On the Generality of Matrix Multiplication

Andrew Alex
aalex@ucsb.edu
UC Santa Barbara

Zachary D. Sisco
zsisco@ucsb.edu
UC Santa Barbara

Jonathan Balkind
jbalkind@ucsb.edu
UC Santa Barbara

Abstract

Matrix multiplication is an operation present in many hardware accelerators on many device classes. While several direct applications for this hardware make it useful on its own, we show that it is possible to leverage this hardware to perform general-purpose computation. We present a mapping from a "Turing-complete", one instruction-set computer and then follow with preliminary work towards mapping to a more realistic RISC-style ISA. We leverage program synthesis to generate the matrix representations of these operations and explain mechanisms that would make this a feasible model of computation given the appropriate hardware.

1 Overview

Matrix multiplication is used in a variety of applications spanning from internet search [2] to quantum mechanics [9] and machine learning [1]. Because of its parallel nature and its ubiquity, various hardware accelerators for matrix multiplication have been built to make it as fast as possible [7, 14, 19]. As these devices become more prevalent, there is a natural opportunity for compiler writers to target this hardware for alternative use cases. Previous attempts to use these devices for general-purpose compute [5] show promise for applications that have matrix operations as a part of their critical path. We instead focus on the wider goal of executing applications without a clear matrix component as a matrix multiplication. We use the solver-aided programming language *Rosette* [12, 13] to find mappings from traditional, assembly-style instructions to a matrix multiplication equivalent. As a proof of concept, we provide a mapping to the one instruction set computer *SUBLEQ* [3, 8]. As a "Turing-complete" instruction set, this mapping provides proof that a matrix multiplication machine is possible and its feasibility in real-world systems is only a matter of finding sufficient optimizations. We argue that our matrix multiplication machine has the following potential benefits:

Unlocking New Compiler Optimizations. Providing compiler developers with a tractable way to target the matrix multiplication hardware available on most modern devices for purposes outside its original intent expands the design space for potential program optimizations and economizes the hardware's inclusion.

Exploitation of Highly-Parallel Hardware. Matrix multiplication hardware for *nxn* matrices performs n^2 operations in parallel. If mapped cleverly, many instructions could have their results computed simultaneously. The reduced control overhead compared with conventional CPUs of matrix multiplication hardware has the potential to make matrix multiplication programs more energy efficient than their CPU counterparts.

Section 2 demonstrates how to compute *SUBLEQ* with matrices. Section 3 presents a sample of the mappings we've found from traditional assembly instructions to matrix multiplications, and Section 4 presents our plans for future work.

2 *SUBLEQ* Representation

SUBLEQ is a one instruction-set computer with a single instruction *SUBLEQ A B C*, defined by:

```
mem[B] := (mem[B] - mem[A])
if (mem[B] ≤ 0)
  pc := C
else
  pc := pc + 1
```

The following is the matrix multiplication equivalent. We use "−" to indicate a "don't care" value.

$$\begin{pmatrix} mem[B] & -1 \\ 1 & C \end{pmatrix} \begin{pmatrix} 1 & C \\ mem[A] & 0 \end{pmatrix} = \begin{pmatrix} mem[B] - mem[A] & - \\ - & C \end{pmatrix}$$

We rely on external zero and sign-bit checks on the $(0, 0)$ element which we'll assume are either available directly on the accelerator or handled by the host (as is typical with element-wise operations on conventional TPUs) [7]. We include C in the result to show how metadata can be persisted through the operation if necessary for the hardware.

We automatically derive the matrix multiplication representation by encoding state updates in the *SUBLEQ* instruction (`mem`, `pc`) as symbolic values. We formulate a program synthesis query to fill holes in a matrix multiplication equation that satisfies the *SUBLEQ* instruction behavior:

$$\exists\, X, Y \;.\; \forall\, A, B, C$$
$$\exists\, i, j \;.\; XY_{i,j} = (mem[B] - mem[A])$$
$$\exists\, m, n \;.\; XY_{m,n} = C$$

This query can be solved on a laptop in 10-15 seconds.

3 Assembly to Matrix Mappings

Figure 1 shows a subset of the mappings from assembly-style instructions to matrix multiplications that our program synthesis tool discovered. We defined a grammar for functions from register operands to matrices and then from a matrix to the expected write-back value of the original instruction.

Set Less Than rd rs rt

$$\begin{pmatrix} reg[rs] & 1 \\ - & - \end{pmatrix}\begin{pmatrix} -1 & - \\ reg[rt] & - \end{pmatrix} = \begin{pmatrix} reg[rt] - reg[rs] & - \\ - & - \end{pmatrix}$$

Add rd rs rt

$$\begin{pmatrix} reg[rs] & 1 \\ - & - \end{pmatrix}\begin{pmatrix} 1 & - \\ reg[rt] & - \end{pmatrix} = \begin{pmatrix} reg[rs] + reg[rt] & - \\ - & - \end{pmatrix}$$

And rd rs rt (2-bit operands)

$$\begin{pmatrix} reg[rs][0] & 0 \\ reg[rs][1] & 0 \end{pmatrix}\begin{pmatrix} reg[rt][0] & reg[rt][1] \\ - & - \end{pmatrix}$$
$$= \begin{pmatrix} reg[rs][0] * reg[rt][0] & - \\ - & reg[rs][1] * reg[rt][1] \end{pmatrix}$$

ShiftLeft rd rt shamt (2-bit operands)

$$\begin{pmatrix} reg[rt] & - \\ - & reg[rt] \end{pmatrix}\begin{pmatrix} 2 & - \\ - & 4 \end{pmatrix} = \begin{pmatrix} reg[rt] * 2 & - \\ - & reg[rt] * 4 \end{pmatrix}$$

Figure 1. Assembly to Matrix Mappings

The tool is able to solve for these functions such that the extracted values from the result matrix follow the semantics of the original instruction.
Set Less Than rd rs rt is defined as:

$$reg[rd] \ := \ (reg[rs] < reg[rt]) \ ? \ 1 \ : \ 0$$

Our matrix equivalent in Figure 1 extracts the value for $reg[rd]$ from the sign bit of the $(0, 0)$ element.
Add rd rs rt is defined as:

$$reg[rd] \ := \ reg[rs] + reg[rt]$$

We similarly extract the value for $reg[rd]$ from the $(0, 0)$-positioned element. Other arithmetic operations like multiplication and subtraction follow from this by some combination of flipping the register positions in the matrix, zeroing or negating a constant. Prior work in quantum computing has shown that arbitrary Boolean algebra can be performed as a matrix multiplication [4]. Our program synthesis technique discovered the *And* operation in Figure 1 where we assumed the ability of the hardware to form a bit-serial value from a single register. For brevity, we only show a 2-bit operand example, but note that by adding additional bits down the first row for *reg[rs]* and first column for *reg[rt]*, then this supports arbitrarily wide operands. Each element on the diagonal of the result matrix corresponds to a bit in the writeback value. A bit-shifting instruction can be computed by enumerating all the possibilities with control logic selecting the desired output. *ShiftLeft rt 2* is shown as the final entry in Figure 1 for a 2-bit machine. We rely on the ability of the hardware to extract the corresponding element off the diagonal or add an additional multiplication by a vector of all 1s if hardware support is better for extracting from a vector. This can be trivially widened for realistic word sizes.

3.1 Exploiting Instruction-Level Parallelism

The "don't care" elements in the examples presented thus far enable our compiler to fit instructions without data dependencies into the same matrix multiplication. If the extraction from the result matrix step includes the proper control logic (either embedded into the matrix itself or as a separate, programmable component), arbitrary instructions can be executed simultaneously. Because of the cascading nature of the matrix multiplication, some sequences of data-dependent instructions can be chained together in a single operation. A simple example is an n-element accumulating sum which can be computed on single row and column of an $n \times n$ matrix multiplication.

4 Paths for Optimization

It is clear that while possible, structuring an application as a sequence of 2×2 matrix multiplications is unlikely to yield benefits over traditional CPUs. We are exploring methods to combine these operations into larger matrices and execute several instructions and paths simultaneously. Additionally, we are targeting tighter integration of the accelerators into a core's pipeline where we can enable more useful, general-purpose instructions that don't map well to matrix multiplications.

4.1 Equality Saturation

By compiling instructions into sequences of matrix multiplications, we form linear algebraic equations. We can then optimize a program through rewriting according to algebraic rules. Equality saturation is one technique, used for program optimization and verification, that is well-suited for exploring large term spaces populated by rewrite rules in a space efficient data structure called an *e-graph* [11]. This direction builds off of a large corpus of related work for optimizing programs that map into algebraic theories [10, 15, 16, 18], using the equality saturation engine egg [17].

4.2 Speculative Execution and Predication

Pre-existing vector engines like AVX-512 make heavy use of predication to perform state-updates on only specific elements [6]. By taking advantage of the highly-parallel nature of matrix multiplication computations and then only updating state once the branching condition is resolved, we can execute multiple control flow streams in parallel. In terms of the matrix multiplication language, one could execute each branch as a block-diagonal multiplication followed by a *predicated extraction* operation. If you had two instruction streams under a branch represented in the language of matrix multiplications as $A \times B$ and $C \times D$, then you could represent the simultaneous execution as the following:

$$\begin{pmatrix} A & 0 \\ 0 & C \end{pmatrix}\begin{pmatrix} B & 0 \\ 0 & D \end{pmatrix} = \begin{pmatrix} AB & 0 \\ 0 & CD \end{pmatrix}$$

References

[1] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

[2] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks* 30 (1998), 107–117. http://www-db.stanford.edu/~backrub/google.html

[3] Esolangs. 2022. Subleq. https://esolangs.org/wiki/Subleq.

[4] Hassan Hajjdiab, Ashraf Khalil, and Hichem Eleuch. 2023. Q-Map: Quantum Circuit Implementation of Boolean Functions. arXiv:2303.00075 [quant-ph]

[5] Kuan-Chieh Hsu and Hung-Wei Tseng. 2021. Accelerating Applications Using Edge Tensor Processing Units. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 56, 14 pages. https://doi.org/10.1145/3458817.3476177

[6] Intel. 2023. Intrinsics for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Instructions. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-9/intrinsics-for-avx-512-instructions.html.

[7] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[8] Farhad Mavaddat and Behrooz Parhami. 1988. URISC: the ultimate reduced instruction set computer. *International Journal of Electrical Engineering Education* 25, 4 (1988), 327–334.

[9] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press. https://doi.org/10.1017/CBO9780511976667

[10] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

[11] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

[12] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[13] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. *SIGPLAN Not.* 49, 6 (June 2014), 530–541. https://doi.org/10.1145/2666356.2594340

[14] UCSBarchlab. [n. d.]. OpenTPU. https://github.com/UCSBarchlab/OpenTPU.

[15] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

[16] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 12 (jul 2020), 1919–1932. https://doi.org/10.14778/3407790.3407799

[17] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

[18] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper_files/paper/2021/file/65ded5353c5ee48d0b7d48c591b8f430-Paper.pdf

[19] Amir Yazdanbakhsh, Berkin Akin, and Kiran K Seshadri. 2021. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. https://arxiv.org/abs/2102.10423.