Matrix Multiplication and Performance

A Short Journey into High Performance Computing (HPC)

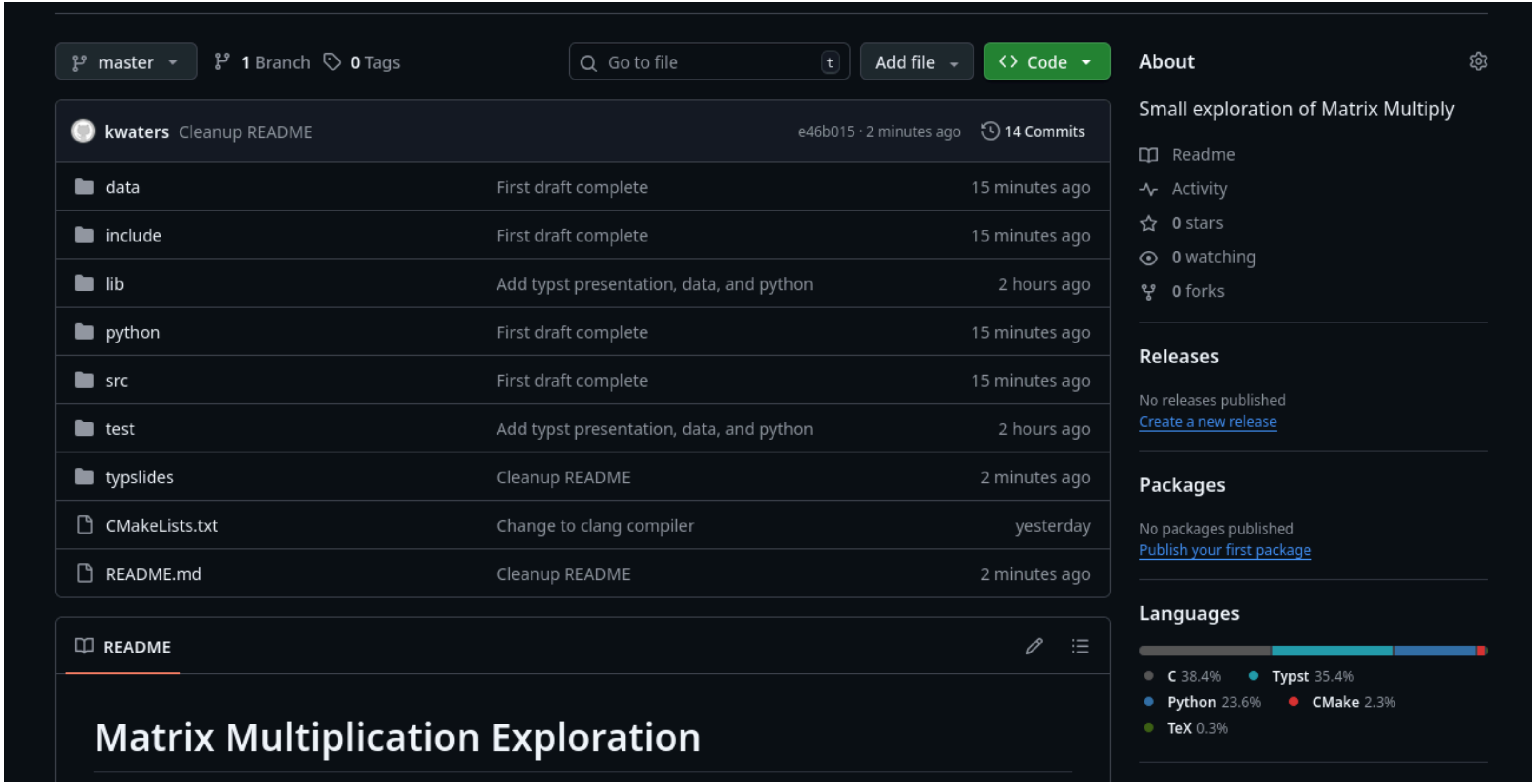
Kevin Waters

Matrix Multiplication and Performance

The Time I Accidentally Bested NumPy

Kevin Waters

- The code and talk for this presentation can be found here.

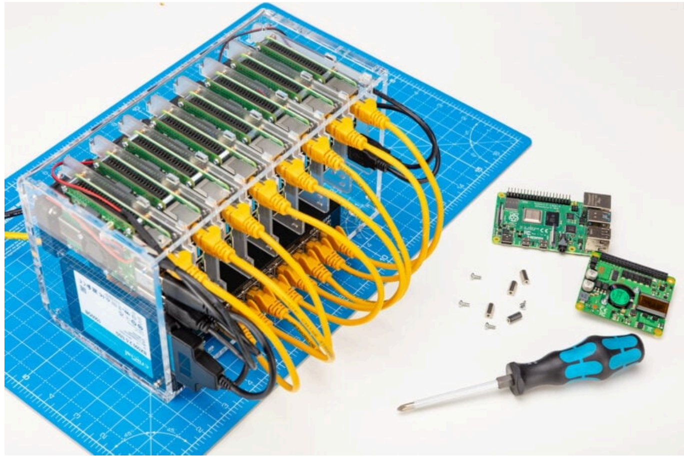


Contents

What is High Performance Computing (HPC)?

What is HPC?

- AI (Training/Inference)?
- Large-scale distributed memory computations?
- Distributed and scalable web services?
- Performance-aware programming:[1]
  - x86 aware?
  - Platform aware (CPU vs. GPU)?
  - Instruction set architecture (ISA) aware?
  - Cache-size aware?



Raspberry Pi cluster[2]

[1] Term taken from Casey Muratori https://www.computerenhance.com/p/welcome-to-the-performance-aware

[2] https://www.raspberrypi.com/tutorials/cluster-raspberry-pi-tutorial/

Matrix Multiplication

Why do we Care?

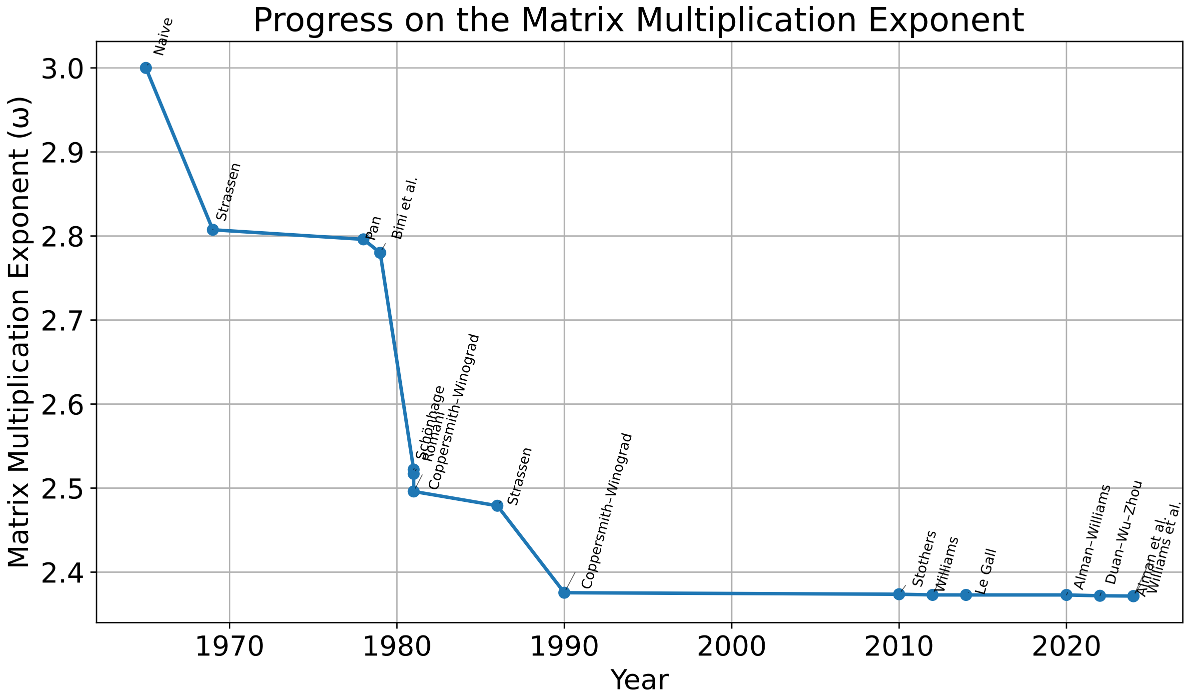
- A few examples of applications (Where is linear algebra used today?):
  - **Finite element analysis** - Aerospace, automotive, material's properties...
  - **Electronic structure theory** - Density Functional Theory, Hartree-Fock++...
  - **Machine learning/data science** - Data analysis, pattern recognition, neural nets...
  - **Genetics** - genotype distribution
  - **Solving (partial) differential equations**... and much more

$$\boldsymbol{AB} = \boldsymbol{C}$$

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \dots & a_{1n} \\ a_{20} & a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \\ a_{n0} & a_{n1} & a_{n3} & \dots & a_{nn} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} & b_{02} & \dots & b_{0n} \\ b_{10} & b_{11} & b_{12} & \dots & b_{1n} \\ b_{20} & b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \vdots & \ddots & \\ b_{n0} & b_{n1} & b_{n3} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} & c_{02} & \dots & c_{0n} \\ c_{10} & c_{11} & c_{12} & \dots & c_{1n} \\ c_{20} & c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \vdots & \ddots & \\ c_{n0} & c_{n1} & c_{n3} & \dots & c_{nn} \end{pmatrix}$$

Where:

$$c_{ij} = \sum a_{ij} b_{kj}$$



Progress of Computation Complexity of Matrix Multiply[3]

[3]Sourced from https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication

Naive Computational Complexity

- Naive Matrix multiplication's computational complexity is $\theta(n^3)$
  - Others scale better, but there are trade-offs
- What does that mean?
  - 2x2 results in 8 multiplication steps and 4 addition steps:

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}b_{00}+a_{01}b_{10} & a_{00}b_{01}+a_{01}b_{11} \\ a_{10}b_{00}+a_{11}b_{10} & a_{10}b_{01}+a_{11}b_{11} \end{pmatrix}$$

  - For the general case the number of operations is given by the following:

$$2n^3 + n^2$$

  - $2n^3$ multiply operations.
  - n^2 addition operations.

Matrix Multiplication in Python (Easy Part)

What can Python do?

Python Benchmark

```
❯ python python_plain.py
1024x1024 matrix multiply...
Int          Time: 80.576309 seconds
Float        Time: 97.653700 seconds
```

- There are many things working against python, just-in-time-compilation (JIT), arbitrary size integers…

What can Python (NumPy) do?

- NumPy is a highly-tuned library where typically expensive functions are implemented in compiled languages such as C/C++ or FORTRAN. (BLAS & LAPACK)

Python (NumPy) Benchmark

```
❯ python python_numpy.py
1024x1024 matrix multiply...
Int32      Time: 1.895825 seconds
Int64      Time: 1.884659 seconds
Float      Time: 0.005361 seconds
Double     Time: 0.010751 seconds
```

- These results are interesting...

Matrix Multiplication in C (Hard Part)

Some Quick Assumptions

- All matrices are square.
- All matrices sizes are a power of 2.
- Matrices are populated with ints (32-bits) randomly distributed [−5,5].
- For benchmarking, one warm-up was followed by five trials (times are per trials).
- To count cycles, the following x86 instruction was used “rdtsc()” (Time Stamp Counter).
- Time stamps were called using clock_gettime().
- Compile flags for all timed runs:
  - *-Wall -O3 -march=native -funroll-loops*
- Work per cycle was calculated using the following:

$$\frac{\text{Work Required}}{\text{Clock Cycle}} = \frac{3n^3+n^2}{\#\ \text{cycles\_elapsed}}$$

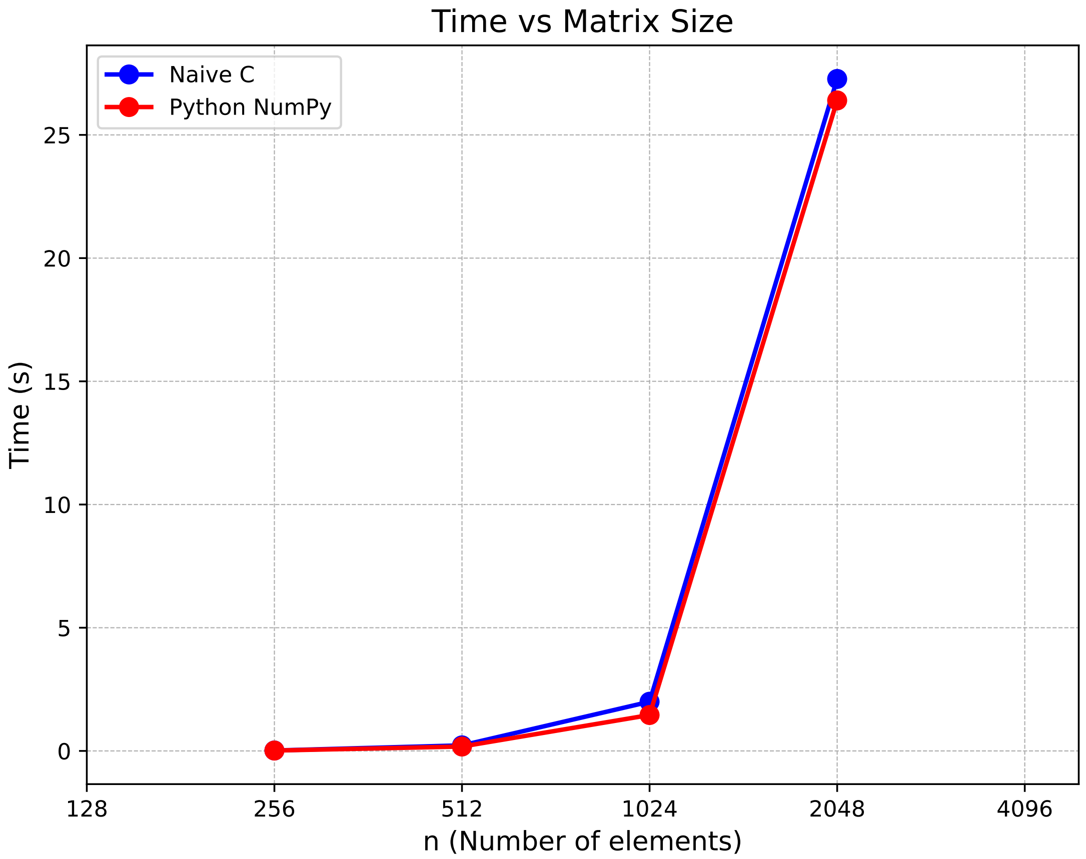
Naive Matrix Multiply

Simple Matrix Multiply

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            result[i * n + j] += matrix1[i * n + k] * matrix2[k * n + j];
        }
    }
}
```

- Readable, simple, and slow.

Naive Matrix Multiply



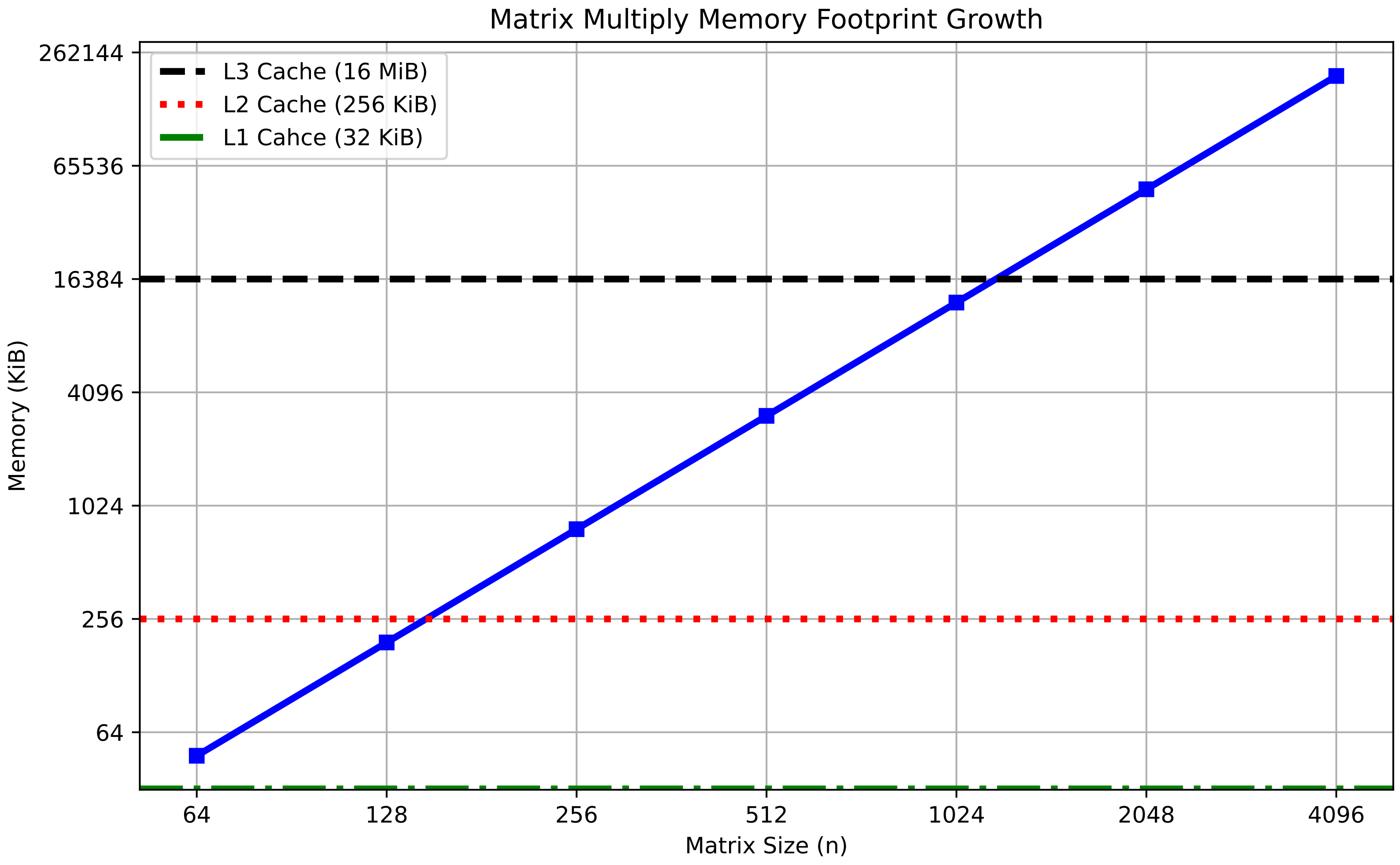
Problem Size Growth (Memory Footprint)

- Assuming 32-bit integers or 32-bit floats

n	Elements (n^2)	Memory/Matrix (KiB)	Total Memory (KiB)
64	4096	16	48
128	16384	64	192
256	65536	256	768
512	262144	1024	3072
1024	1048576	4096	12288
2048	4194304	16384	49152
4096	16777216	65536	196608

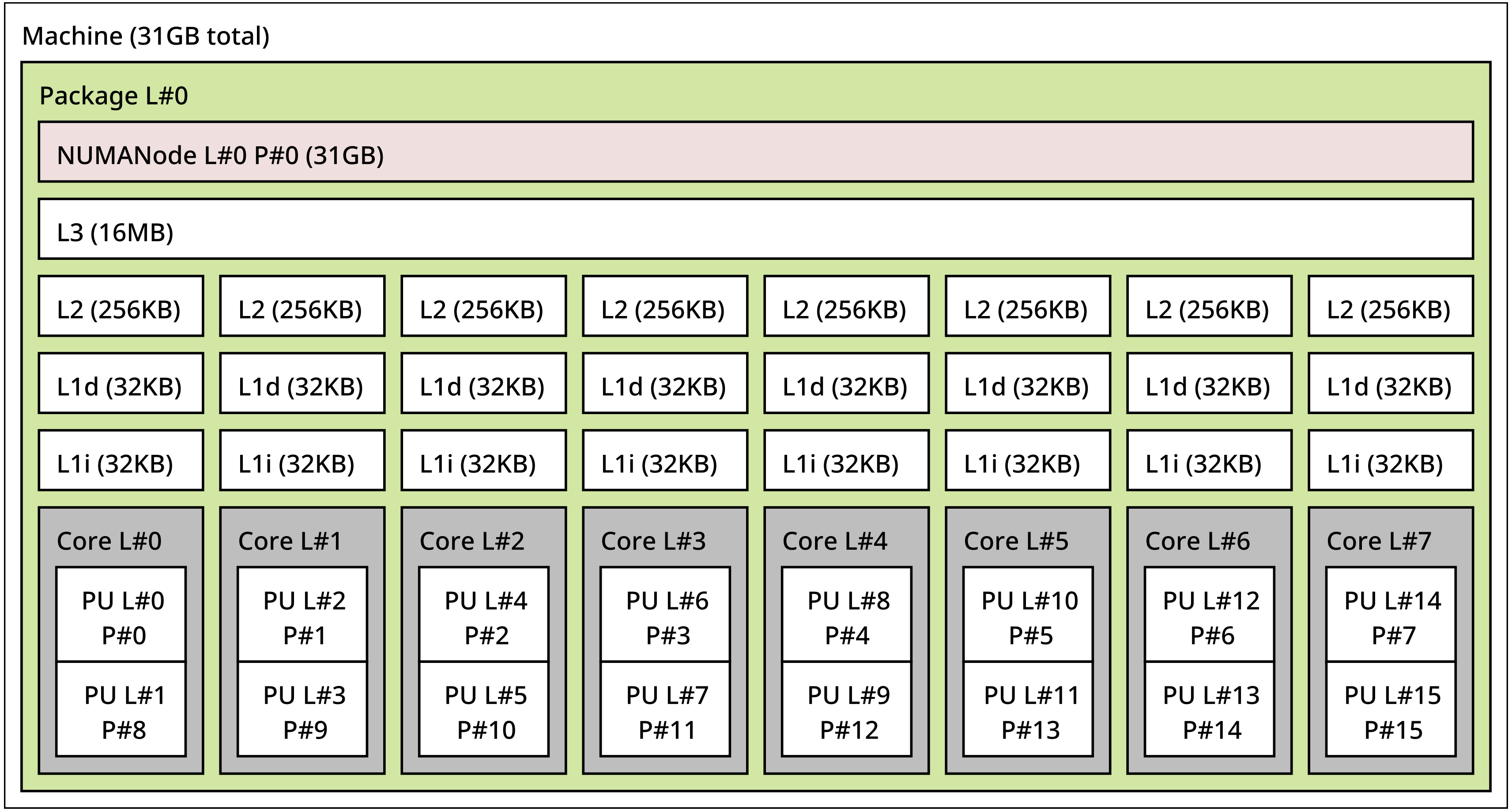
- Total memory is 3x the memory per matrix.

Memory Footprint Revisted



Memory requirement for three nxn matrices.

Skylake Topology



8-core client Skylake topology ($ lstopo)

Skylake Topology (Highlights)

- One NUMA domain
- One 16 MiB shared L3 cache
- Individual 256 KiB L2 cache
- Individual 32 KiB instruction and data caches
- Two logical cores per physical core (8 physical, 16 logical)

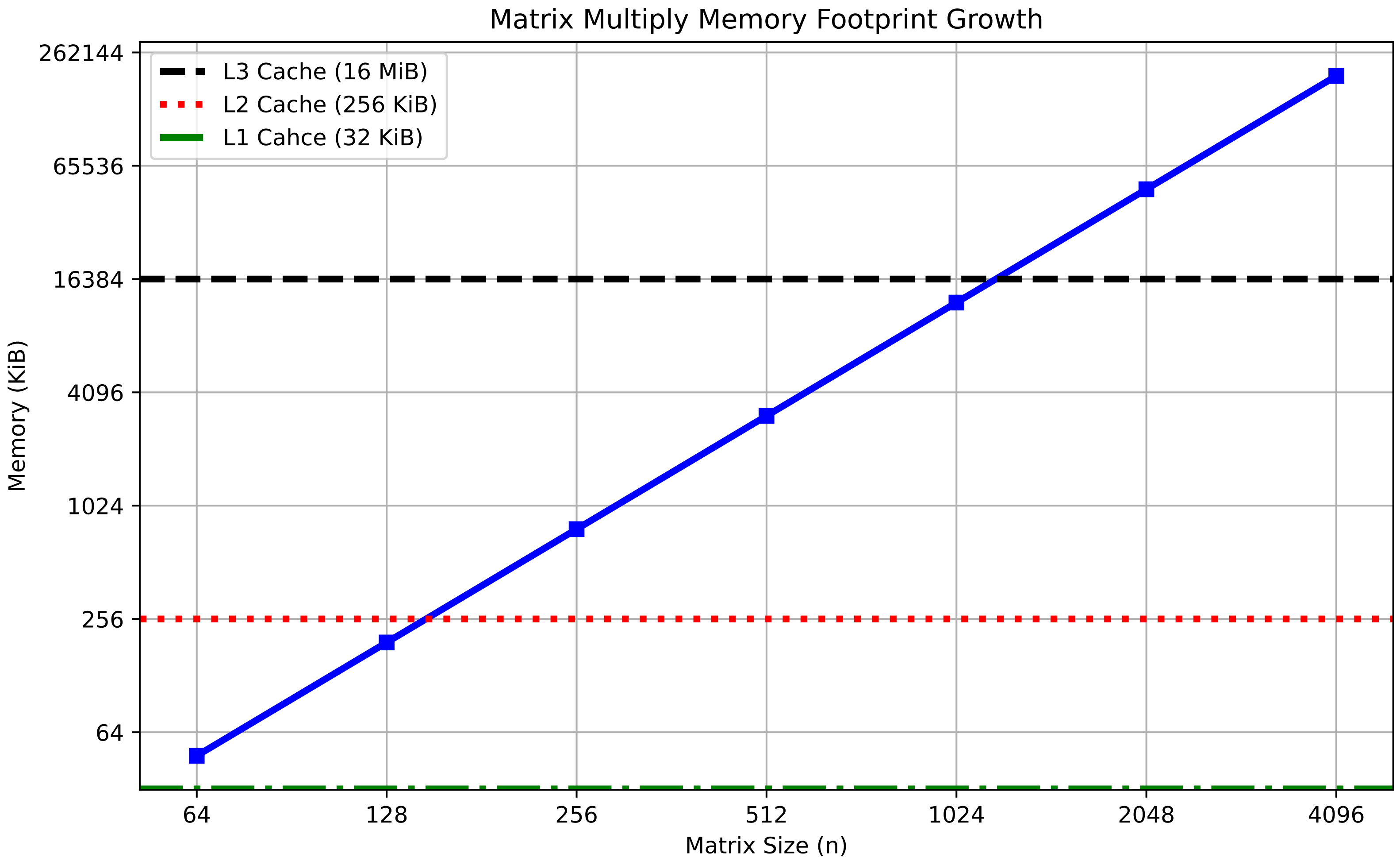
Latency Comparison Numbers (Everything is a Cache) (~2012)

L1 cache reference	0.5 ns			
Branch mispredict	5 ns			
L2 cache reference	7 ns			14x L1 cache
Mutex lock/unlock	25 ns			
Main memory reference	100 ns			20x L2 cache, 200x L1 cache
Send 1K bytes over 1 Gbps network	10,000 ns	10 us		
Read 4K randomly from SSD	150,000 ns	150 us		1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	250 us		
Read 1 MB sequentially from SSD	1,000,000 ns	1,000 us	1 ms	1GB/sec SSD, 4X memory
Disk seek	10,000,000 ns	10,000 us	10 ms	10x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	20,000 us	20 ms	80x memory, 20X SSD

Latencies to generate intuition for the cost of an operation.[4]

[4]Originally by Peter Norvig: http://norvig.com/21-days.html#answers

Memory Footprint Revisted



Memory requirement for three nxn matrices.

Whole Program Performance Metrics

Intermediate Sum

```
❯ perf stat -e cycles,instructions,cache-references,cache-misses ./build/driver 2048 2048 5
n, trials, req. memory (KiB), time/trial (s), work/cycle, Read Bandwidth (GiB/s)
s2048, 5, 49152, 21.019992, 0.215482, 0.000297

 Performance counter stats for './build/driver 2048 2048 5':

   603,229,008,482      cycles:u
   156,197,169,903      instructions:u                   #    0.26  insn per cycle
   100,483,589,712      cache-references:u
     7,718,024,466      cache-misses:u                   #    7.68% of all cache refs
```

- More targeted data can be found with PAPI (Performance Application Programming Interface).

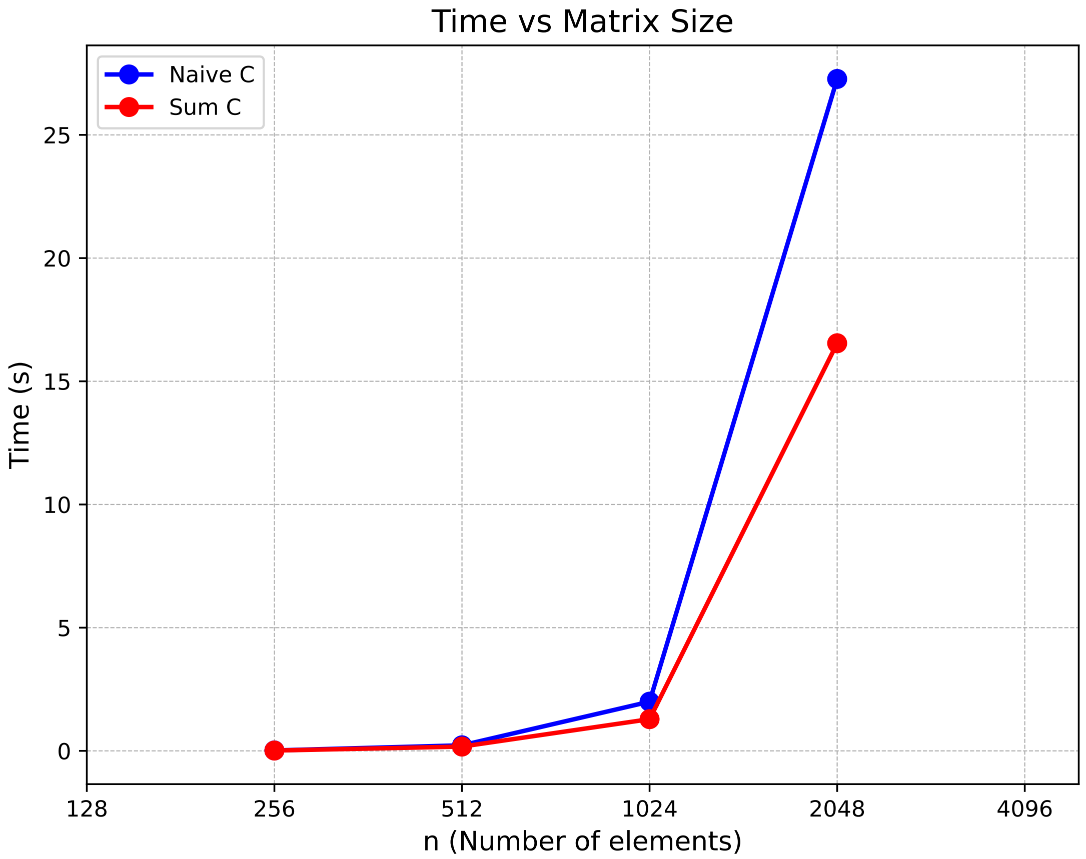
Intermediate Sum Matrix Multiply

Intermediate Sum

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    int sum = 0;
    for (int k = 0; k < n; k++) {
      sum = matrix1[i * n + k] * matrix2[k * n + j] + sum;
    }
  result[i * n + j] = sum;
  }
}
```

- Allows for sum to stay in registers requiring less fetching from memory, a little bit faster.

Intermediate Sum Matrix Multiply



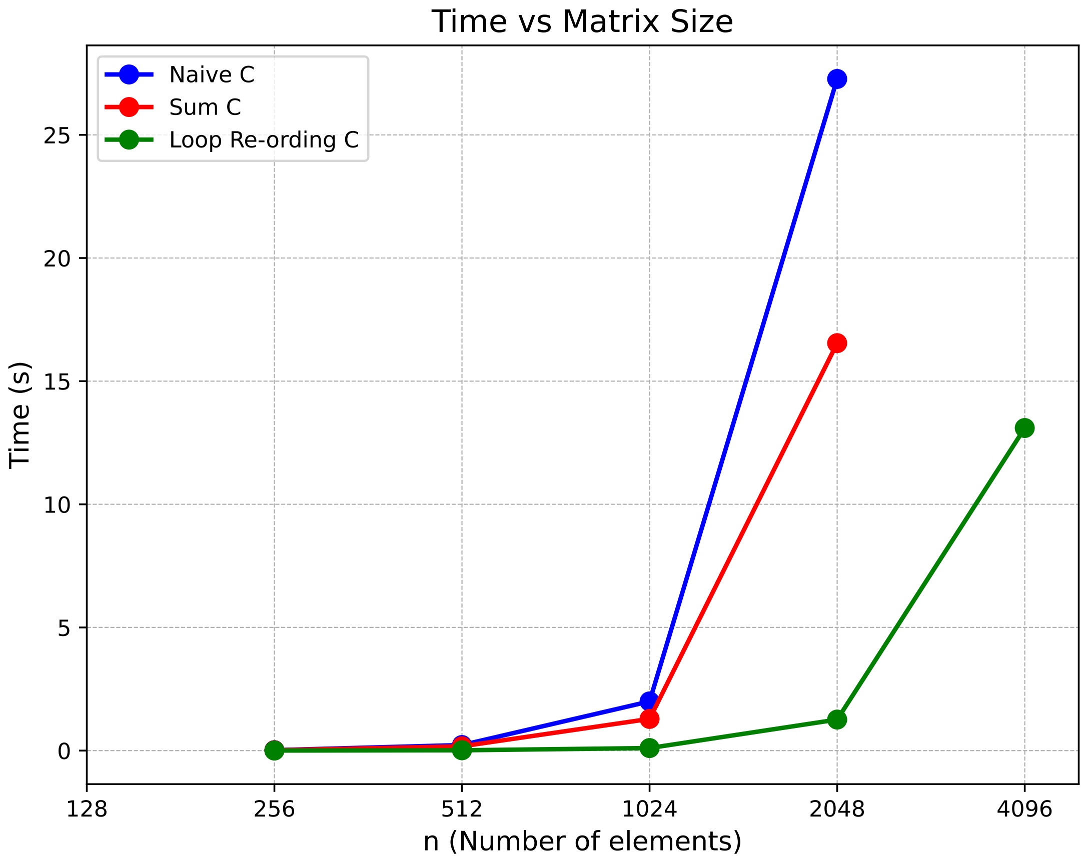
Loop Re-ording Matrix Multiply

Loop Re-ording Sum

```
for (int i = 0; i < n; i++) {
  for (int k = 0; k < n; k++) {
    for (int j = 0; j < n; j++) {
      result[i * n + j] += matrix1[i * n + k] * matrix2[k * n + j];
    }
  }
}
```

- Re-ording the last two loops (j with k) enabling better caching behavior.

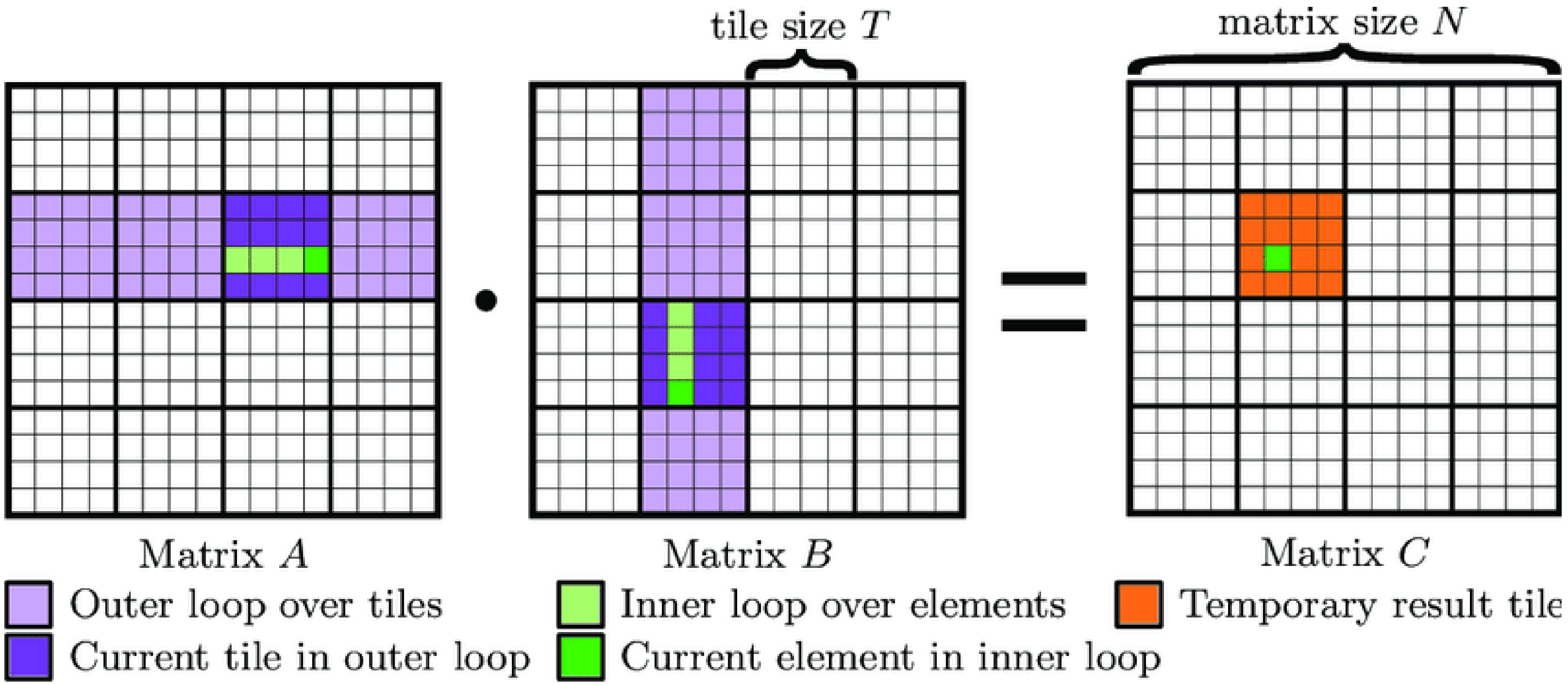
Loop Re-ordering Matrix Multiply



Block Matrix Multiply

Blocking

```
for (int ii = 0; ii < n; ii+= BLOCK_SIZE) {
    for (int kk = 0; kk < n; kk+= BLOCK_SIZE) {
      for (int jj = 0; jj < n; jj+= BLOCK_SIZE) {
        int limit_i = ((ii + BLOCK_SIZE) < n) ? (ii + BLOCK_SIZE) : n;
        int limit_j = ((jj + BLOCK_SIZE) < n) ? (jj + BLOCK_SIZE) : n;
        int limit_k = ((kk + BLOCK_SIZE) < n) ? (kk + BLOCK_SIZE) : n;
        for (int i = ii; i < limit_i; ++i) {
          for (int k = kk; k < limit_k; ++k) {
            int ki = i * n + k;
            for (int j = jj; j < limit_j; j++) {
              result[i * n + j] += matrix1[ki] * matrix2[k * n + j];
            }
          }
        }
      }
    }
  }
```



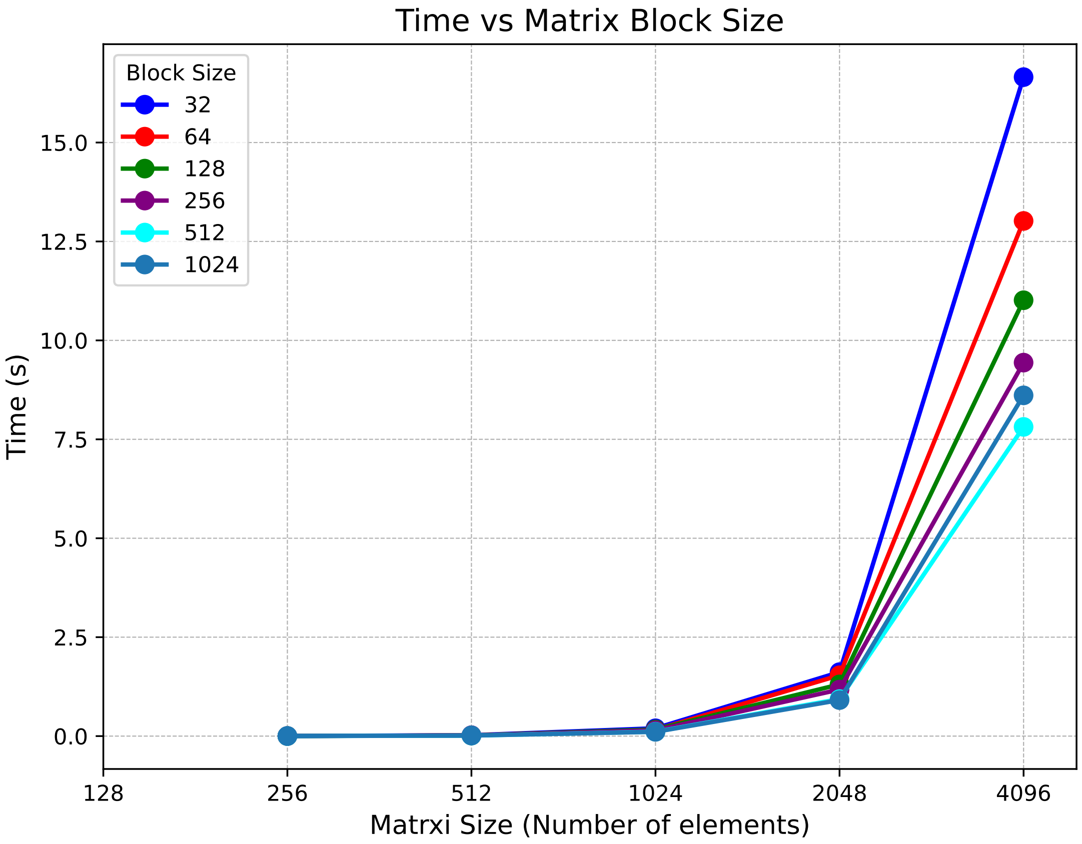
GEMM tiling or BLOCK_SIZE.[5]

[5] Mathhes et. al. Tuning and Optimization for a Variety of Many-Core Architectures Without Changing a Single Line of Implementation Code Using the Alpaka Library (2017)

Block Matrix Multiply

- This is where optimizations start becoming unpleasant as it is not hardware agnostic, however we are not using intrinsics yet!
- The BLOCK_SIZE variable is a compile time constant, requiring the library to be recompiled.
- We will recompile until we find an optimal BLOCK_SIZE value.

Loop Re-ordering Matrix Multiply BLOCK_SIZE



Whole Program Performance Metrics

BLOCK_SIZE 512

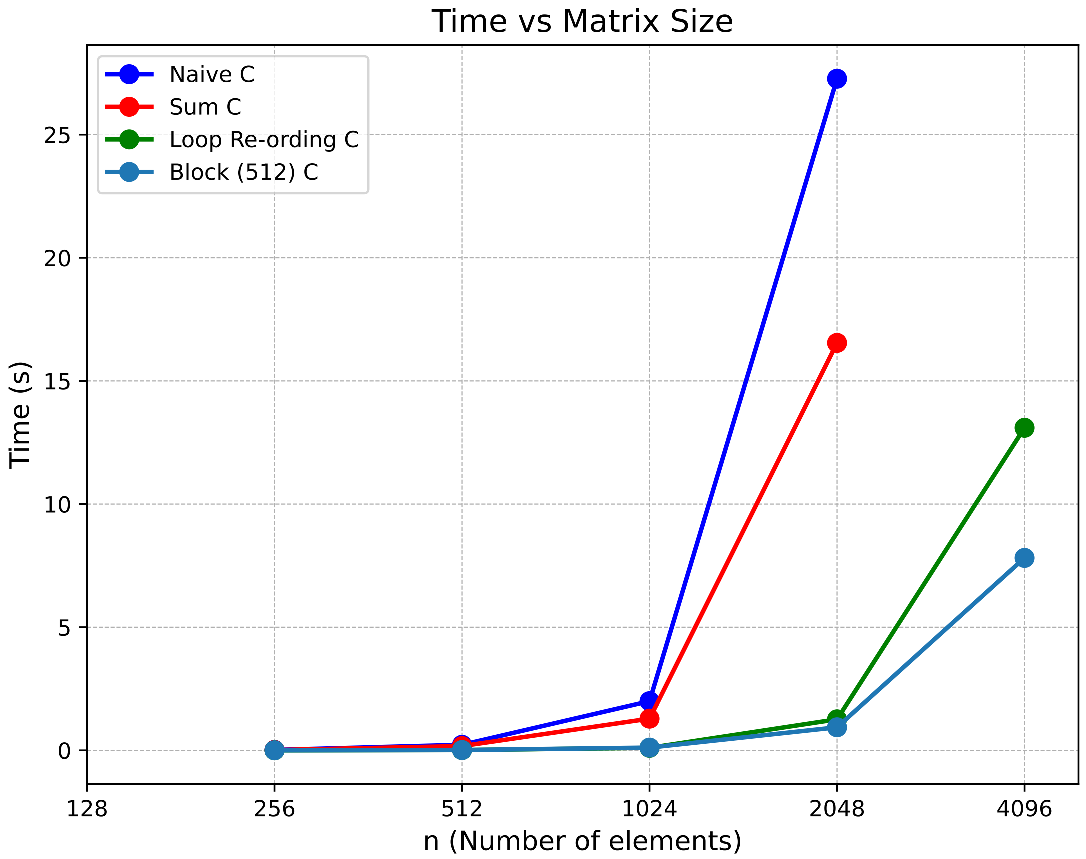
```
❯ perf stat -e cycles,instructions,cache-references,cache-misses ./build/driver 2048 2048 5
2048, 5, 49152, 0.792592, 5.714694, 0.007886

Performance counter stats for './build/driver 2048 2048 5':

  22,990,040,828      cycles:u
  26,574,845,684      instructions:u                   #    1.16  insn per cycle
   7,550,989,191      cache-references:u
     121,521,554      cache-misses:u                   #    1.61% of all cache refs
```

- Naive L3 cache misses was 7.68%, PAPI would give better resolution. There is still better cache performance possible.

All Methods


Time vs Matrix Size
Naive C
Sum C
Loop Re-ording C
Block (512) C
Time (s)
0
5
10
15
20
25
128
256
512
1024
2048
4096
n (Number of elements)

What CPU do I have?

Checking the Hardware

```
❯ for i in Architecture "CPU(s):" "Model name" Thread Socket "NUMA node(s)"; do
lscpu | grep "$i" | grep -v "node0"; done

Architecture:                         x86_64
CPU(s):                               16
Model name:                           Intel(R) Core(TM) i7-10700KF CPU @ 3.80GHz
Thread(s) per core:                   2
Socket(s):                            1
NUMA node(s):                         1
```

- x86[6] SIMD[7] extensions:
  - **SSE (Streaming SIMD Extensions)** - 128-bit floating point registers
  - **SSE2** - 128-bit doubles and integer registers
  - **AVX (Advanced Vector Extensions)** - 256-bit floating/double point registers
  - **AVX2** - 256-bit integer SSE instructions
  - **AVX512** - 512 bit registers!
- **FMA(Fused Multiply-Add)** - Exists in AVX, AVX512, but only for floats and doubles.

[6] ARM has different names for everything

[7] Single Instruction, Multiple Data

Checking the Architecture and ISA

```
❯ cat /sys/devices/cpu/caps/pmu_name
skylake

❯ for isa in sse sse2 avx avx2 avx512 fma; do grep -q "$isa" /proc/cpuinfo && echo "$isa  1" ||
echo "$isa  0"; done
sse  1
sse2  1
avx  1
avx2  1
avx512  0
fma  1
```

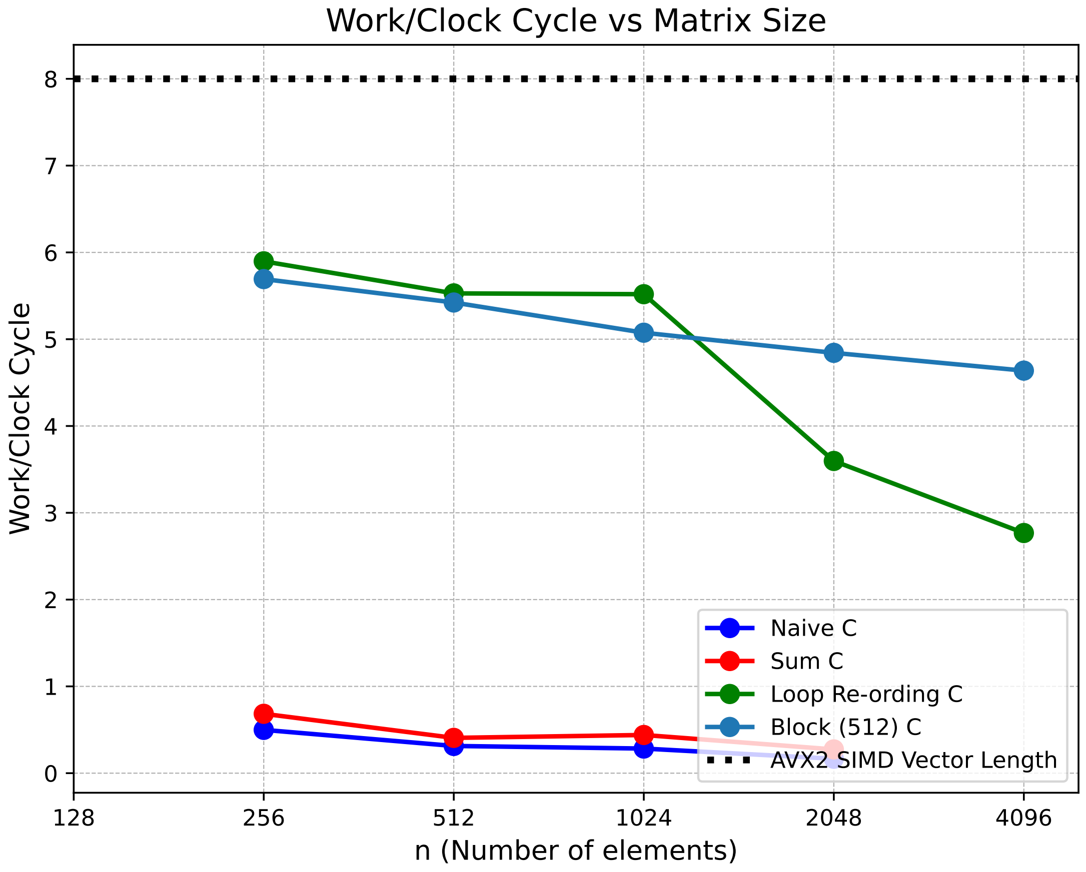
Checking for SIMD (AVX2)

```
10b0: c4 c2 7d 40 4c 8a a0  vpmulld -0x60(%r10,%rcx,4),%ymm0,%ymm1
10b7: c4 c2 7d 40 54 8a c0  vpmulld -0x40(%r10,%rcx,4),%ymm0,%ymm2
10be: c4 c2 7d 40 5c 8a e0  vpmulld -0x20(%r10,%rcx,4),%ymm0,%ymm3
10c5: c4 c2 7d 40 24 8a     vpmulld (%r10,%rcx,4),%ymm0,%ymm4
10cb: c4 c1 75 fe 4c 88 a0  vpaddd -0x60(%r8,%rcx,4),%ymm1,%ymm1
10d2: c4 c1 6d fe 54 88 c0  vpaddd -0x40(%r8,%rcx,4),%ymm2,%ymm2
10d9: c4 c1 65 fe 5c 88 e0  vpaddd -0x20(%r8,%rcx,4),%ymm3,%ymm3
10e0: c4 c1 5d fe 24 88     vpaddd (%r8,%rcx,4),%ymm4,%ymm4
10e6: c4 c1 7e 7f 4c 88 a0  vmovdqu %ymm1,-0x60(%r8,%rcx,4)
10ed: c4 c1 7e 7f 54 88 c0  vmovdqu %ymm2,-0x40(%r8,%rcx,4)
10f4: c4 c1 7e 7f 5c 88 e0  vmovdqu %ymm3,-0x20(%r8,%rcx,4)
10fb: c4 c1 7e 7f 24 88     vmovdqu %ymm4,(%r8,%rcx,4)
```

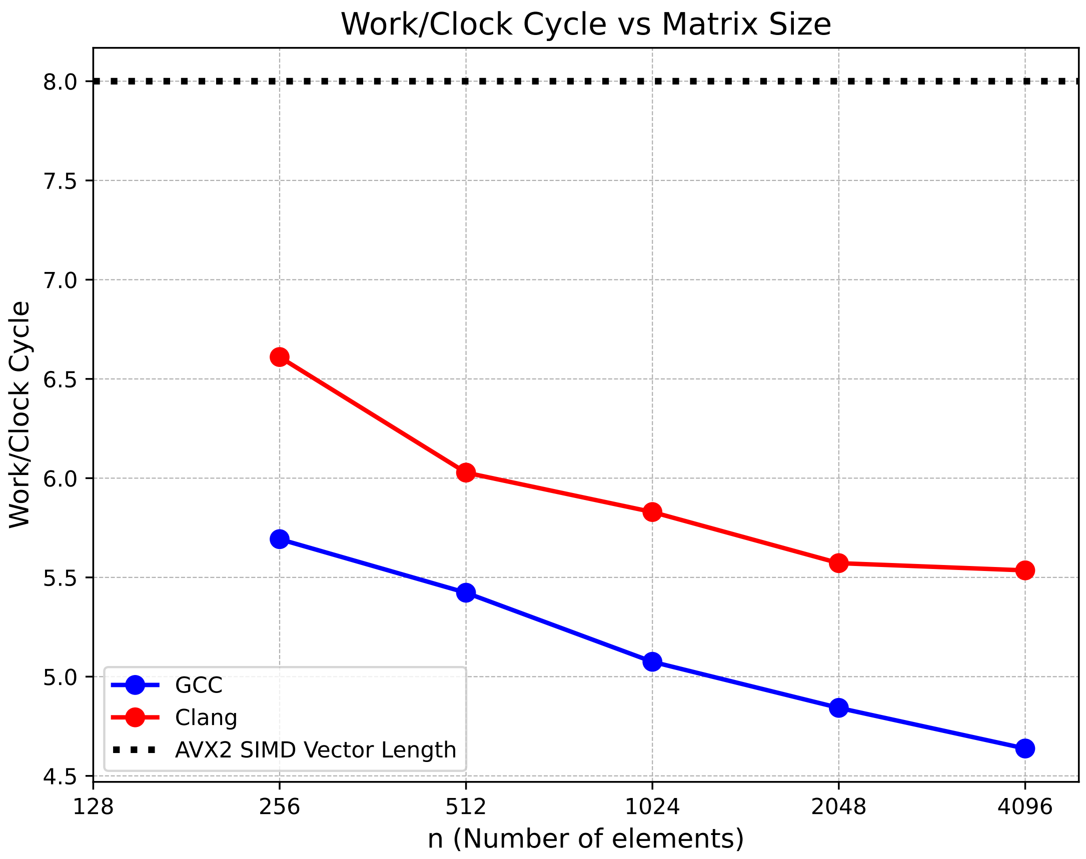
- Excerpt from multiplication library (appears some pipelining is going on!)[8].

[8] More details can be found in the Intel Intrinsics Guide

Work per Clock Cycles (Normalized Data)



GCC vs. Clang


Work/Clock Cycle vs Matrix Size
Work/Clock Cycle
n (Number of elements)
8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
128
256
512
1024
2048
4096
GCC
Clang
AVX2 SIMD Vector Length

Long Journey (Int32)

	Runtime (s)		
Matrix size (n)	**Pure Python**	**Python (NumPy)**	**Blocked C**
256	0.24	0.02	0.001
512	1.79	0.18	0.01
1024	15.32	1.46	0.10
2048		26.40	0.81
4096			6.55

- Final results with the different run times. More optimization is possible for C and NumPy.

More Discussion

Integers, Floats, Bits, each Matter

- Libraries exists with hyper optimized floating-point matrix operations standardized by BLAS[9].
  - **ATLAS** - Automatically Tuned Linear Algebra Software
  - **OpenBLAS** - open-source CPU based BLAS
  - **rocBLAS** - AMD's GPUs version via ROCM
  - and many more[10]
- Sparsity may drive to different algorithms.
- If working with integers you may have to write your own kernels.
- If working with Boolean matrices they allow for new algorthms using look-up tables[11]

[9] Basic Linear Algebra Subprograms

[10] https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Implementations

[11] Method of Four Russians

Parallelization (Things we did not go over)

- **OpenMP/PThreads**
  - Using all cores on a socket/node
- **Simultaneous Multithreading (SMT/HyperThreading)**
  - Should it be used?
- **Non-Uniform Memory Access (NUMA)**
  - Even more levels to the memory subsystem
  - AMD's Core Complex (CCX) have made this harder
- **MPI/SHMEM**
  - Inter-node communication using remote direct memory access (RDMA)

Questions?