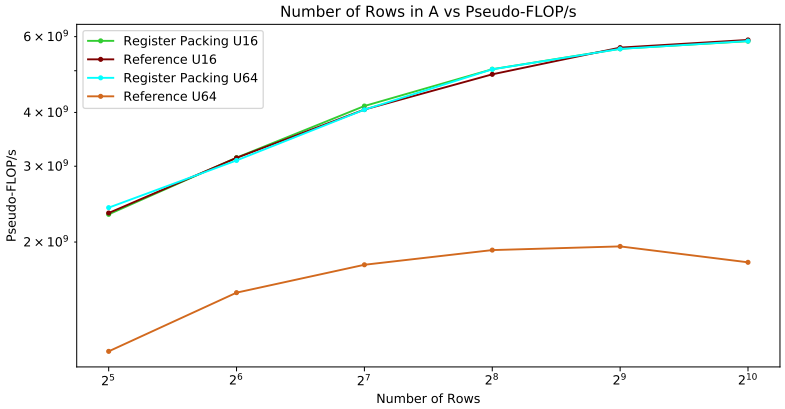
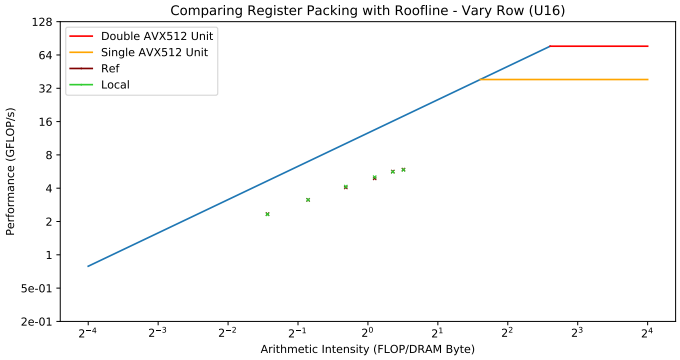
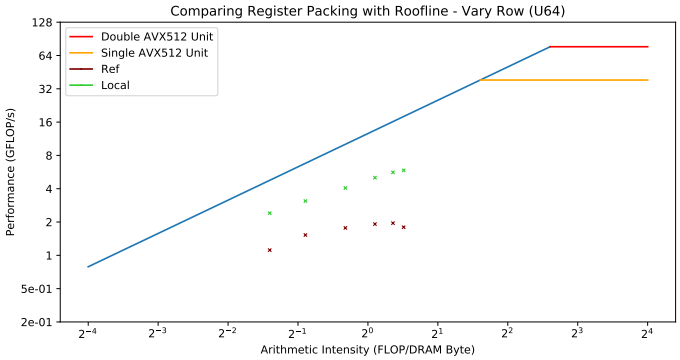


Figure 5.14: Register Packing vs Reference: Performance - Synthetic Suite Vary Number of Rows



(a) 16 Unique Non-Zeros



(b) 64 Unique Non-Zeros

Figure 5.15: Register Packing vs Reference: Roofline Plot - Synthetic Suite Vary Number of Rows

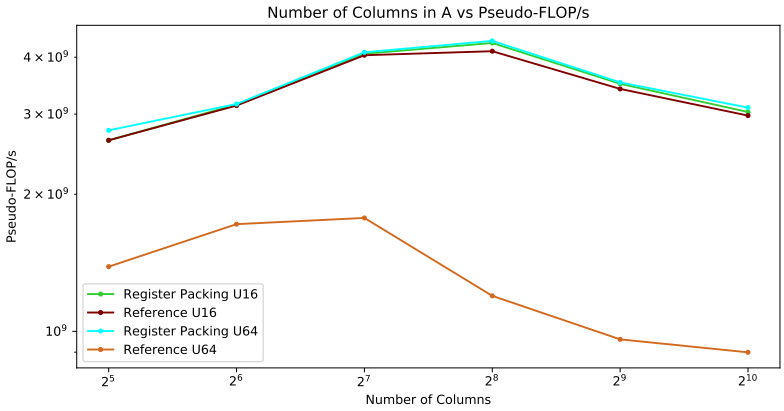
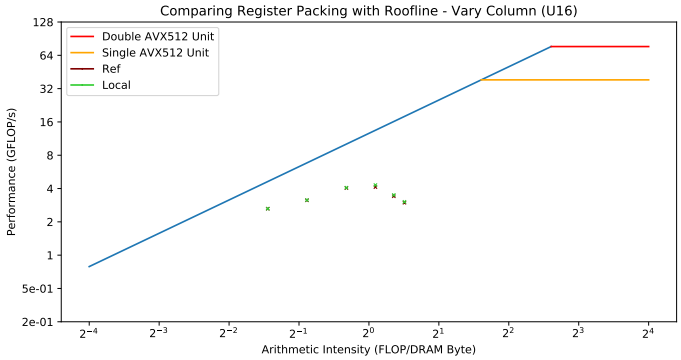
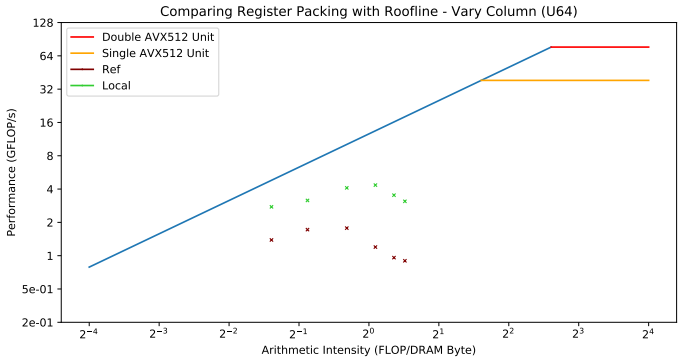


Figure 5.16: Register Packing vs Reference: Performance - Synthetic Suite Vary Number of Columns



(a) 16 Unique Non-Zeros



(b) 64 Unique Non-Zeros

Figure 5.17: Register Packing vs Reference: Roofline Plot - Synthetic Suite Vary Number of Columns

5.3.2 Vary Number of Columns

5.3.3 Vary Density

5.3.4 Vary Number of Unique Non-Zeros

5.4 Profiling Results

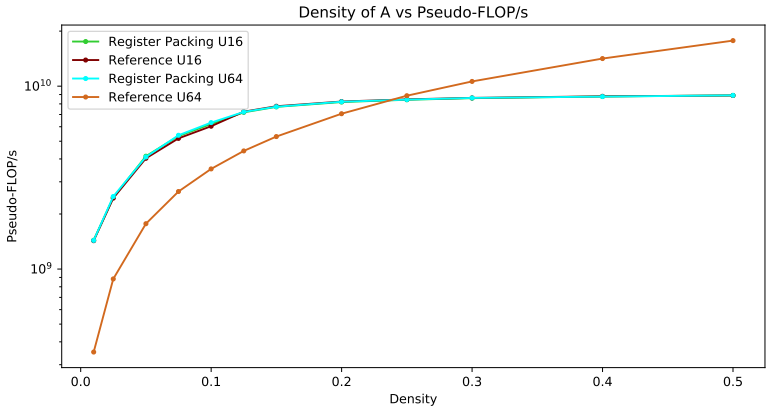
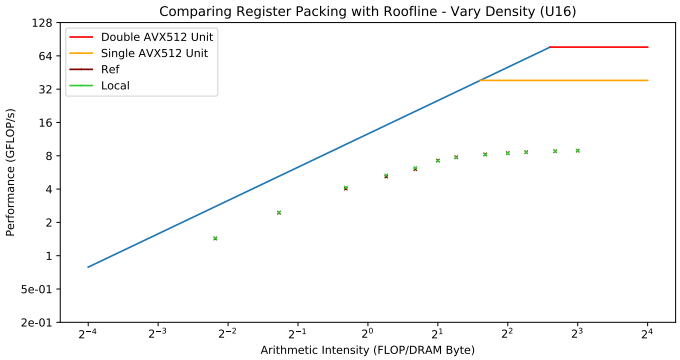
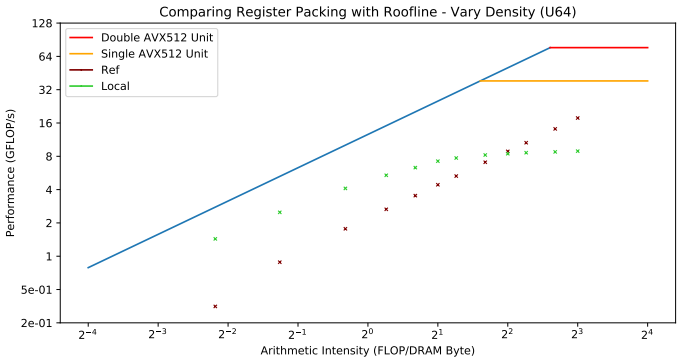


Figure 5.18: Register Packing vs Reference: Performance - Synthetic Suite Vary Number of Density



(a) 16 Unique Non-Zeros



(b) 64 Unique Non-Zeros

Figure 5.19: Register Packing vs Reference: Roofline Plot - Synthetic Suite Vary Density

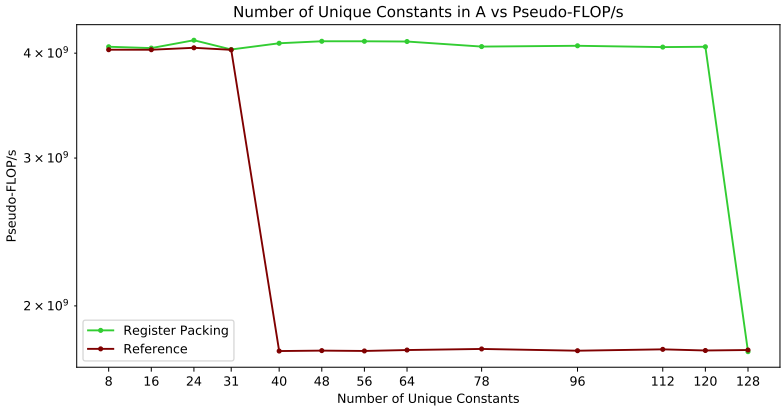


Figure 5.20: Register Packing vs Reference: Performance - Synthetic Suite Vary Number of Unique Non-Zeros

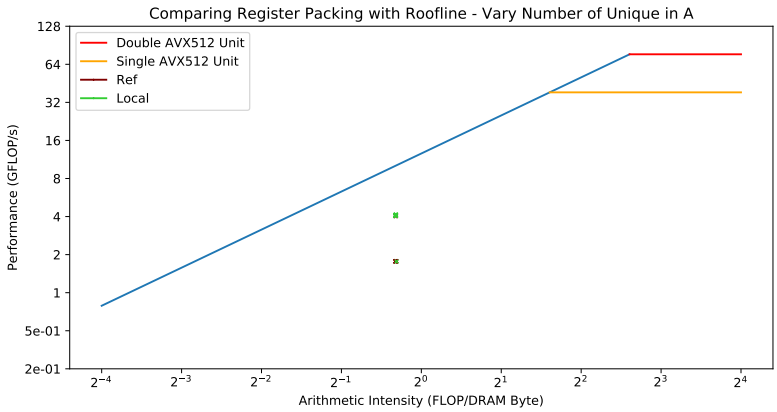


Figure 5.21: Register Packing vs Reference: Roofline Plot - Synthetic Suite Vary Number of Unique Non-Zeros

Bibliography

[1] Simd instruction list.

[2] R. Kastner A. Hosangadi, F. Fallah. Optimizing polynomial expressions by algebraic factorization and common subexpression elimination. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25:2012 – 2022, 2006.

[3] Gerrit Becker, Michael Schäfer, and Antony Jameson. An advanced nurbs fitting procedure for post-processing of grid-based shape optimizations. *49th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 01 2011.

[4] Patrice Castonguay, David Williams, Peter Vincent, Manuel López Morales, and Antony Jameson. On the development of a high-order, multi-gpu enabled, compressible viscous flow solver for mixed unstructured grids. 06 2011.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. 10 2014.

[6] Intel Corporation. Libxsmm. 2020.

[7] Intel Corporation. Math kernel library. 2020.

[8] NVIDIA Corporation. cusparse. 2020.

[9] P.E. Vincent F.D. Witherden, A.M. Farrington. Pyfr: An open source framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185:3028–3040, 2014.

[10] P.E. Vincent F.D. Witherden, B.C. Vermeire. Heterogeneous computing on mixed unstructured grids with pyfr. *Computers Fluids*, 120:173–186, 2015.

[11] Agner Fog. Instruction tables, Aug 2019.

[12] P.H.J. Kelly F.P. Russel. Optimized code generation for finite element local assembly using symbolic manipulation. *ACM Transactions on Mathematical Software*, 39, 2013.

[13] Johan De Gelas and Ian Cutress. Sizing up servers: Intel's skylake-sp xeon versus amd's epyc 7000 - the server cpu battle of the decade?, Jul 2017.

[14] H. Zang D.H. Park H. Zhang, X. Cheng. Compiler-level matrix multiplication optimization for deep learning. *ArXiv*, abs/1909.10616, 2019.

[15] H.T. Huynh. A flux reconstruction approach to high-order schemes including discontinuous galerkin methods. *AIAA paper*, 4079, 2007.

[16] J.M. Park. Full unrolling combined with tiling for small sparse matrix multiplies in fluid dynamics. *MEng Thesis, Department of Computing, Imperial College London*, 2016.

[17] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. Faster cnns with direct sparse convolutions and guided pruning, 2016.

[18] T. Price. Optimising small sparse special matrix multiplies through cunning vectorisation. *MEng Thesis, Department of Computing, Imperial College London*, 2019.

[19] L.R. Scott A.R. Terrel R.C. Kirby, A. Logg. Topological optimization of the evaluation of finite element matrices. *Society for Industrial and Applied Mathematics*, 28:224–240, 2006.

[20] M. Roth. Compiler optimisations in high-performance matrix multiplication kernels. *MEng Thesis, Department of Computing, Imperial College London*, 2015.

[21] Z.J. Wang, Y. Liu, C. Lacor, and J.L.F. Azevedo. Chapter 9 - spectral volume and spectral difference methods. In Rémi Abgrall and Chi-Wang Shu, editors, *Handbook of Numerical Methods for Hyperbolic Problems*, volume 17 of *Handbook of Numerical Analysis*, pages 199 – 226. Elsevier, 2016.

[22] F.D. Witherden, P.E. Vincent, and A. Jameson. Chapter 10 - high-order flux reconstruction schemes. In Rémi Abgrall and Chi-Wang Shu, editors, *Handbook of Numerical Methods for Hyperbolic Problems*, volume 17 of *Handbook of Numerical Analysis*, pages 227 – 263. Elsevier, 2016.

[23] B.D. Wozniak. Gimmik-generating bespoke matrix multiplication kernels for accelerators: Application to high-order computational [U+FB02]uid dynamics. *MEng Thesis, Department of Computing, Imperial College London*, 2014.

Appendix A

PyFR Example Operator Matrices Characteristics

The follwing tables describe the example operator matrices from PyFR in terms of:

- Number of Rows, R
- Number of Columns, C
- Density, ρ
- Number of unique non-zeros, U

Matrix	R	C	ρ	U
m0	8	4	0.5000	2
m3	4	8	0.5000	2
m6	8	8	0.2500	4
m132	4	8	0.5000	2
m460	8	4	0.5000	2

(a) First Order

Matrix	R	C	ρ	U
m0	12	9	0.3333	3
m3	9	12	0.3333	3
m6	18	12	0.1667	6
m132	9	18	0.2963	8
m460	18	9	0.2963	8

(b) Second Order

Matrix	R	C	ρ	U
m0	16	16	0.2500	4
m3	16	16	0.2500	4
m6	32	16	0.1250	8
m132	16	32	0.2500	16
m460	32	16	0.2500	16

(c) Third Order

Matrix	R	C	ρ	U
m0	20	25	0.2000	5
m3	25	20	0.2000	5
m6	50	20	0.1000	10
m132	25	50	0.1920	24
m460	50	25	0.1920	24

(d) Fourth Order

Matrix	R	C	ρ	U
m0	24	36	0.1667	6
m3	36	24	0.1667	6
m6	72	24	0.0833	12
m132	36	72	0.1667	36
m460	72	36	0.1667	36

(e) Fifth Order

Matrix	R	C	ρ	U
m0	28	49	0.1429	7
m3	49	28	0.1429	7
m6	98	28	0.0714	14
m132	49	98	0.1399	48
m460	98	49	0.1399	48

(f) Sixth Order

Table A.1: Quadrilateral Gauss-Legendre Operator Matrices Characteristics

Matrix	R	C	ρ	U
m0	12	9	0.1111	1
m3	9	12	0.3333	3
m6	18	12	0.1667	6
m132	9	18	0.2963	6
m460	18	9	0.2963	6

(a) Second Order

Matrix	R	C	ρ	U
m0	16	16	0.0625	1
m3	16	16	0.2500	4
m6	32	16	0.1250	6
m132	16	32	0.2188	14
m460	32	16	0.2188	14

(b) Third Order

Matrix	R	C	ρ	U
m0	20	25	0.0400	1
m3	25	20	0.2000	4
m6	50	20	0.1000	8
m132	25	50	0.1760	22
m460	50	25	0.1760	22

(c) Fourth Order

Matrix	R	C	ρ	U
m0	24	36	0.0278	1
m3	36	24	0.1667	6
m6	72	24	0.0833	8
m132	36	72	0.1481	32
m460	72	36	0.1481	32

(d) Fifth Order

Matrix	R	C	ρ	U
m0	28	49	0.0204	1
m3	49	28	0.1429	5
m6	98	28	0.0714	10
m132	49	98	0.1283	44
m460	98	49	0.1283	44

(e) Sixth Order

Table A.2: Quadrilateral Gauss-Legendre-Lobatto Operator Matrices Characteristics

Matrix	R	C	ρ	U
m0	24	8	0.2500	2
m3	8	24	0.2500	2
m6	24	24	0.0833	4
m132	8	24	0.2500	2
m460	24	8	0.2500	2

(a) First Order

Matrix	R	C	ρ	U
m0	54	27	0.1111	3
m3	27	54	0.1111	3
m6	81	54	0.0370	6
m132	27	81	0.0988	8
m460	81	27	0.0988	8

(b) Second Order

Matrix	R	C	ρ	U
m0	96	64	0.0625	4
m3	64	96	0.0625	4
m6	192	96	0.0208	8
m132	64	192	0.0625	16
m460	192	64	0.0625	16

(c) Third Order

Matrix	R	C	ρ	U
m0	150	125	0.0400	5
m3	125	150	0.0400	5
m6	375	150	0.0133	10
m132	125	375	0.0384	24
m460	375	125	0.0384	24

(d) Fourth Order

Matrix	R	C	ρ	U
m0	216	216	0.0278	6
m3	216	216	0.0278	6
m6	648	216	0.0093	12
m132	216	648	0.0278	36
m460	648	216	0.0278	36

(e) Fifth Order

Matrix	R	C	ρ	U
m0	294	343	0.0204	7
m3	343	294	0.0204	7
m6	1029	294	0.0068	14
m132	343	1029	0.0200	48
m460	1029	343	0.0200	48

(f) Sixth Order

Table A.3: Hexahedra Gauss-Legendre Operator Matrices Characteristics

Matrix	R	C	ρ	U
m0	54	27	0.0370	1
m3	27	54	0.1111	3
m6	81	54	0.0370	6
m132	27	81	0.0988	6
m460	81	27	0.0988	6

(a) Second Order

Matrix	R	C	ρ	U
m0	96	64	0.0156	1
m3	64	96	0.0625	4
m6	192	96	0.0208	6
m132	64	192	0.0547	14
m460	192	64	0.0547	14

(b) Third Order

Matrix	R	C	ρ	U
m0	150	125	0.0080	1
m3	125	150	0.0400	4
m6	375	150	0.0133	8
m132	125	375	0.0352	22
m460	375	125	0.0352	22

(c) Fourth Order

Matrix	R	C	ρ	U
m0	216	216	0.0046	1
m3	216	216	0.0278	6
m6	648	216	0.0093	8
m132	216	648	0.0247	32
m460	648	216	0.0247	32

(d) Fifth Order

Matrix	R	C	ρ	U
m0	294	343	0.0029	1
m3	343	294	0.0204	5
m6	1029	294	0.0068	10
m132	343	1029	0.0183	44
m460	1029	343	0.0183	44

(e) Sixth Order

Table A.4: Hexahedra Gauss-Legendre-Lobatto Operator Matrices Characteristics

Matrix	R	C	ρ	U
m0	6	3	1.0000	3
m3	3	6	1.0000	6
m6	6	6	0.6667	6
m132	3	6	0.6667	2
m460	6	3	0.6667	2

(a) First Order

Matrix	R	C	ρ	U
m0	9	6	1.0000	10
m3	6	9	1.0000	20
m6	12	9	0.6667	20
m132	6	12	0.8889	28
m460	12	6	0.8889	28

(b) Second Order

Matrix	R	C	ρ	U
m0	12	10	1.0000	20
m3	10	12	1.0000	40
m6	20	12	0.6667	40
m132	10	20	0.9600	96
m460	20	10	0.9600	96

(c) Third Order

Matrix	R	C	ρ	U
m0	15	15	1.0000	39
m3	15	15	1.0000	78
m6	30	15	0.6667	78
m132	15	30	0.9600	216
m460	30	15	0.9600	216

(d) Fourth Order

Matrix	R	C	ρ	U
m0	18	21	1.0000	63
m3	21	18	1.0000	126
m6	42	18	0.6667	126
m132	21	42	0.9796	432
m460	42	21	0.9796	432

(e) Fifth Order

Matrix	R	C	ρ	U
m0	21	28	1.0000	100
m3	28	21	1.0000	200
m6	56	21	0.6667	200
m132	28	56	0.9796	768
m460	56	28	0.9796	768

(f) Sixth Order

Table A.5: Triangles Williams-Shunn Operator Matrices Characteristics

Matrix	R	C	ρ	U
m0	12	4	1.0000	3
m3	4	12	1.0000	6
m6	12	12	0.5000	6
m132	4	12	0.5000	2
m460	12	4	0.5000	2

(a) First Order

Matrix	R	C	ρ	U
m0	24	10	1.0000	14
m3	10	24	1.0000	28
m6	30	24	0.5000	28
m132	10	30	0.8400	46
m460	30	10	0.8400	46

(b) Second Order

Matrix	R	C	ρ	U
m0	40	20	1.0000	40
m3	20	40	1.0000	80
m6	60	40	0.5000	80
m132	20	60	0.9100	200
m460	60	20	0.9100	200

(c) Third Order

Matrix	R	C	ρ	U
m0	60	35	1.0000	101
m3	35	60	1.0000	202
m6	105	60	0.5000	202
m132	35	105	0.9339	608
m460	105	35	0.9339	608

(d) Fourth Order

Matrix	R	C	ρ	U
m0	84	56	1.0000	214
m3	56	84	1.0000	428
m6	168	84	0.5000	428
m132	56	168	0.9541	1568
m460	168	56	0.9541	1568

(e) Fifth Order

Matrix	R	C	ρ	U
m0	112	84	1.0000	425
m3	84	112	1.0000	850
m6	252	112	0.5000	850
m132	84	252	0.9637	3520
m460	252	84	0.9637	3520

(f) Sixth Order

Table A.6: Tetrahedra Shunn-Ham Operator Matrices Characteristics

Appendix B

LIBXSMM Code Buffer Size

LIBXSMM has a maximum code buffer size of 128 KB (v1.15 [6]). This limit is set to prevent the JIT process from writing too large a function for the user process. A very large code size would lead to bad instruction cache rates. Reaching the last-level of cache or even memory due to the routine being so large would be bad for performance. IF the limit was reached, a fallback strategy is used. If the limit is reached in the case of A of being sparse, then LIBXSMM v1.15 fallbacks to a dense matrix multiply routine.
However, in our evaluation, the aim was to compare the algorithms used to generate the code, and not necessarily what would be best for the general user in a production release of LIBXSMM. For that reason, the code buffer size was doubled to 256 KB, to be able to handle larger loop unrolling. This removed buffer size limits for 5 operator matrices in the benchmark.
The following list details which matrices ran into the previous 128 KB limit, when using the Register Packing update from Chapter 5:

- PyFR Example: Hexahedra Sixth-Order Gauss-Legendre $m132$
- PyFR Example: Hexahedra Sixth-Order Gauss-Legendre $m460$
- PyFR Example: Hexahedra Sixth-Order Gauss-Legendre-Lobatto $m132$
- PyFR Example: Hexahedra Sixth-Order Gauss-Legendre-Lobatto $m460$

These were the largest size operator matrices in the benchmark, all having a size of $1029 * 343$.
Only one matrix from the synthetic suite ran into the buffer size limit, when using the reference LIBXSMM version:

- Synthetic: Density of 0.5, with 64 unique non-zeros. Size of $128 * 128$