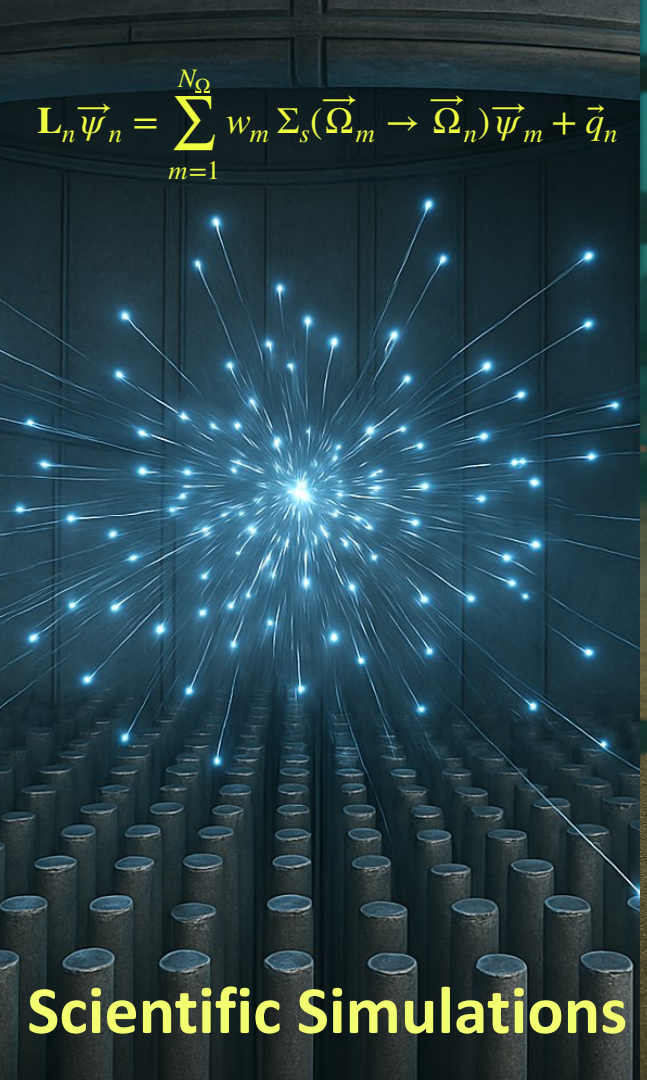
C++ now

2025

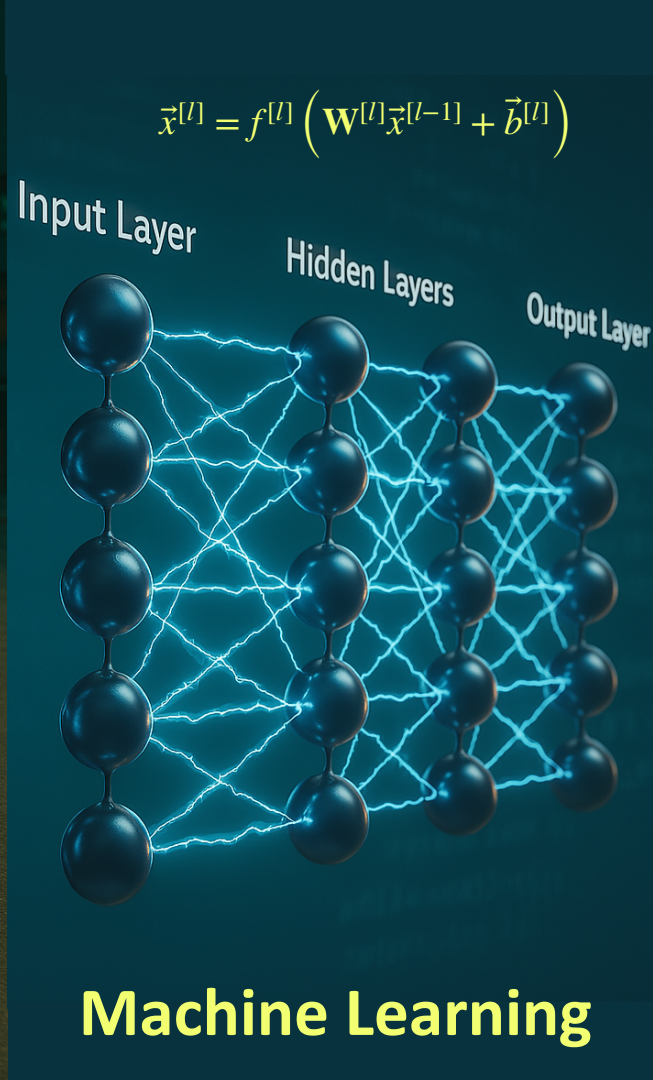
Achieving Peak Performance for Matrix Multiplication

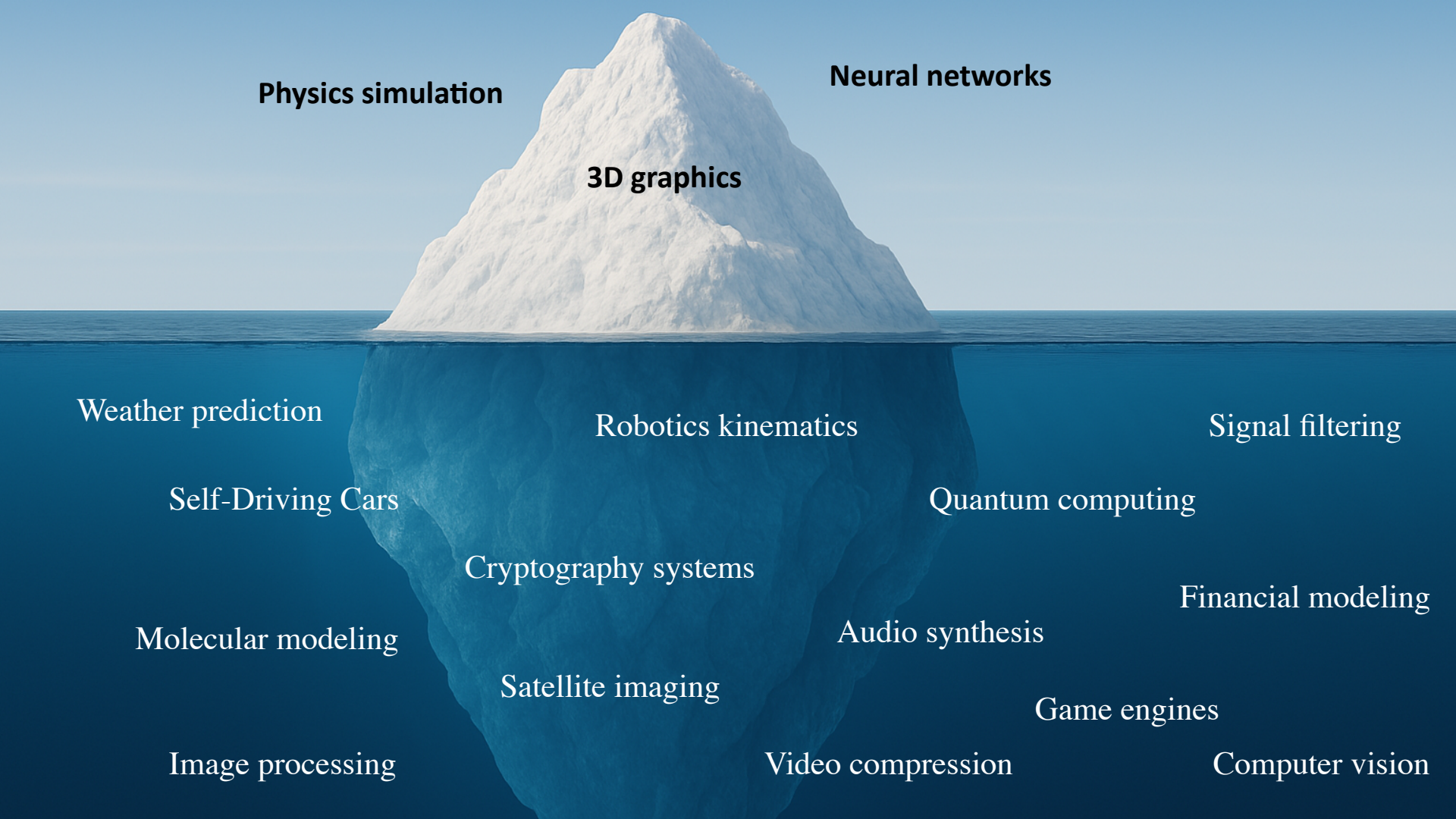
Aliaksei Sala

Why matrix multiplication?


$\mathbf{L}_n \vec{\psi}_n = \sum_{m=1}^{N_\Omega} w_m \Sigma_s(\vec{\Omega}_m \to \vec{\Omega}_n)\vec{\psi}_m + \vec{q}_n$
Scientific Simulations


$\mathbf{v}' = \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M} \cdot \mathbf{v}$
Computer Graphics

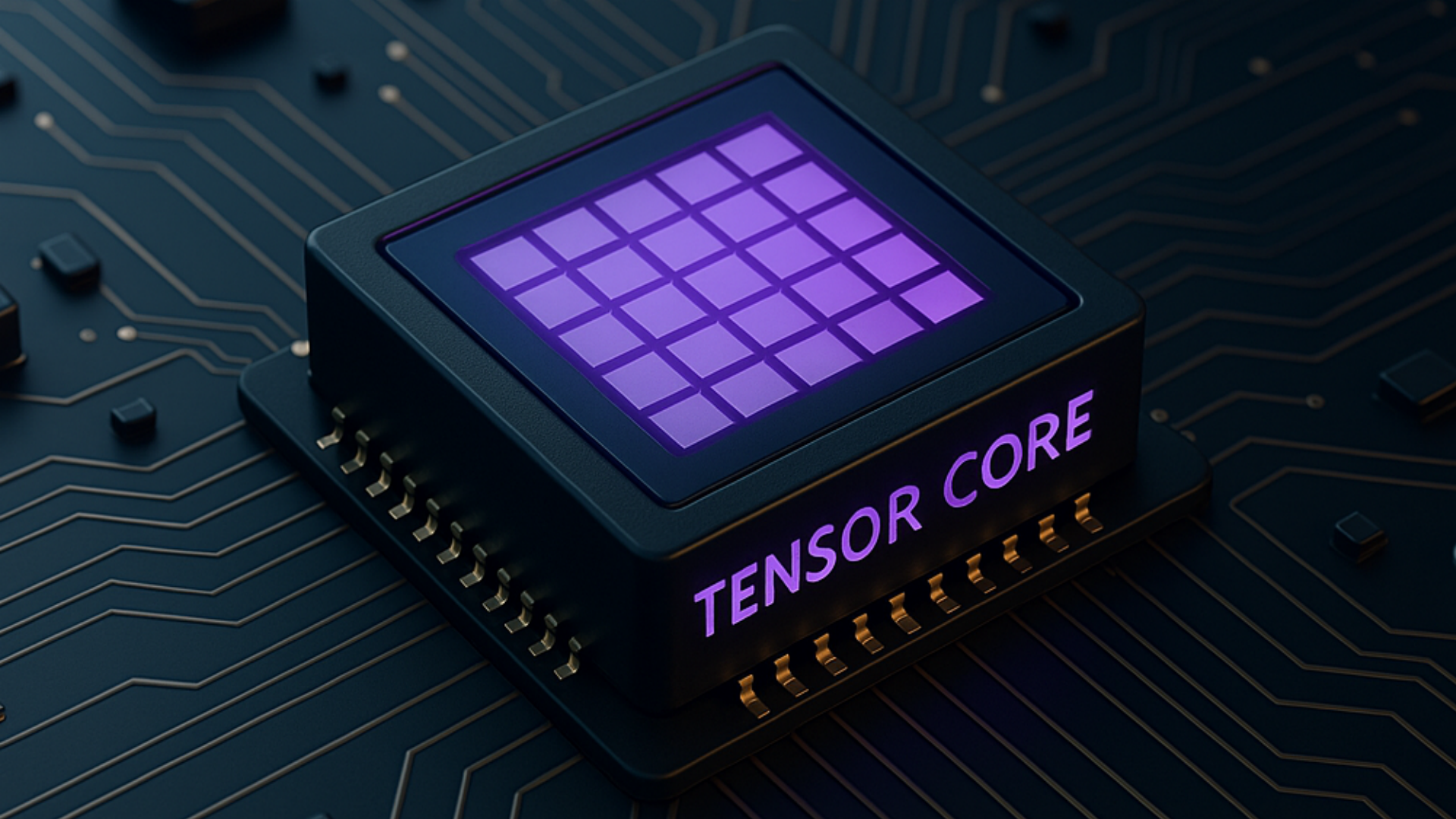

$\vec{x}^{[l]} = f^{[l]}\left(\mathbf{W}^{[l]}\vec{x}^{[l-1]} + \vec{b}^{[l]}\right)$
Input Layer
Hidden Layers
Output Layer
Machine Learning

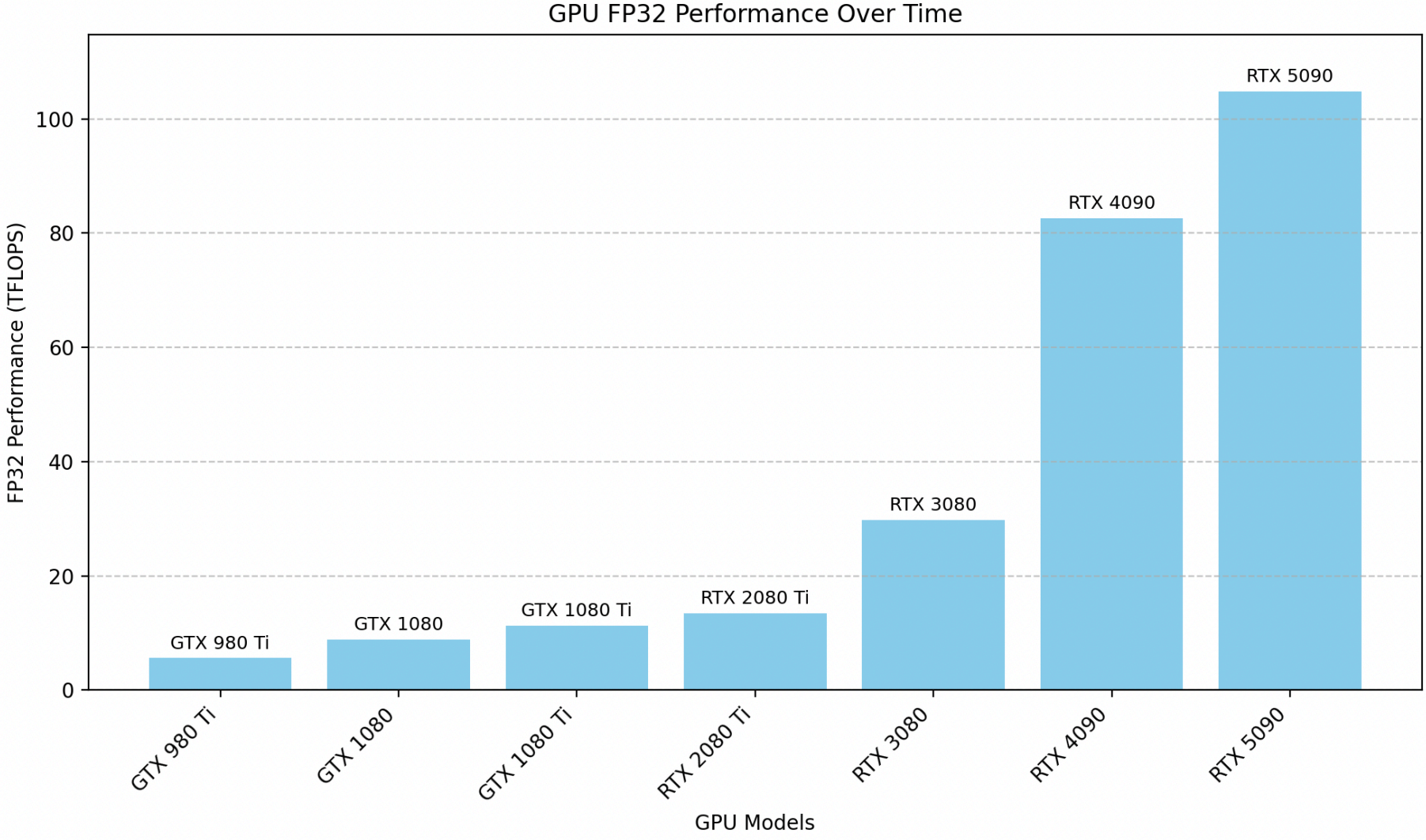

Physics simulation
Neural networks
3D graphics
Weather prediction
Robotics kinematics
Signal filtering
Self-Driving Cars
Quantum computing
Cryptography systems
Financial modeling
Molecular modeling
Audio synthesis
Satellite imaging
Game engines
Image processing
Video compression
Computer vision

Why peak performance?

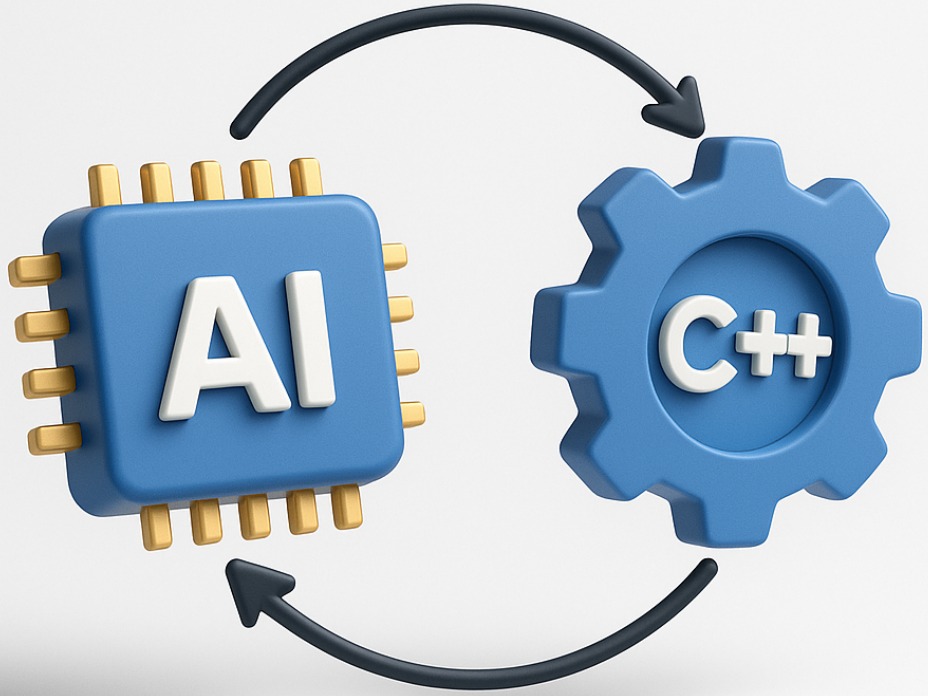
Year	AI Breakthrough	Model/Approach	Hardware Used	Matrix Multiply Perf.	Total Compute (FLOPs)
2012	*AlexNet*	Convolutional Neural Network	2× GeForce **GTX 580** GPUs	**~1.58 TFLOPS** per GPU (FP32)	~**3.16 TFLOPS** combined
2018	*BERT-Large* (NLP SOTA)	Transformer (bidirectional)	**64 TPU v2 chips** (16 Cloud TPU devices)	**~45 TFLOPS** per chip (bfloat16)	~**2.9 PFLOPS** total
2019	*AlphaStar*	Multi-agent RL (league training)	**16 TPU v3** per agent	**~90 TFLOPS** per TPU (bfloat16)	**Multi-petaflop** distributed training over 14 days
2020	*GPT-3* (175B params)	Transformer LM (few-shot learning)	**10,000 V100 GPUs**	**~125 TFLOPS** (FP16) per GPU	~**3×10^23 FLOPs** for training
2022	*PaLM* (540B params)	Transformer LM (dense, Pathways)	**6144 TPU v4 chips** (2× 3072-chip pods)	**~100+ TFLOPS** per chip (est. BF16)	***~0.6–1.2 EFLOPS*** combined; 2× TPU v4 pods, 57.8% utilization
2023	*GPT-4* (est. >1T params)	Transformer LM (multimodal)	**Thousands of A100 GPUs**	**156 TFLOPS** (BF16)	Estimated *multiple* ***EFLOPS*** total;

Rise of domain-specific architectures


TENSOR CORE


GPU FP32 Performance Over Time
FP32 Performance (TFLOPS)
0
20
40
60
80
100
GTX 980 Ti
GTX 1080
GTX 1080 Ti
RTX 2080 Ti
RTX 3080
RTX 4090
RTX 5090
GPU Models

Hardware-Software Co-design



Can we leverage the full capabilities of the CPU while staying within the bounds of standard C++?

Background context

Background context

- Familiarity with hardware fundamentals (SIMD, cache)
- Matrix is a std::vector<double> aligned on cpu cache line = 64 bytes.
- Elements are stored in row-major order.
- Dense matrix multiplication of square matrices with dimensions 2880 × 2880
- Size of the matrix > L3 cache size (63MB > 6 MB)

Flow of the Session

- C++ implementation
- Evaluate performance
- Review results (asm)
- Proceed to the next optimized version

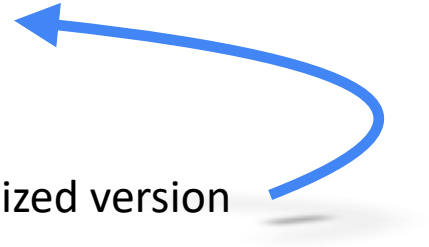
Flow of the Session

- C++ implementation
- Evaluate performance
- Review results (asm)
- Proceed to the next optimized version

- **Benchmark**
- **OS: Linux**
- **CPU: i5-6300HQ**
- **Compiler flags: -O3 -march=haswell -ffast-math**

Flow of the Session

- C++ implementation
- Evaluate performance
- Review results (asm)
- Proceed to the next optimized version



- **Benchmark**
- **OS: Linux**
- **CPU: i5-6300HQ**
- **Compiler flags: -O3 -march=haswell -ffast-math**

Peak performance.

$$FLOPS = cores \times \frac{cycles}{seconds} \times \frac{FLOPs}{cycles}$$

$$FLOPS = 4 \times 2.3GHz \times (8 * 2) = 147.2GFLOPS$$

Peak performance.

$$FLOPS = cores \times \frac{cycles}{seconds} \times \frac{FLOPs}{cycles}$$

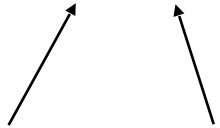
$$FLOPS = 4 \times 2.3GHz \times (8 * 2) = 147.2GFLOPS$$

Vectorization

Peak performance.

$$FLOPS = cores \times \frac{cycles}{seconds} \times \frac{FLOPs}{cycles}$$

$$FLOPS = 4 \times 2.3GHz \times (8 * 2) = 147.2GFLOPS$$



Vectorization Instruction throughput

Matrix multiplication implementations

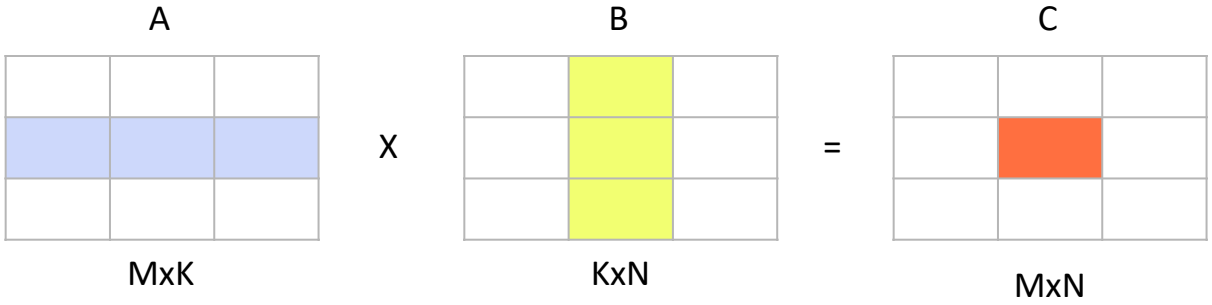
Naive

Naive

```
void matMulNaive(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto K = A.col();
    auto N = B.col();

    for(int i = 0; i < M; ++i){
        for(int j = 0; j < N; ++j){
            for(int k = 0; k < K; ++k){
                C(i, j) += A(i, k) * B(k, j);
            }
        }
    }
}
```

Naive

```
void matMulNaive(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto K = A.col();
    auto N = B.col();

    for(int i = 0; i < M; ++i){
        for(int j = 0; j < N; ++j){
            for(int k = 0; k < K; ++k){
                C(i, j) += A(i, k) * B(k, j);
            }
        }
    }
}
```



Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%

Changing Loop Order

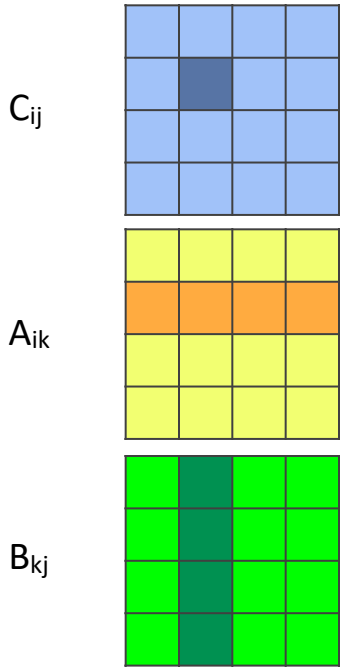
Changing Loop Order

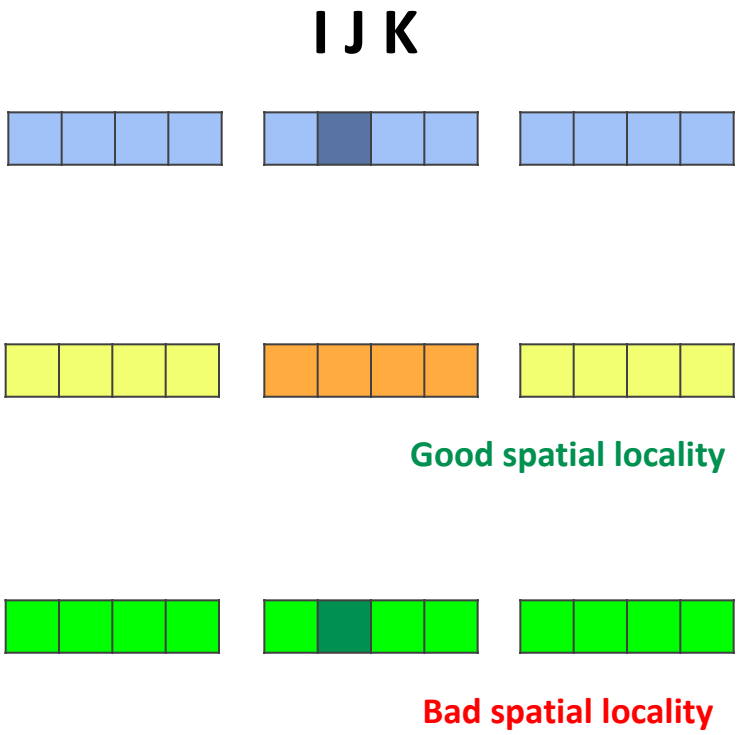
```
void matMulNaive(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto K = A.col();
    auto N = B.col();

    for(int i = 0; i < M; ++i){
        for(int j = 0; j < N; ++j){
            for(int k = 0; k < K; ++k){
                C(i, j) += A(i, k) * B(k, j);
            }
        }
    }
}
```

Changing Loop Order

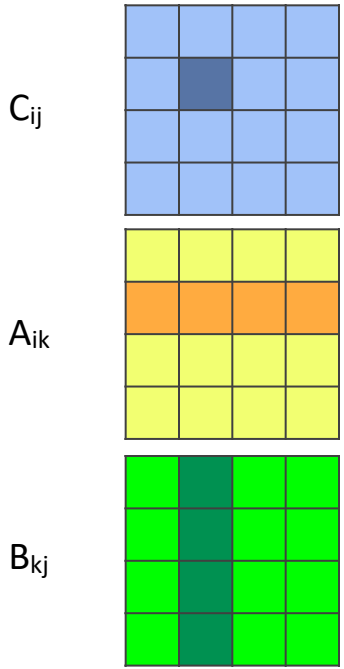
```
void matMulNaive_Order(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto K = A.col();
    auto N = B.col();

    for(int i = 0; i < M; ++i){
        for(int k = 0; k < K; ++k){
            for(int j = 0; j < N; ++j){
                C(i, j) += A(i, k) * B(k, j);
            }
        }
    }
}
```

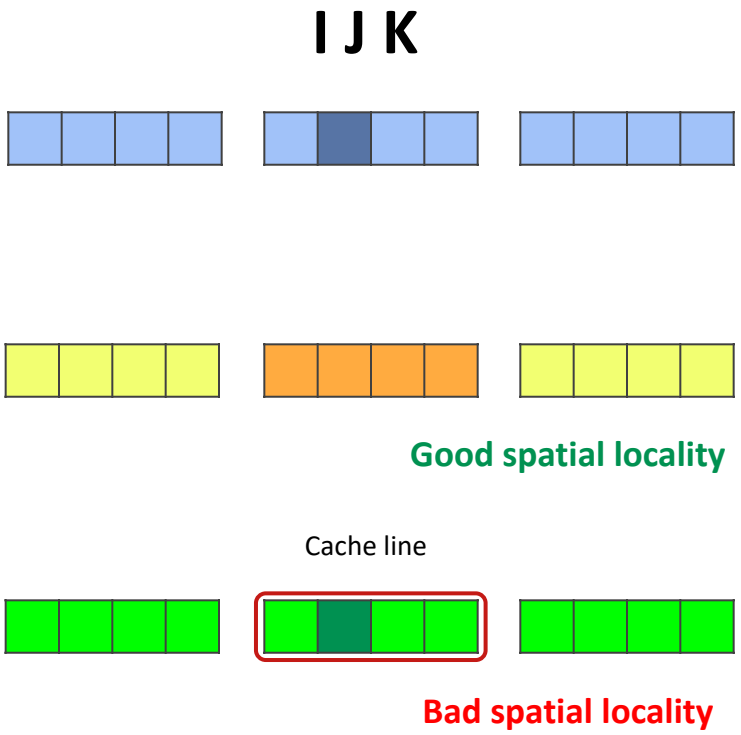
Changing Loop Order



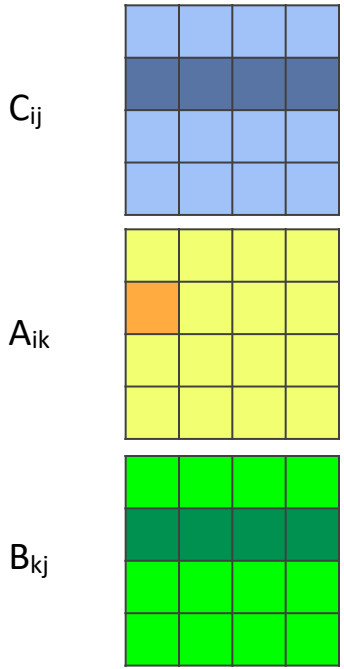


Changing Loop Order





Changing Loop Order



I K J

Good spatial locality

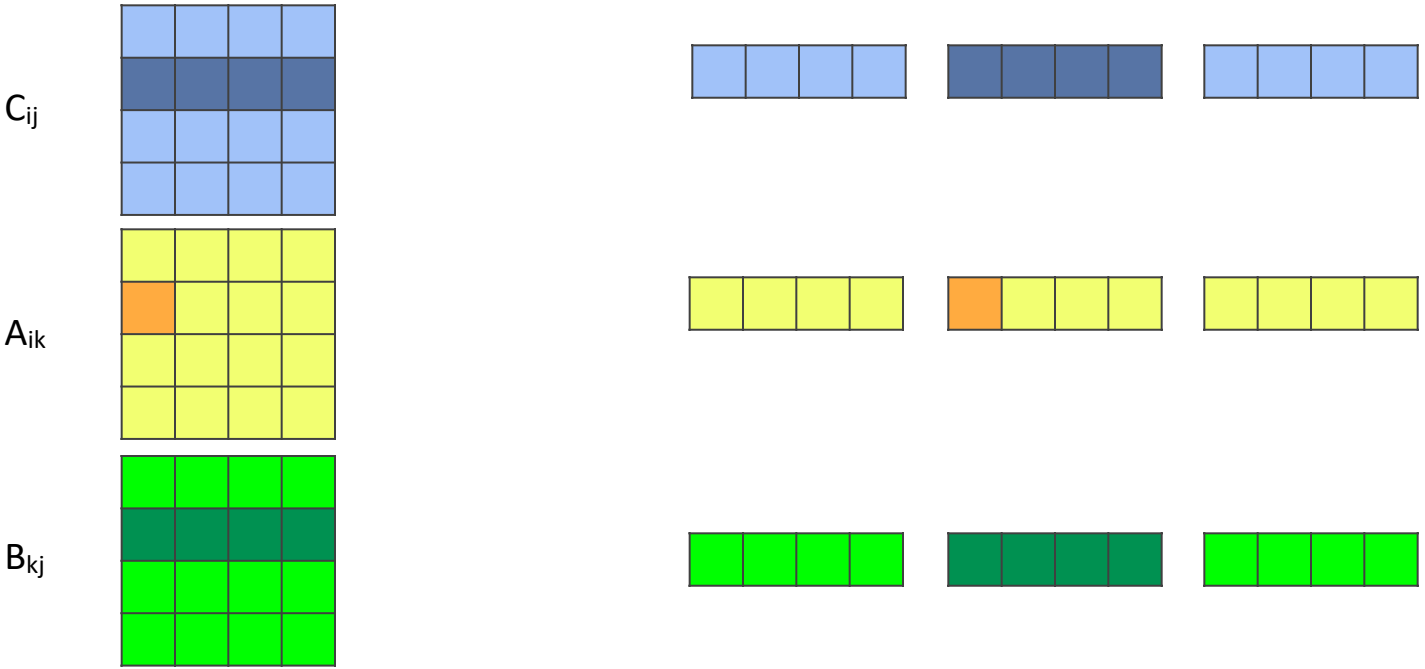
Cache line

Good spatial locality

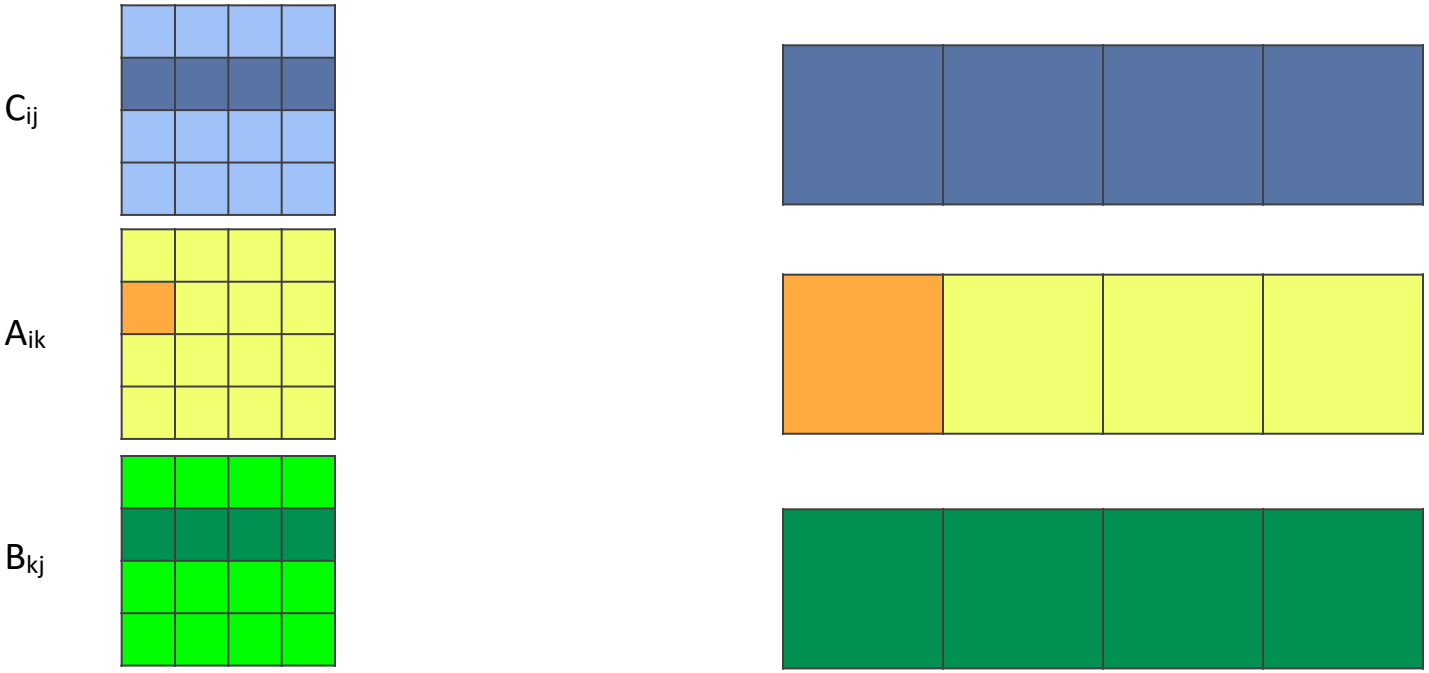
Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%

Tiling

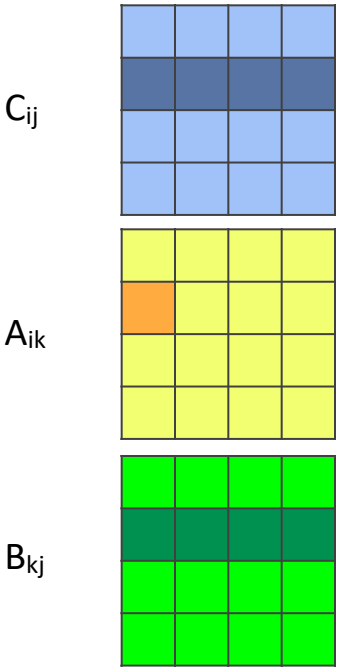
Tiling



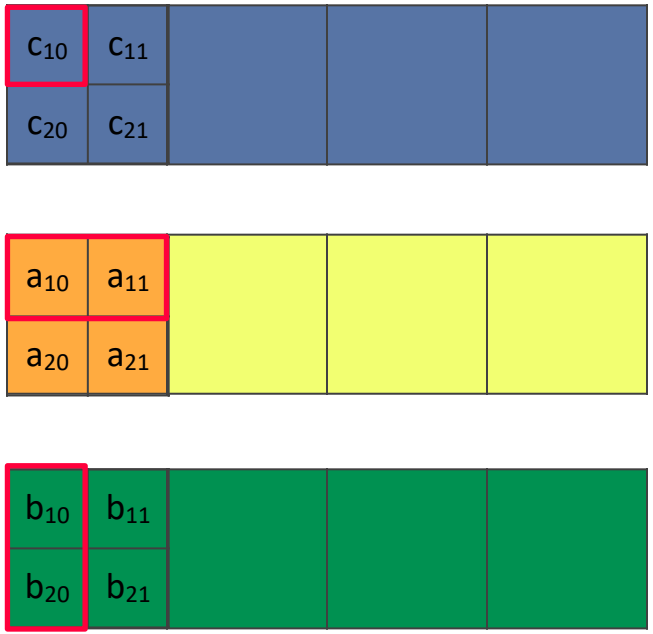
Tiling



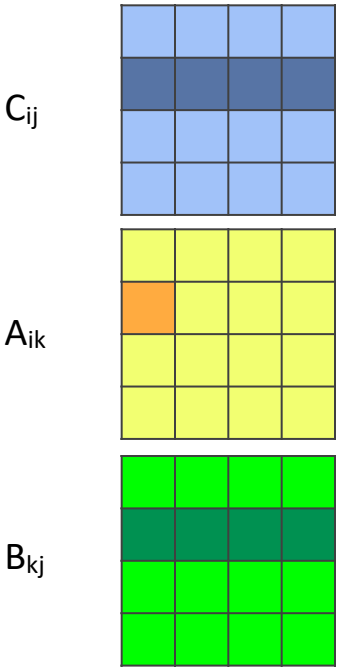
Tiling



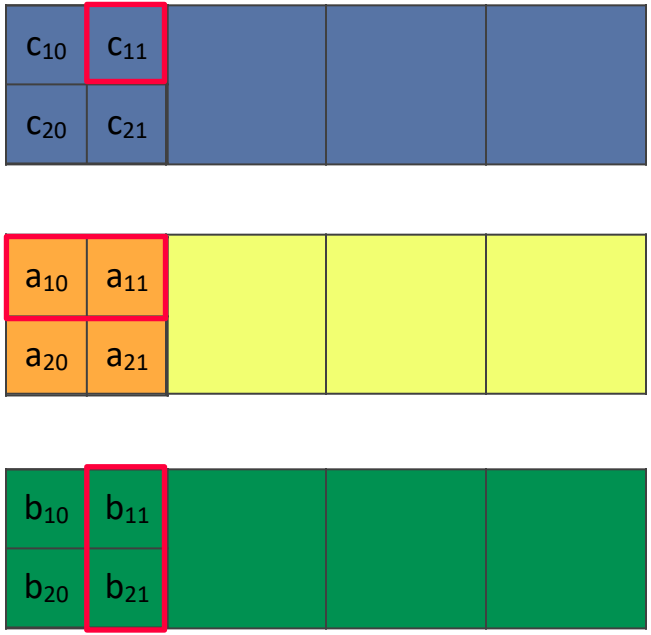
I K J



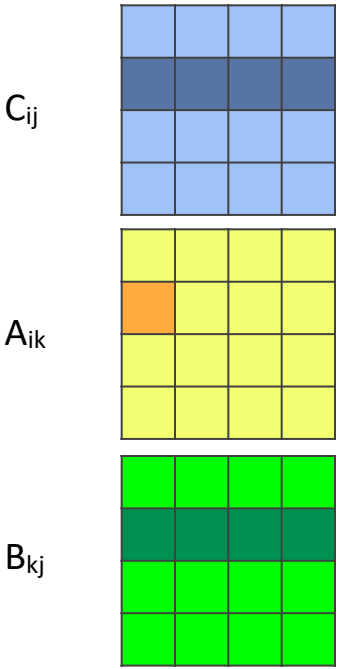
Tiling



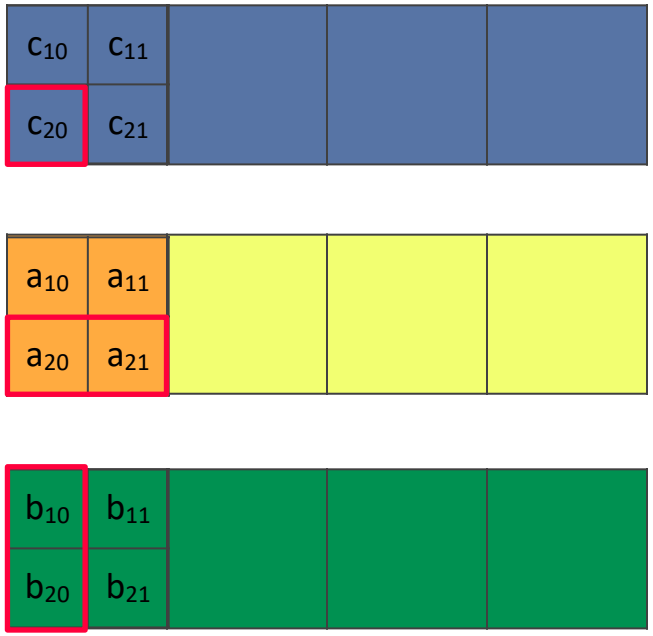
I K J



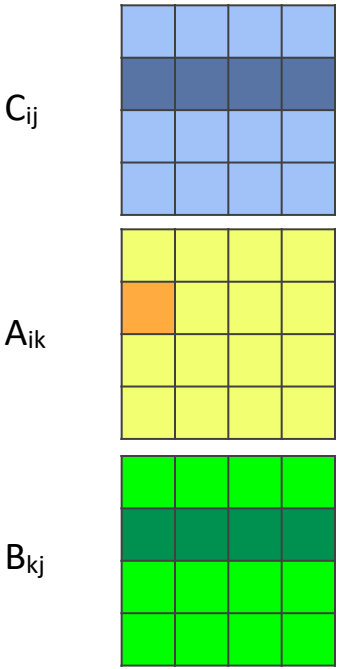
Tiling

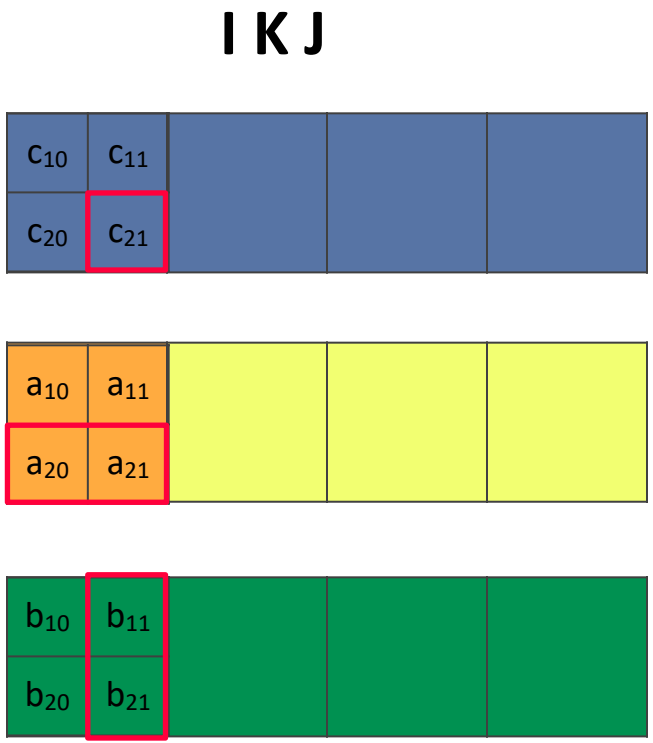


I K J



Tiling





Tiling

```
void matMulNaive_Order(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto K = A.col();
    auto N = B.col();

    for(int i = 0; i < M; ++i){
        for(int k = 0; k < K; ++k){
            for(int j = 0; j < N; ++j){
                C(i, j) += A(i, k) * B(k, j);
            }
        }
    }
}
```

Tiling

```
void matMul_Naive_Tile(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto N = B.col();
    auto K = A.col();

    constexpr auto BLOCK = 64;
    for(int ib = 0; ib < M; ib += BLOCK){
        for(int kb = 0; kb < K; kb += BLOCK){
            for(int jb = 0; jb < N; jb += BLOCK){
                const double* a  = &A(ib, kb);
                const double* mb = &B(kb, jb);
                double*       c  = &C(ib, jb);

                for(int i = 0; i < BLOCK; ++i, c += N, a += K){
                    const double* b = mb;
                    for(int k = 0; k < BLOCK; ++k, b += N){
                        for(int j = 0; j < BLOCK; ++j){
                            c[j] += a[k] * b[j];
                        }
                    }
                }
                //...
```

Tiling

```
void matMul_Naive_Tile(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto N = B.col();
    auto K = A.col();

    constexpr auto BLOCK = 64;
    for(int ib = 0; ib < M; ib += BLOCK){
        for(int kb = 0; kb < K; kb += BLOCK){
            for(int jb = 0; jb < N; jb += BLOCK){
                const double* a  = &A(ib, kb);
                const double* mb = &B(kb, jb);
                double*       c  = &C(ib, jb);

                for(int i = 0; i < BLOCK; ++i, c += N, a += K){
                    const double* b = mb;
                    for(int k = 0; k < BLOCK; ++k, b += N){
                        for(int j = 0; j < BLOCK; ++j){
                            c[j] += a[k] * b[j];
                        }
                    }
                }
                //...
```

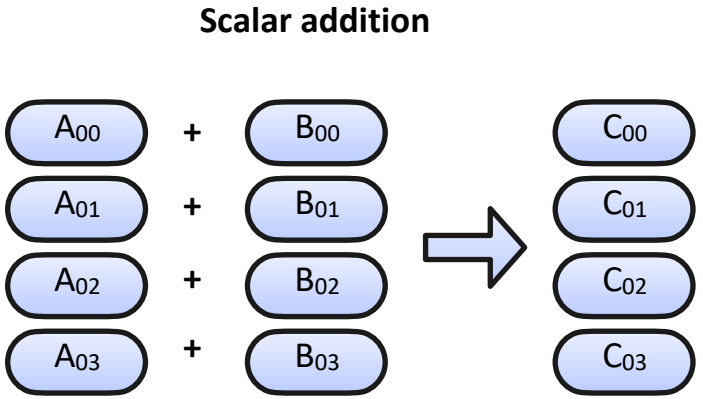
Tiling

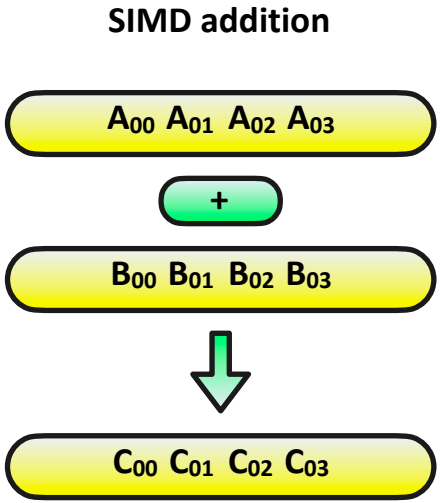
```
template<int BLOCK>
void naive_block(const double*  a,
                 const double*  mb,
                 double*  c,
                 int N,
                 int K)
{
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            for(int j = 0; j < BLOCK; ++j){
                c[j] += a[k] * b[j];
            }
        }
    }
}
```

Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%
3	+Tiling	3.9 s	3	38.9	12.2	8.3%

Vectorization

Vectorization





Vectorization

```
void matMul_Naive_Tile(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto N = B.col();
    auto K = A.col();

    constexpr auto BLOCK = 64;
    for(int ib = 0; ib < M; ib += BLOCK){
        for(int kb = 0; kb < K; kb += BLOCK){
            for(int jb = 0; jb < N; jb += BLOCK){
                const double* a  = &A(ib, kb);
                const double* mb = &B(kb, jb);
                double*       c  = &C(ib, jb);

                for(int i = 0; i < BLOCK; ++i, c += N, a += K){
                    const double* b = mb;
                    for(int k = 0; k < BLOCK; ++k, b += N){
                        for(int j = 0; j < BLOCK; ++j){
                            c[j] += a[k] * b[j];
                        }
                    }
                }
                //...
```

Vectorization

```
void matMul_Naive_Tile(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto N = B.col();
    auto K = A.col();

    constexpr auto BLOCK = 64;
    for(int ib = 0; ib < M; ib += BLOCK){
        for(int kb = 0; kb < K; kb += BLOCK){
            for(int jb = 0; jb < N; jb += BLOCK){
                const double* a  = &A(ib, kb);
                const double* mb = &B(kb, jb);
                double*       c  = &C(ib, jb);

                for(int i = 0; i < BLOCK; ++i, c += N, a += K){
                    const double* b = mb;
                    for(int k = 0; k < BLOCK; ++k, b += N){
                        for(int j = 0; j < BLOCK; ++j){
                            c[j] += a[k] * b[j];
                        }
                    }
                }
                //...
```

```
 0.67 | 310:   vbroadcastsd (%rax,%rdi,8), %ymm0
 5.35 |        vmovupd      (%r14), %ymm1
10.00 |        vfmadd213pd  (%r10), %ymm0, %ymm1
 2.92 |        vmovupd      %ymm1, (%r10)
 0.02 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.12 |        vmovupd      0x20(%r14), %ymm1
 5.55 |        vfmadd213pd  0x20(%r10), %ymm0, %ymm1
 2.96 |        vmovupd      %ymm1, 0x20(%r10)
 0.01 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.78 |        vmovupd      0x40(%r14), %ymm1
 1.95 |        vfmadd213pd  0x40(%r10), %ymm0, %ymm1
 2.90 |        vmovupd      %ymm1, 0x40(%r10)
 0.01 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.04 |        vmovupd      0x60(%r14), %ymm1
 1.82 |        vfmadd213pd  0x60(%r10), %ymm0, %ymm1
 2.86 |        vmovupd      %ymm1, 0x60(%r10)
 0.02 |        vbroadcastsd (%rax,%rdi,8), %ymm0
```

The compiler has **vectorized** the block.

Vectorization

```
void matMul_Naive_Tile(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto N = B.col();
    auto K = A.col();

    constexpr auto BLOCK = 64;
    for(int ib = 0; ib < M; ib += BLOCK){
        for(int kb = 0; kb < K; kb += BLOCK){
            for(int jb = 0; jb < N; jb += BLOCK){
                const double* a  = &A(ib, kb);
                const double* mb = &B(kb, jb);
                double*       c  = &C(ib, jb);

                for(int i = 0; i < BLOCK; ++i, c += N, a += K){
                    const double* b = mb;
                    for(int k = 0; k < BLOCK; ++k, b += N){
                        for(int j = 0; j < BLOCK; ++j){
                            c[j] += a[k] * b[j];
                        }
                    }
                }
                //...
```

```
 0.67 | 310:   vbroadcastsd (%rax,%rdi,8), %ymm0
 5.35 |        vmovupd      (%r14), %ymm1
10.00 |        vfmadd213pd  (%r10), %ymm0, %ymm1
 2.92 |        vmovupd      %ymm1, (%r10)
 0.02 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.12 |        vmovupd      0x20(%r14), %ymm1
 5.55 |        vfmadd213pd  0x20(%r10), %ymm0, %ymm1
 2.96 |        vmovupd      %ymm1, 0x20(%r10)
 0.01 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.78 |        vmovupd      0x40(%r14), %ymm1
 1.95 |        vfmadd213pd  0x40(%r10), %ymm0, %ymm1
 2.90 |        vmovupd      %ymm1, 0x40(%r10)
 0.01 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.04 |        vmovupd      0x60(%r14), %ymm1
 1.82 |        vfmadd213pd  0x60(%r10), %ymm0, %ymm1
 2.86 |        vmovupd      %ymm1, 0x60(%r10)
 0.02 |        vbroadcastsd (%rax,%rdi,8), %ymm0
```

The compiler has **vectorized** the block.

Vectorization

```
void matMul_Naive_Tile(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto N = B.col();
    auto K = A.col();

    constexpr auto BLOCK = 64;
    for(int ib = 0; ib < M; ib += BLOCK){
        for(int kb = 0; kb < K; kb += BLOCK){
            for(int jb = 0; jb < N; jb += BLOCK){
                const double* a  = &A(ib, kb);
                const double* b  = &B(kb, jb);
                double*       c  = &C(ib, jb);

                ikernels::naive_block<Nr, Mr, Kc, Nc>
                   (c, a, b, N, K);

                //...
```

```
1.53 | 180: ┌─→vmovapd      %ymm0, %ymm1
1.27 |       |  vbroadcastsd (%r12,%rbx,8), %ymm0
8.98 |       |  vfmadd231pd  -0x1e0(%r11), %ymm0, %ymm15
2.81 |       |  vmovupd      %ymm15, 0x70(%rsp)
4.38 |       |  vfmadd231pd  -0x1c0(%r11), %ymm0, %ymm14
3.02 |       |  vfmadd231pd  -0x1a0(%r11), %ymm0, %ymm13
4.55 |       |  vfmadd231pd  -0x180(%r11), %ymm0, %ymm12
3.01 |       |  vfmadd231pd  -0x160(%r11), %ymm0, %ymm11
1.91 |       |  vfmadd231pd  -0x140(%r11), %ymm0, %ymm10
1.26 |       |  vfmadd231pd  -0x120(%r11), %ymm0, %ymm9
1.58 |       |  vfmadd231pd  -0x100(%r11), %ymm0, %ymm8
2.32 |       |  vfmadd231pd  -0xe0(%r11), %ymm0, %ymm7
2.04 |       |  vfmadd231pd  -0xc0(%r11), %ymm0, %ymm6
1.07 |       |  vfmadd231pd  -0xa0(%r11), %ymm0, %ymm5
1.93 |       |  vfmadd231pd  -0x80(%r11), %ymm0, %ymm4
2.65 |       |  vfmadd231pd  -0x60(%r11), %ymm0, %ymm3
2.02 |       |  vfmadd231pd  -0x40(%r11), %ymm0, %ymm2
```

The compiler has **optimized** the block.

Vectorization

```
template<int BLOCK>
void naive_block(const double*  a,
                 const double*  mb,
                 double*  c,
                 int N,
                 int K)
{
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            for(int j = 0; j < BLOCK; ++j){
                c[j] += a[k] * b[j];
            }
        }
    }
}
```

Vectorization

```
template<int BLOCK>
void simd_block(const double*  a,
                const double*  mb,
                double*  c,
                int N,
                int K)
{
    constexpr int simd_doubles = 128 / (sizeof(double) * 8); // Equals 2
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m128d a_reg = _mm_load_sd(&a[k]);
            a_reg         = _mm_unpacklo_pd(a_reg, a_reg);
            for(int j = 0; j < BLOCK; j += simd_doubles){
                __m128d b_reg = _mm_load_pd(&b[j]);
                __m128d c_reg = _mm_load_pd(&c[j]);
                c_reg = _mm_add_pd(_mm_mul_pd(b_reg, a_reg), c_reg);
                _mm_store_pd(&c[j], c_reg);
            }
        }
    }
}
```

Vectorization

```
template<int BLOCK>
void simd_block(const double*  a,
                const double*  mb,
                double*  c,
                int N,
                int K)
{
    constexpr int simd_doubles = 128 / (sizeof(double) * 8); // Equals 2
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m128d a_reg = _mm_load_sd(&a[k]);
            a_reg         = _mm_unpacklo_pd(a_reg, a_reg);
            for(int j = 0; j < BLOCK; j += simd_doubles){
                __m128d b_reg = _mm_load_pd(&b[j]);
                __m128d c_reg = _mm_load_pd(&c[j]);
                c_reg = _mm_add_pd(_mm_mul_pd(b_reg, a_reg), c_reg);
                _mm_store_pd(&c[j], c_reg);
            }
        }
    }
}
```

What Every Programmer Should Know About Memory

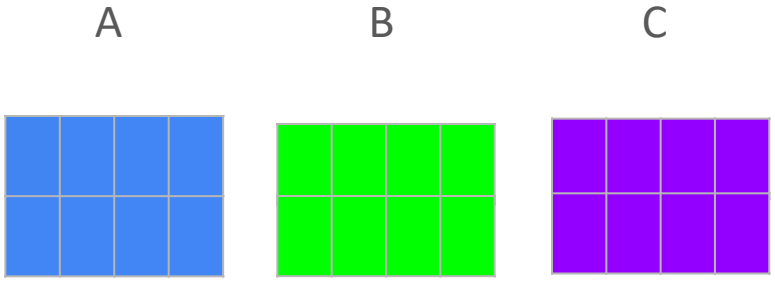
Ulrich Drepper
Red Hat, Inc.

November 21, 2007

Vectorization

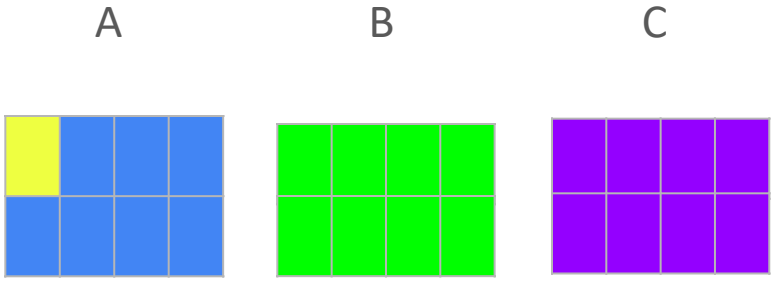
```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8); // Equals 4
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```

Vectorization




```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8); // Equals 4
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```

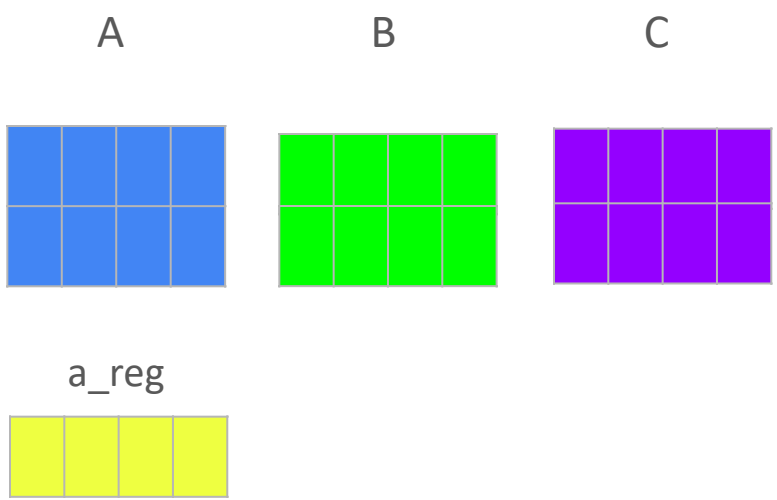
Vectorization




```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8);
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```

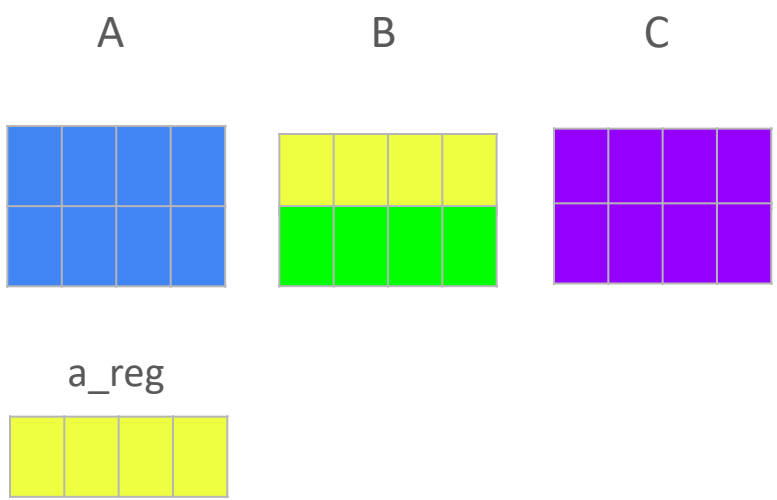
Vectorization

```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8);
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg =  _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```



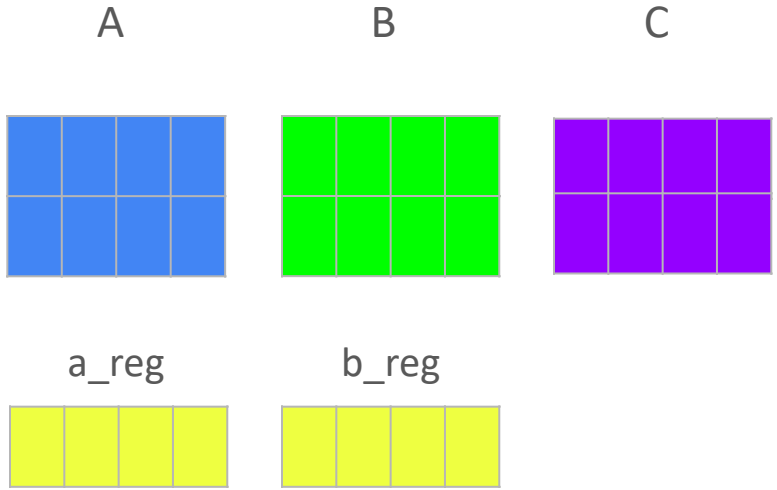
Vectorization

```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8);
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```



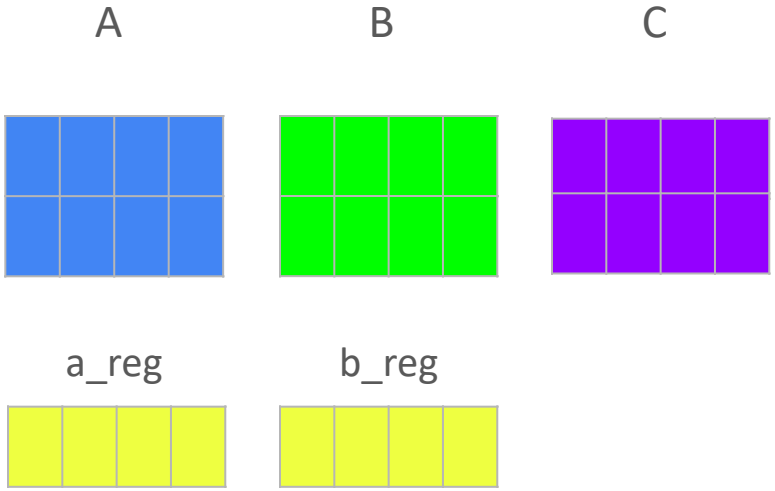
Vectorization

```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8);
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```



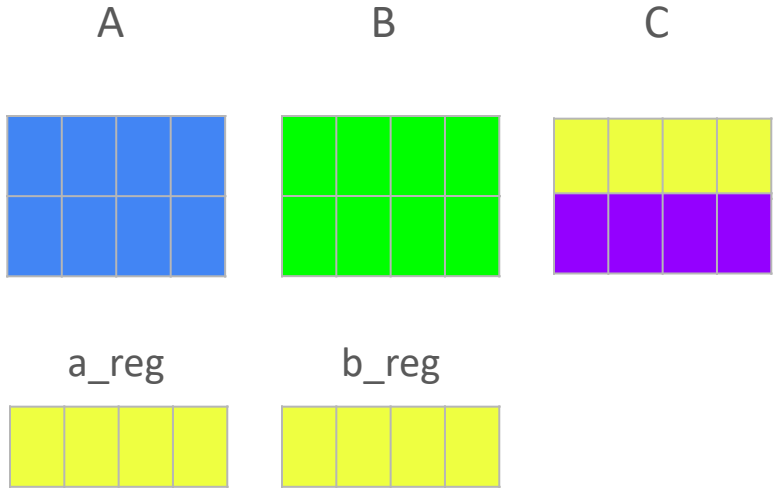
Vectorization

```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8);
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg =  _mm256_loadu_pd(&b[j]);
                __m256d c_reg =  _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```



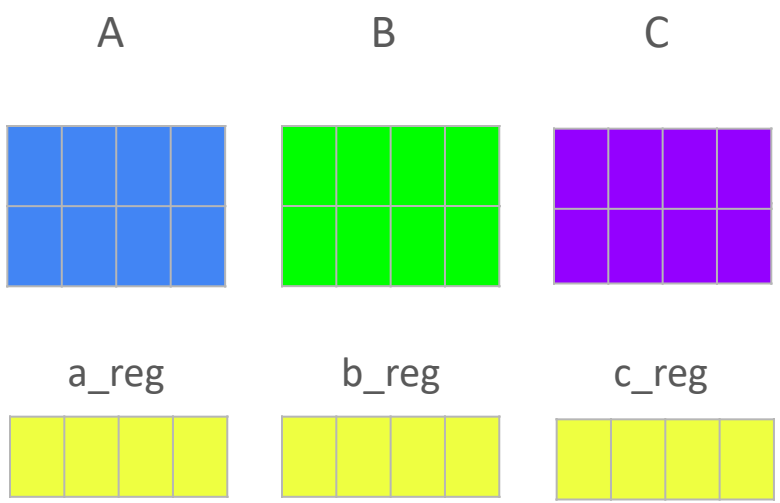
Vectorization

```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8);
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg =  _mm256_loadu_pd(&b[j]);
                __m256d c_reg =  _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```



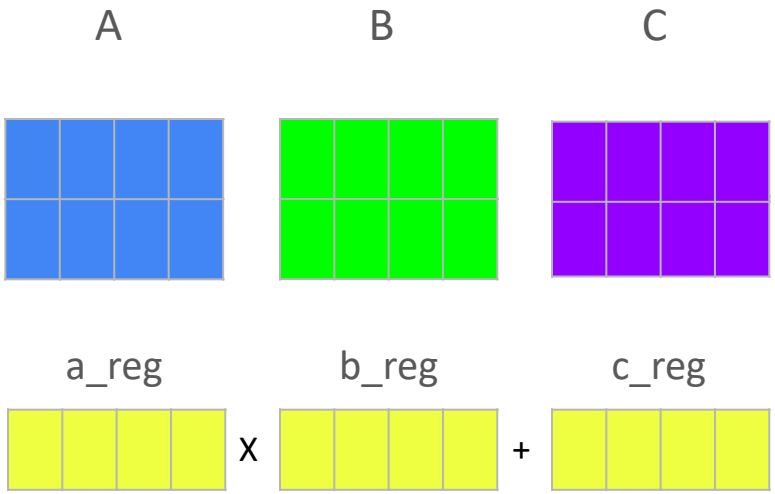
Vectorization

```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8);
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg =  _mm256_loadu_pd(&b[j]);
                __m256d c_reg =  _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```



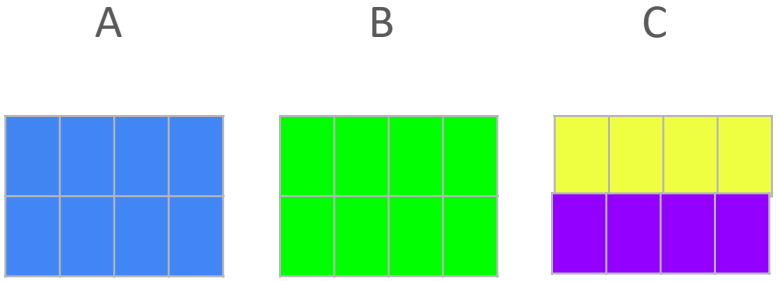
Vectorization

```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8);
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg =  mm256 loadu pd(&c[j]);
                c_reg =  mm256 fmadd pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```



Fused Multiply Add (FMA)

Vectorization




```
template<int BLOCK>
void avx_block(const double*  a,
               const double*  mb,
               double*  c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8);
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```

Vectorization

```
 0.67 | 310:   vbroadcastsd (%rax,%rdi,8), %ymm0
 5.35 |        vmovupd      (%r14), %ymm1
10.00 |        vfmadd213pd  (%r10), %ymm0, %ymm1
 2.92 |        vmovupd      %ymm1, (%r10)
 0.02 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.12 |        vmovupd      0x20(%r14), %ymm1
 5.55 |        vfmadd213pd  0x20(%r10), %ymm0, %ymm1
 2.96 |        vmovupd      %ymm1, 0x20(%r10)
 0.01 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.78 |        vmovupd      0x40(%r14), %ymm1
 1.95 |        vfmadd213pd  0x40(%r10), %ymm0, %ymm1
 2.90 |        vmovupd      %ymm1, 0x40(%r10)
 0.01 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.04 |        vmovupd      0x60(%r14), %ymm1
 1.82 |        vfmadd213pd  0x60(%r10), %ymm0, %ymm1
 2.86 |        vmovupd      %ymm1, 0x60(%r10)
 0.02 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.32 |        vmovupd      0x80(%r14), %ymm1
 1.53 |        vfmadd213pd  0x80(%r10), %ymm0, %ymm1
 3.21 |        vmovupd      %ymm1, 0x80(%r10)
 0.03 |        vbroadcastsd (%rax,%rdi,8), %ymm0
```

Auto

```
 3.06 | 240: ┌─>vbroadcastsd (%rdx,%rcx), %ymm12
 4.17 |      |  vmovupd      (%r13), %ymm13
 6.02 |      |  vmovupd      0x20(%r13), %ymm14
 4.72 |      |  vmovupd      0x40(%r13), %ymm15
 3.15 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm11
 3.84 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm10
 4.98 |      |  vfmadd231pd  %ymm12, %ymm15, %ymm9
 1.09 |      |  vbroadcastsd (%r12,%rcx), %ymm12
 2.40 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm8
 1.37 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm7
 2.40 |      |  vfmadd231pd  %ymm12, %ymm15, %ymm6
 1.03 |      |  vbroadcastsd (%rax,%rcx), %ymm12
 2.08 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm5
 1.09 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm4
 2.26 |      |  vfmadd231pd  %ymm12, %ymm15, %ymm3
 0.85 |      |  vbroadcastsd (%rbx,%rcx), %ymm12
 2.13 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm2
 1.10 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm1
 2.05 |      |  vfmadd231pd  %ymm15, %ymm12, %ymm0
 0.43 |      |  vbroadcastsd 0x8(%rdx,%rcx), %ymm12
 2.50 |      |  vmovupd      (%r13,%r15), %ymm13
 2.42 |      |  vmovupd      0x20(%r13,%r15), %ymm14
 3.79 |      |  vmovupd      0x40(%r13,%r15), %ymm15
 0.35 |      |  leaq         (%r13,%r15),%r13
```

Manual

Vectorization

```
 0.67 | 310:   vbroadcastsd (%rax,%rdi,8), %ymm0
 5.35 |        vmovupd      (%r14), %ymm1
10.00 |        vfmadd213pd  (%r10), %ymm0, %ymm1
 2.92 |        vmovupd      %ymm1, (%r10)
 0.02 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.12 |        vmovupd      0x20(%r14), %ymm1
 5.55 |        vfmadd213pd  0x20(%r10), %ymm0, %ymm1
 2.96 |        vmovupd      %ymm1, 0x20(%r10)
 0.01 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.78 |        vmovupd      0x40(%r14), %ymm1
 1.95 |        vfmadd213pd  0x40(%r10), %ymm0, %ymm1
 2.90 |        vmovupd      %ymm1, 0x40(%r10)
 0.01 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.04 |        vmovupd      0x60(%r14), %ymm1
 1.82 |        vfmadd213pd  0x60(%r10), %ymm0, %ymm1
 2.86 |        vmovupd      %ymm1, 0x60(%r10)
 0.02 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.32 |        vmovupd      0x80(%r14), %ymm1
 1.53 |        vfmadd213pd  0x80(%r10), %ymm0, %ymm1
 3.21 |        vmovupd      %ymm1, 0x80(%r10)
 0.03 |        vbroadcastsd (%rax,%rdi,8), %ymm0
```

Auto

```
 3.06 | 240: ┌─>vbroadcastsd (%rdx,%rcx), %ymm12
 4.17 |      |  vmovupd      (%r13), %ymm13
 6.02 |      |  vmovupd      0x20(%r13), %ymm14
 4.72 |      |  vmovupd      0x40(%r13), %ymm15
 3.15 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm11
 3.84 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm10
 4.98 |      |  vfmadd231pd  %ymm12, %ymm15, %ymm9
 1.09 |      |  vbroadcastsd (%r12,%rcx), %ymm12
 2.40 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm8
 1.37 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm7
 2.40 |      |  vfmadd231pd  %ymm12, %ymm15, %ymm6
 1.03 |      |  vbroadcastsd (%rax,%rcx), %ymm12
 2.08 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm5
 1.09 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm4
 2.26 |      |  vfmadd231pd  %ymm12, %ymm15, %ymm3
 0.85 |      |  vbroadcastsd (%rbx,%rcx), %ymm12
 2.13 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm2
 1.10 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm1
 2.05 |      |  vfmadd231pd  %ymm15, %ymm12, %ymm0
 0.43 |      |  vbroadcastsd 0x8(%rdx,%rcx), %ymm12
 2.50 |      |  vmovupd      (%r13,%r15), %ymm13
 2.42 |      |  vmovupd      0x20(%r13,%r15), %ymm14
 3.79 |      |  vmovupd      0x40(%r13,%r15), %ymm15
 0.35 |      |  leaq         (%r13,%r15),%r13
```

Manual

Vectorization

```
 0.67 | 310:   vbroadcastsd (%rax,%rdi,8), %ymm0
 5.35 |        vmovupd      (%r14), %ymm1
10.00 |        vfmadd213pd  (%r10), %ymm0, %ymm1
 2.92 |        vmovupd      %ymm1, (%r10)
 0.02 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.12 |        vmovupd      0x20(%r14), %ymm1
 5.55 |        vfmadd213pd  0x20(%r10), %ymm0, %ymm1
 2.96 |        vmovupd      %ymm1, 0x20(%r10)
 0.01 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.78 |        vmovupd      0x40(%r14), %ymm1
 1.95 |        vfmadd213pd  0x40(%r10), %ymm0, %ymm1
 2.90 |        vmovupd      %ymm1, 0x40(%r10)
 0.01 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.04 |        vmovupd      0x60(%r14), %ymm1
 1.82 |        vfmadd213pd  0x60(%r10), %ymm0, %ymm1
 2.86 |        vmovupd      %ymm1, 0x60(%r10)
 0.02 |        vbroadcastsd (%rax,%rdi,8), %ymm0
 0.32 |        vmovupd      0x80(%r14), %ymm1
 1.53 |        vfmadd213pd  0x80(%r10), %ymm0, %ymm1
 3.21 |        vmovupd      %ymm1, 0x80(%r10)
 0.03 |        vbroadcastsd (%rax,%rdi,8), %ymm0
```

Auto

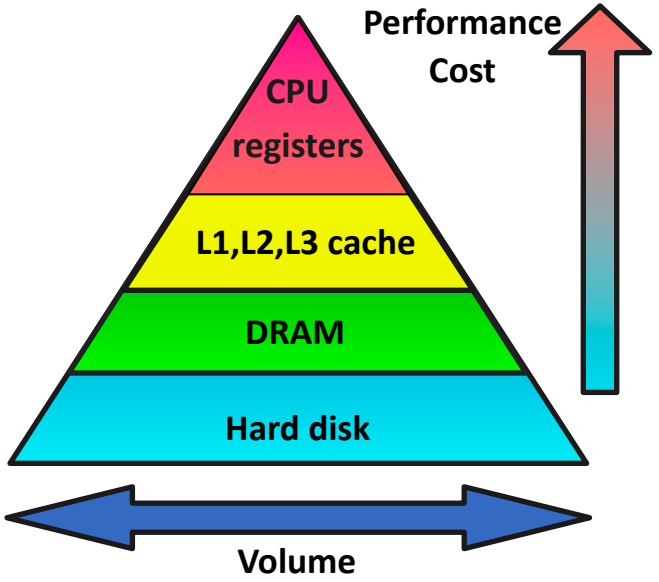
```
 3.06 | 240: ┌─>vbroadcastsd (%rdx,%rcx), %ymm12
 4.17 |      |  vmovupd      (%r13), %ymm13
 6.02 |      |  vmovupd      0x20(%r13), %ymm14
 4.72 |      |  vmovupd      0x40(%r13), %ymm15
 3.15 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm11
 3.84 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm10
 4.98 |      |  vfmadd231pd  %ymm12, %ymm15, %ymm9
 1.09 |      |  vbroadcastsd (%r12,%rcx), %ymm12
 2.40 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm8
 1.37 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm7
 2.40 |      |  vfmadd231pd  %ymm12, %ymm15, %ymm6
 1.03 |      |  vbroadcastsd (%rax,%rcx), %ymm12
 2.08 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm5
 1.09 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm4
 2.26 |      |  vfmadd231pd  %ymm12, %ymm15, %ymm3
 0.85 |      |  vbroadcastsd (%rbx,%rcx), %ymm12
 2.13 |      |  vfmadd231pd  %ymm13, %ymm12, %ymm2
 1.10 |      |  vfmadd231pd  %ymm14, %ymm12, %ymm1
 2.05 |      |  vfmadd231pd  %ymm15, %ymm12, %ymm0
 0.43 |      |  vbroadcastsd 0x8(%rdx,%rcx), %ymm12
 2.50 |      |  vmovupd      (%r13,%r15), %ymm13
 2.42 |      |  vmovupd      0x20(%r13,%r15), %ymm14
 3.79 |      |  vmovupd      0x40(%r13,%r15), %ymm15
 0.35 |      |  leaq         (%r13,%r15),%r13
```

Manual

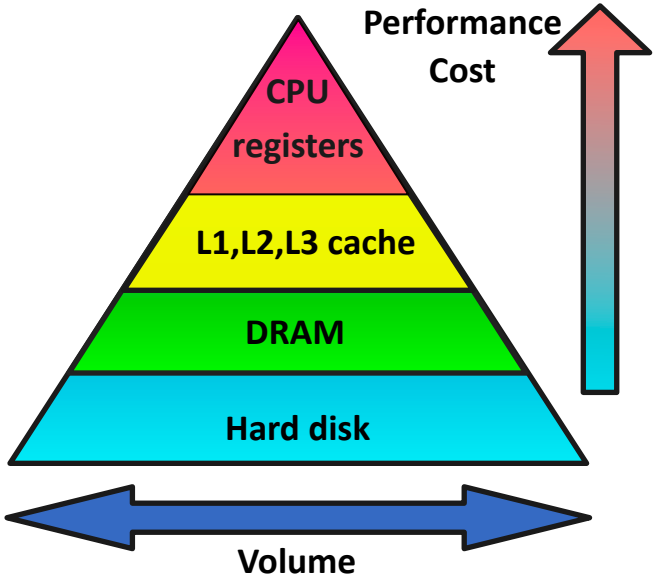
Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%
3	+Tiling	3.9 s	3	38.9	12.2	8.3%
4	+Avx	3.158 s	1.23	48.13	15.1	10.2%

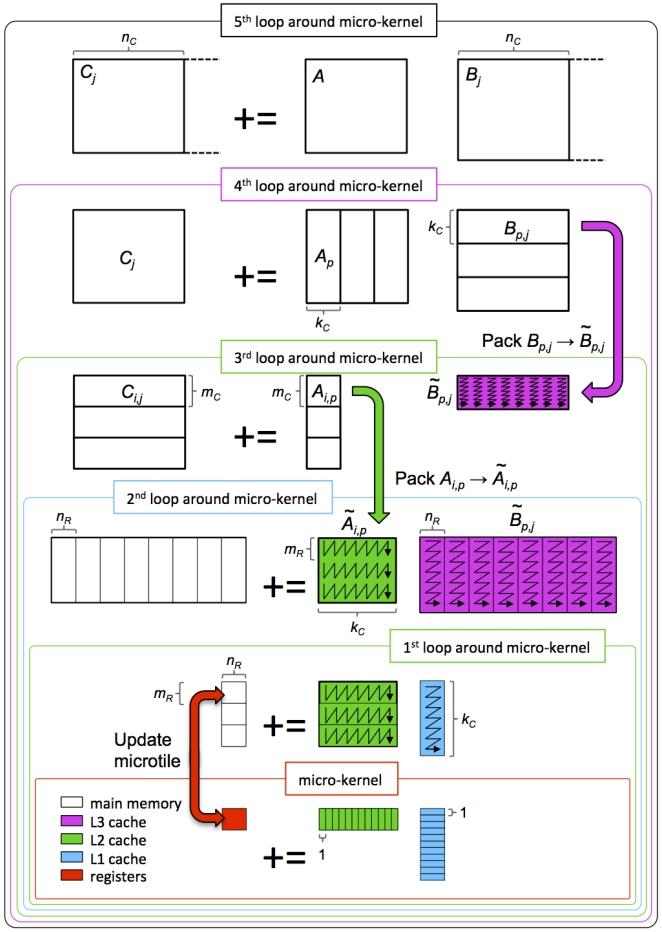
Cache Aware Implementation

Cache aware implementation



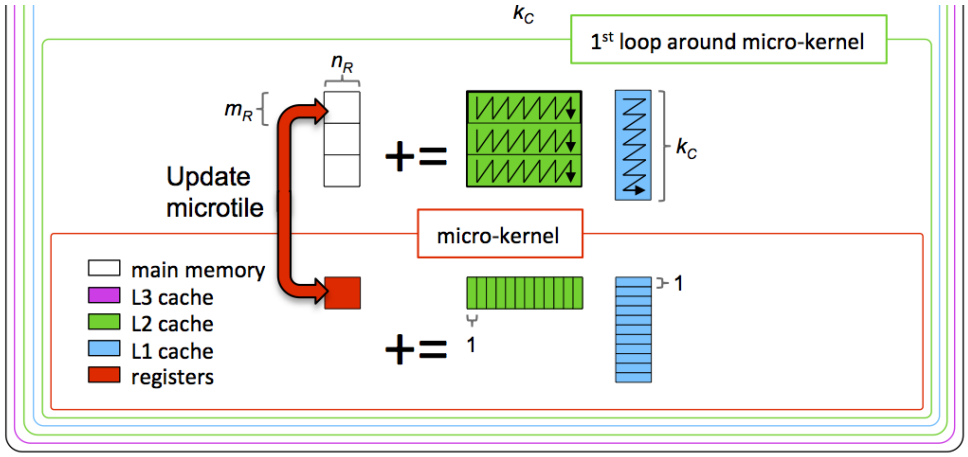
Cache aware implementation



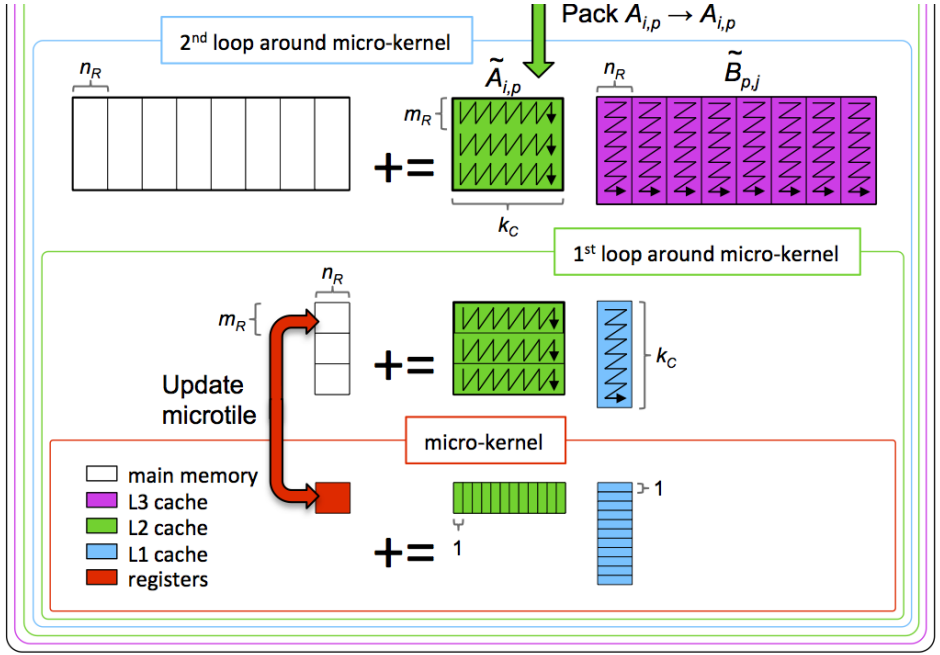


Robert A. van de Geijn is a professor of Computer Sciences at the University of Texas at Austin.

Cache aware implementation



Cache aware implementation



Cache aware implementation

```
void matMul_Naive_Tile(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto N = B.col();
    auto K = A.col();

    constexpr auto BLOCK = 64;
    for(int ib = 0; ib < M; ib += BLOCK){
        for(int kb = 0; kb < K; kb += BLOCK){
            for(int jb = 0; jb < N; jb += BLOCK){
                const double* a  = &A(ib, kb);
                const double* b  = &B(kb, jb);
                double*       c  = &C(ib, jb);

                ikernels::naive_block<Nr, Mr, Kc, Nc>
                   (c, a, b, N, K);

                //...
```

Cache aware implementation

```
void matMul_Avx_Cache(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto K = A.col();
    auto N = B.col();

    constexpr auto Mc = 180, Nc = 96, Kc = 240, Nr = 12, Mr = 4;

    for(int ib = 0; ib < M; ib += Mc){
        for(int kb = 0; kb < K; kb += Kc){
            for(int jb = 0; jb < N; jb += Nc){
                const double* ma = &A(ib, kb);
                const double* mb = &B(kb, jb);
                double*       mc = &C(ib, jb);

                for(int i2 = 0; i2 < Mc; i2 += Mr){
                    for(int j2 = 0; j2 < Nc; j2 += Nr){
                        const double* a = &ma[i2 * K];
                        const double* b = &mb[j2];
                        double*       c = &mc[i2 * N + j2];
                        ikernels::avx_block<Nr, Mr, Kc, Nc>(c, a, b, N, K);
                    }
                }
                // ...
```

Cache aware implementation

```
void matMul_Avx_Cache(const Matrix<double>& A, const Matrix<double>& B, Matrix<double>& C)
{
    auto M = A.row();
    auto K = A.col();
    auto N = B.col();

    constexpr auto Mc = 180, Nc = 96, Kc = 240, Nr = 12, Mr = 4;

    for(int ib = 0; ib < M; ib += Mc){
        for(int kb = 0; kb < K; kb += Kc){
            for(int jb = 0; jb < N; jb += Nc){
                const double* ma = &A(ib, kb);
                const double* mb = &B(kb, jb);
                double*       mc = &C(ib, jb);

                for(int i2 = 0; i2 < Mc; i2 += Mr){
                    for(int j2 = 0; j2 < Nc; j2 += Nr){
                        const double* a = &ma[i2 * K];
                        const double* b = &mb[j2];
                        double*       c = &mc[i2 * N + j2];
                        ikernels::avx_block<Nr, Mr, Kc, Nc>(c, a, b, N, K);
                    }
                }
                // ...
```

Nc,Mc,Kc - size of cache blocks

Nr,Mr - size of register blocks

Cache aware implementation

```
template<int BLOCK>
void avx_block(const double* a,
               const double* mb,
               double* c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8);
    for(int i = 0; i < BLOCK; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < BLOCK; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < BLOCK; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```

Cache aware implementation

```
template<int Nr, int Mr, int Kc, int Nc>
void avx_cache(const double* a,
               const double* mb,
               double* c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8); // Equals 4
    for(int i = 0; i < Mr; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < Kc; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```

Cache aware implementation

```
3.06  240: ┌─→vbroadcastsd (%rdx,%rcx), %ymm12
4.17       │  vmovupd     (%r13), %ymm13
6.02       │  vmovupd     0x20(%r13), %ymm14
4.72       │  vmovupd     0x40(%r13), %ymm15
3.15       │  vfmadd231pd %ymm13, %ymm12, %ymm11
3.84       │  vfmadd231pd %ymm14, %ymm12, %ymm10
4.98       │  vfmadd231pd %ymm12, %ymm15, %ymm9
1.09       │  vbroadcastsd (%r12,%rcx), %ymm12
2.40       │  vfmadd231pd %ymm13, %ymm12, %ymm8
1.37       │  vfmadd231pd %ymm14, %ymm12, %ymm7
2.40       │  vfmadd231pd %ymm12, %ymm15, %ymm6
1.03       │  vbroadcastsd (%rax,%rcx), %ymm12
2.08       │  vfmadd231pd %ymm13, %ymm12, %ymm5
1.09       │  vfmadd231pd %ymm14, %ymm12, %ymm4
2.26       │  vfmadd231pd %ymm12, %ymm15, %ymm3
0.85       │  vbroadcastsd (%rbx,%rcx), %ymm12
2.13       │  vfmadd231pd %ymm13, %ymm12, %ymm2
1.10       │  vfmadd231pd %ymm14, %ymm12, %ymm1
2.05       │  vfmadd231pd %ymm15, %ymm12, %ymm0
0.43       │  vbroadcastsd 0x8(%rdx,%rcx), %ymm12
2.50       │  vmovupd     (%r13,%r15), %ymm13
2.42       │  vmovupd     0x20(%r13,%r15), %ymm14
3.79       │  vmovupd     0x40(%r13,%r15), %ymm15
0.35       │  leaq        (%r13,%r15),%r13
```

AVX

```
0.63  230: ┌─→vbroadcastsd -0x18(%r10,%rbp,8), %ymm3
0.58       │  vfmadd231pd  (%r13), %ymm3, %ymm2
2.10       │  vfmadd231pd  0x20(%r13), %ymm3, %ymm1
1.83       │  vfmadd231pd  0x40(%r13), %ymm3, %ymm0
0.09       │  vbroadcastsd -0x10(%r10,%rbp,8), %ymm3
1.45       │  vfmadd231pd  (%r13,%rbx), %ymm3, %ymm2
2.45       │  vfmadd231pd  0x20(%r13,%rbx), %ymm3, %ymm1
2.16       │  vfmadd231pd  0x40(%r13,%rbx), %ymm3, %ymm0
0.34       │  leaq         (%r13,%rbx),%rcx
0.05       │  vbroadcastsd -0x8(%r10,%rbp,8), %ymm3
1.53       │  vfmadd231pd  (%rbx,%rcx), %ymm3, %ymm2
1.95       │  vfmadd231pd  0x20(%rbx,%rcx), %ymm3, %ymm1
2.40       │  vfmadd231pd  0x40(%rbx,%rcx), %ymm3, %ymm0
0.55       │  leaq         (%rcx,%rbx),%r13
0.05       │  vbroadcastsd (%r10,%rbp,8), %ymm3
1.62       │  vfmadd231pd  (%rbx,%r13), %ymm3, %ymm2
1.74       │  vfmadd231pd  0x20(%rbx,%r13), %ymm3, %ymm1
2.37       │  vfmadd231pd  0x40(%rbx,%r13), %ymm3, %ymm0
0.50       │  addq         $0x4,%rbp
0.04       │  leaq         (%rbx,%rbx),%rcx
1.07       │  addq         %rcx,%r13
0.01       │  cmpq         $0x33,%rbp
0.10       └─ jne          230
```

AVX + cache awareness

Cache aware implementation

```
3.06  240: ┌→vbroadcastsd (%rdx,%rcx), %ymm12
4.17       |  vmovupd     (%r13), %ymm13
6.02       |  vmovupd     0x20(%r13), %ymm14
4.72       |  vmovupd     0x40(%r13), %ymm15
3.15       |  vfmadd231pd %ymm13, %ymm12, %ymm11
3.84       |  vfmadd231pd %ymm14, %ymm12, %ymm10
4.98       |  vfmadd231pd %ymm12, %ymm15, %ymm9
1.09       |  vbroadcastsd (%r12,%rcx), %ymm12
2.40       |  vfmadd231pd %ymm13, %ymm12, %ymm8
1.37       |  vfmadd231pd %ymm14, %ymm12, %ymm7
2.40       |  vfmadd231pd %ymm12, %ymm15, %ymm6
1.03       |  vbroadcastsd (%rax,%rcx), %ymm12
2.08       |  vfmadd231pd %ymm13, %ymm12, %ymm5
1.09       |  vfmadd231pd %ymm14, %ymm12, %ymm4
2.26       |  vfmadd231pd %ymm12, %ymm15, %ymm3
0.85       |  vbroadcastsd (%rbx,%rcx), %ymm12
2.13       |  vfmadd231pd %ymm13, %ymm12, %ymm2
1.10       |  vfmadd231pd %ymm14, %ymm12, %ymm1
2.05       |  vfmadd231pd %ymm15, %ymm12, %ymm0
0.43       |  vbroadcastsd 0x8(%rdx,%rcx), %ymm12
2.50       |  vmovupd     (%r13,%r15), %ymm13
2.42       |  vmovupd     0x20(%r13,%r15), %ymm14
3.79       |  vmovupd     0x40(%r13,%r15), %ymm15
0.35       |  leaq        (%r13,%r15),%r13
```

AVX

```
0.63  230: ┌→vbroadcastsd -0x18(%r10,%rbp,8), %ymm3
0.58       |  vfmadd231pd  (%r13), %ymm3, %ymm2
2.10       |  vfmadd231pd  0x20(%r13), %ymm3, %ymm1
1.83       |  vfmadd231pd  0x40(%r13), %ymm3, %ymm0
0.09       |  vbroadcastsd -0x10(%r10,%rbp,8), %ymm3
1.45       |  vfmadd231pd  (%r13,%rbx), %ymm3, %ymm2
2.45       |  vfmadd231pd  0x20(%r13,%rbx), %ymm3, %ymm1
2.16       |  vfmadd231pd  0x40(%r13,%rbx), %ymm3, %ymm0
0.34       |  leaq         (%r13,%rbx),%rcx
0.05       |  vbroadcastsd -0x8(%r10,%rbp,8), %ymm3
1.53       |  vfmadd231pd  (%rbx,%rcx), %ymm3, %ymm2
1.95       |  vfmadd231pd  0x20(%rbx,%rcx), %ymm3, %ymm1
2.40       |  vfmadd231pd  0x40(%rbx,%rcx), %ymm3, %ymm0
0.55       |  leaq         (%rcx,%rbx),%r13
0.05       |  vbroadcastsd (%r10,%rbp,8), %ymm3
1.62       |  vfmadd231pd  (%rbx,%r13), %ymm3, %ymm2
1.74       |  vfmadd231pd  0x20(%rbx,%r13), %ymm3, %ymm1
2.37       |  vfmadd231pd  0x40(%rbx,%r13), %ymm3, %ymm0
0.50       |  addq         $0x4,%rbp
0.04       |  leaq         (%rbx,%rbx),%rcx
1.07       |  addq         %rcx,%r13
0.01       |  cmpq         $0x33,%rbp
0.10       └─ jne          230
```

AVX + cache awareness

Cache aware implementation

```
3.06  240: ┌─→vbroadcastsd (%rdx,%rcx), %ymm12
4.17       │  vmovupd     (%r13), %ymm13
6.02       │  vmovupd     0x20(%r13), %ymm14
4.72       │  vmovupd     0x40(%r13), %ymm15
3.15       │  vfmadd231pd %ymm13, %ymm12, %ymm11
3.84       │  vfmadd231pd %ymm14, %ymm12, %ymm10
4.98       │  vfmadd231pd %ymm12, %ymm15, %ymm9
1.09       │  vbroadcastsd (%r12,%rcx), %ymm12
2.40       │  vfmadd231pd %ymm13, %ymm12, %ymm8
1.37       │  vfmadd231pd %ymm14, %ymm12, %ymm7
2.40       │  vfmadd231pd %ymm12, %ymm15, %ymm6
1.03       │  vbroadcastsd (%rax,%rcx), %ymm12
2.08       │  vfmadd231pd %ymm13, %ymm12, %ymm5
1.09       │  vfmadd231pd %ymm14, %ymm12, %ymm4
2.26       │  vfmadd231pd %ymm12, %ymm15, %ymm3
0.85       │  vbroadcastsd (%rbx,%rcx), %ymm12
2.13       │  vfmadd231pd %ymm13, %ymm12, %ymm2
1.10       │  vfmadd231pd %ymm14, %ymm12, %ymm1
2.05       │  vfmadd231pd %ymm15, %ymm12, %ymm0
0.43       │  vbroadcastsd 0x8(%rdx,%rcx), %ymm12
2.50       │  vmovupd     (%r13,%r15), %ymm13
2.42       │  vmovupd     0x20(%r13,%r15), %ymm14
3.79       │  vmovupd     0x40(%r13,%r15), %ymm15
0.35       │  leaq        (%r13,%r15),%r13
```

AVX

```
0.63  230: ┌─→vbroadcastsd -0x18(%r10,%rbp,8), %ymm3
0.58       │  vfmadd231pd  (%r13), %ymm3, %ymm2
2.10       │  vfmadd231pd  0x20(%r13), %ymm3, %ymm1
1.83       │  vfmadd231pd  0x40(%r13), %ymm3, %ymm0
0.09       │  vbroadcastsd -0x10(%r10,%rbp,8), %ymm3
1.45       │  vfmadd231pd  (%r13,%rbx), %ymm3, %ymm2
2.45       │  vfmadd231pd  0x20(%r13,%rbx), %ymm3, %ymm1
2.16       │  vfmadd231pd  0x40(%r13,%rbx), %ymm3, %ymm0
0.34       │  leaq         (%r13,%rbx),%rcx
0.05       │  vbroadcastsd -0x8(%r10,%rbp,8), %ymm3
1.53       │  vfmadd231pd  (%rbx,%rcx), %ymm3, %ymm2
1.95       │  vfmadd231pd  0x20(%rbx,%rcx), %ymm3, %ymm1
2.40       │  vfmadd231pd  0x40(%rbx,%rcx), %ymm3, %ymm0
0.55       │  leaq         (%rcx,%rbx),%r13
0.05       │  vbroadcastsd (%r10,%rbp,8), %ymm3
1.62       │  vfmadd231pd  (%rbx,%r13), %ymm3, %ymm2
1.74       │  vfmadd231pd  0x20(%rbx,%r13), %ymm3, %ymm1
2.37       │  vfmadd231pd  0x40(%rbx,%r13), %ymm3, %ymm0
0.50       │  addq         $0x4,%rbp
0.04       │  leaq         (%rbx,%rbx),%rcx
1.07       │  addq         %rcx,%r13
0.01       │  cmpq         $0x33,%rbp
0.10       └─ jne          230
```

AVX + cache awareness

Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%
3	+Tiling	3.9 s	3	38.9	12.2	8.3%
4	+Avx	3.158 s	1.23	48.13	15.1	10.2%
5	+Cache	2.709 s	1.165	56.1	17.6	11.95%

CPU registers aware implementation

CPU register aware implementation

```
template<int Nr, int Mr, int Kc, int Nc>
void avx_cache(const double* a,
               const double* mb,
               double* c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8); // Equals 4
    for(int i = 0; i < Mr; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < Kc; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```

CPU register aware implementation

```
template<int Nr, int Mr, int Kc, int Nc>
void avx_cache(const double* a,
               const double* mb,
               double* c,
               int N,
               int K)
{
    constexpr int avx_doubles = 256 / (sizeof(double) * 8); // Equals 4
    for(int i = 0; i < Mr; ++i, c += N, a += K){
        const double* b = mb;
        for(int k = 0; k < Kc; ++k, b += N){
            __m256d a_reg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += avx_doubles){
                __m256d b_reg = _mm256_loadu_pd(&b[j]);
                __m256d c_reg = _mm256_loadu_pd(&c[j]);
                c_reg = _mm256_fmadd_pd(a_reg, b_reg, c_reg);
                _mm256_storeu_pd(&c[j], c_reg);
            }
        }
    }
}
```

Let’s hint to the compiler
how many CPU registers we have.

CPU register aware implementation

```
template<int Nr, int Mr, int Kc, int Nc>
void avx_regs(const double* ma, const double* b, double* c, int N, int K)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();


    for(int k = 0; k < Kc; ++k, b += N){
        const double* a = ma;

        int idx = 0;
        for(int i = 0; i < Mr; ++i, a += K){
            __m256d areg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

CPU register aware implementation

```
template<int Nr, int Mr, int Kc, int Nc>
void avx_regs(const double* ma, const double* b, double* c, int N, int K)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();


    for(int k = 0; k < Kc; ++k, b += N){
        const double* a = ma;

        int idx = 0;
        for(int i = 0; i < Mr; ++i, a += K){
            __m256d areg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

Nr=12, Mr=4

12 registers for c elements

3 registers for b elements

1 register for a elements

—----------

16 total

CPU register aware implementation

```
0.63  230: ┌─→vbroadcastsd -0x18(%r10,%rbp,8), %ymm3
0.58       │  vfmadd231pd  (%r13), %ymm3, %ymm2
2.10       │  vfmadd231pd  0x20(%r13), %ymm3, %ymm1
1.83       │  vfmadd231pd  0x40(%r13), %ymm3, %ymm0
0.09       │  vbroadcastsd -0x10(%r10,%rbp,8), %ymm3
1.45       │  vfmadd231pd  (%r13,%rbx), %ymm3, %ymm2
2.45       │  vfmadd231pd  0x20(%r13,%rbx), %ymm3, %ymm1
2.16       │  vfmadd231pd  0x40(%r13,%rbx), %ymm3, %ymm0
0.34       │  leaq         (%r13,%rbx),%rcx
0.05       │  vbroadcastsd -0x8(%r10,%rbp,8), %ymm3
1.53       │  vfmadd231pd  (%rbx,%rcx), %ymm3, %ymm2
1.95       │  vfmadd231pd  0x20(%rbx,%rcx), %ymm3, %ymm1
2.40       │  vfmadd231pd  0x40(%rbx,%rcx), %ymm3, %ymm0
0.55       │  leaq         (%rcx,%rbx),%r13
0.05       │  vbroadcastsd (%r10,%rbp,8), %ymm3
1.62       │  vfmadd231pd  (%rbx,%r13), %ymm3, %ymm2
1.74       │  vfmadd231pd  0x20(%rbx,%r13), %ymm3, %ymm1
2.37       │  vfmadd231pd  0x40(%rbx,%r13), %ymm3, %ymm0
0.50       │  addq         $0x4,%rbp
```

AVX + cache awareness

```
3.06  240: ┌─→vbroadcastsd (%rdx,%rcx), %ymm12
4.17       │  vmovupd      (%r13), %ymm13
6.02       │  vmovupd      0x20(%r13), %ymm14
4.72       │  vmovupd      0x40(%r13), %ymm15
3.15       │  vfmadd231pd  %ymm13, %ymm12, %ymm11
3.84       │  vfmadd231pd  %ymm14, %ymm12, %ymm10
4.98       │  vfmadd231pd  %ymm12, %ymm15, %ymm9
1.09       │  vbroadcastsd (%r12,%rcx), %ymm12
2.40       │  vfmadd231pd  %ymm13, %ymm12, %ymm8
1.37       │  vfmadd231pd  %ymm14, %ymm12, %ymm7
2.40       │  vfmadd231pd  %ymm12, %ymm15, %ymm6
1.03       │  vbroadcastsd (%rax,%rcx), %ymm12
2.08       │  vfmadd231pd  %ymm13, %ymm12, %ymm5
1.09       │  vfmadd231pd  %ymm14, %ymm12, %ymm4
2.26       │  vfmadd231pd  %ymm12, %ymm15, %ymm3
0.85       │  vbroadcastsd (%rbx,%rcx), %ymm12
2.13       │  vfmadd231pd  %ymm13, %ymm12, %ymm2
1.10       │  vfmadd231pd  %ymm14, %ymm12, %ymm1
2.05       │  vfmadd231pd  %ymm15, %ymm12, %ymm0
```

AVX + cache/regs awareness

CPU register aware implementation

```
0.63  230: ┌─→vbroadcastsd -0x18(%r10,%rbp,8), %ymm3
0.58       │  vfmadd231pd  (%r13), %ymm3, %ymm2
2.10       │  vfmadd231pd  0x20(%r13), %ymm3, %ymm1
1.83       │  vfmadd231pd  0x40(%r13), %ymm3, %ymm0
0.09       │  vbroadcastsd -0x10(%r10,%rbp,8), %ymm3
1.45       │  vfmadd231pd  (%r13,%rbx), %ymm3, %ymm2
2.45       │  vfmadd231pd  0x20(%r13,%rbx), %ymm3, %ymm1
2.16       │  vfmadd231pd  0x40(%r13,%rbx), %ymm3, %ymm0
0.34       │  leaq         (%r13,%rbx),%rcx
0.05       │  vbroadcastsd -0x8(%r10,%rbp,8), %ymm3
1.53       │  vfmadd231pd  (%rbx,%rcx), %ymm3, %ymm2
1.95       │  vfmadd231pd  0x20(%rbx,%rcx), %ymm3, %ymm1
2.40       │  vfmadd231pd  0x40(%rbx,%rcx), %ymm3, %ymm0
0.55       │  leaq         (%rcx,%rbx),%r13
0.05       │  vbroadcastsd (%r10,%rbp,8), %ymm3
1.62       │  vfmadd231pd  (%rbx,%r13), %ymm3, %ymm2
1.74       │  vfmadd231pd  0x20(%rbx,%r13), %ymm3, %ymm1
2.37       │  vfmadd231pd  0x40(%rbx,%r13), %ymm3, %ymm0
0.50       │  addq         $0x4,%rbp
```

AVX + cache awareness

```
3.06  240: ┌─→vbroadcastsd (%rdx,%rcx), %ymm12
4.17       │  vmovupd      (%r13), %ymm13
6.02       │  vmovupd      0x20(%r13), %ymm14
4.72       │  vmovupd      0x40(%r13), %ymm15
3.15       │  vfmadd231pd  %ymm13, %ymm12, %ymm11
3.84       │  vfmadd231pd  %ymm14, %ymm12, %ymm10
4.98       │  vfmadd231pd  %ymm12, %ymm15, %ymm9
1.09       │  vbroadcastsd (%r12,%rcx), %ymm12
2.40       │  vfmadd231pd  %ymm13, %ymm12, %ymm8
1.37       │  vfmadd231pd  %ymm14, %ymm12, %ymm7
2.40       │  vfmadd231pd  %ymm12, %ymm15, %ymm6
1.03       │  vbroadcastsd (%rax,%rcx), %ymm12
2.08       │  vfmadd231pd  %ymm13, %ymm12, %ymm5
1.09       │  vfmadd231pd  %ymm14, %ymm12, %ymm4
2.26       │  vfmadd231pd  %ymm12, %ymm15, %ymm3
0.85       │  vbroadcastsd (%rbx,%rcx), %ymm12
2.13       │  vfmadd231pd  %ymm13, %ymm12, %ymm2
1.10       │  vfmadd231pd  %ymm14, %ymm12, %ymm1
2.05       │  vfmadd231pd  %ymm15, %ymm12, %ymm0
```

AVX + cache/regs awareness

CPU register aware implementation

```
0.63  230: ┌─→vbroadcastsd -0x18(%r10,%rbp,8), %ymm3
0.58       │  vfmadd231pd  (%r13), %ymm3, %ymm2
2.10       │  vfmadd231pd  0x20(%r13), %ymm3, %ymm1
1.83       │  vfmadd231pd  0x40(%r13), %ymm3, %ymm0
0.09       │  vbroadcastsd -0x10(%r10,%rbp,8), %ymm3
1.45       │  vfmadd231pd  (%r13,%rbx), %ymm3, %ymm2
2.45       │  vfmadd231pd  0x20(%r13,%rbx), %ymm3, %ymm1
2.16       │  vfmadd231pd  0x40(%r13,%rbx), %ymm3, %ymm0
0.34       │  leaq         (%r13,%rbx),%rcx
0.05       │  vbroadcastsd -0x8(%r10,%rbp,8), %ymm3
1.53       │  vfmadd231pd  (%rbx,%rcx), %ymm3, %ymm2
1.95       │  vfmadd231pd  0x20(%rbx,%rcx), %ymm3, %ymm1
2.40       │  vfmadd231pd  0x40(%rbx,%rcx), %ymm3, %ymm0
0.55       │  leaq         (%rcx,%rbx),%r13
0.05       │  vbroadcastsd (%r10,%rbp,8), %ymm3
1.62       │  vfmadd231pd  (%rbx,%r13), %ymm3, %ymm2
1.74       │  vfmadd231pd  0x20(%rbx,%r13), %ymm3, %ymm1
2.37       │  vfmadd231pd  0x40(%rbx,%r13), %ymm3, %ymm0
0.50       │  addq         $0x4,%rbp
```

AVX + cache awareness

```
3.06  240: ┌─→vbroadcastsd (%rdx,%rcx), %ymm12
4.17       │  vmovupd      (%r13), %ymm13
6.02       │  vmovupd      0x20(%r13), %ymm14
4.72       │  vmovupd      0x40(%r13), %ymm15
3.15       │  vfmadd231pd  %ymm13, %ymm12, %ymm11
3.84       │  vfmadd231pd  %ymm14, %ymm12, %ymm10
4.98       │  vfmadd231pd  %ymm12, %ymm15, %ymm9
1.09       │  vbroadcastsd (%r12,%rcx), %ymm12
2.40       │  vfmadd231pd  %ymm13, %ymm12, %ymm8
1.37       │  vfmadd231pd  %ymm14, %ymm12, %ymm7
2.40       │  vfmadd231pd  %ymm12, %ymm15, %ymm6
1.03       │  vbroadcastsd (%rax,%rcx), %ymm12
2.08       │  vfmadd231pd  %ymm13, %ymm12, %ymm5
1.09       │  vfmadd231pd  %ymm14, %ymm12, %ymm4
2.26       │  vfmadd231pd  %ymm12, %ymm15, %ymm3
0.85       │  vbroadcastsd (%rbx,%rcx), %ymm12
2.13       │  vfmadd231pd  %ymm13, %ymm12, %ymm2
1.10       │  vfmadd231pd  %ymm14, %ymm12, %ymm1
2.05       │  vfmadd231pd  %ymm15, %ymm12, %ymm0
```

AVX + cache/regs awareness

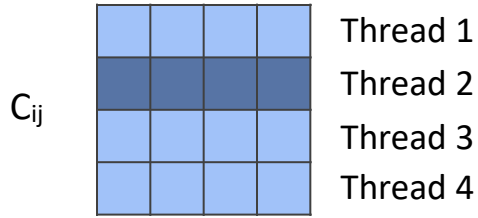
Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%
3	+Tiling	3.9 s	3	38.9	12.2	8.3%
4	+Avx	3.158 s	1.23	48.13	15.1	10.2%
5	+Cache	2.709 s	1.165	56.1	17.6	11.95%
6	+Regs	1.407 s	1.92	108	33.95	23%

Multithreading

Multithreading

- Apply parallelization to multiple loops, both top-level and nested loops
- Pthread pinned to cpu cores

```
#pragma omp parallel for   // No sync needed between threads
    for(int ib = 0; ib < M; ib += Mc){
        for(int kb = 0; kb < K; kb += Kc){
            for(int jb = 0; jb < N; jb += Nc){
                // ...
```



Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%
3	+Tiling	3.9 s	3	38.9	12.2	8.3%
4	+Avx	3.158 s	1.23	48.13	15.1	10.2%
5	+Cache	2.709 s	1.165	56.1	17.6	11.95%
6	+Regs	1.407 s	1.92	108	33.95	23%
7	+Multithreading	0.466 s	3.02	326.1	102.52	69.5%

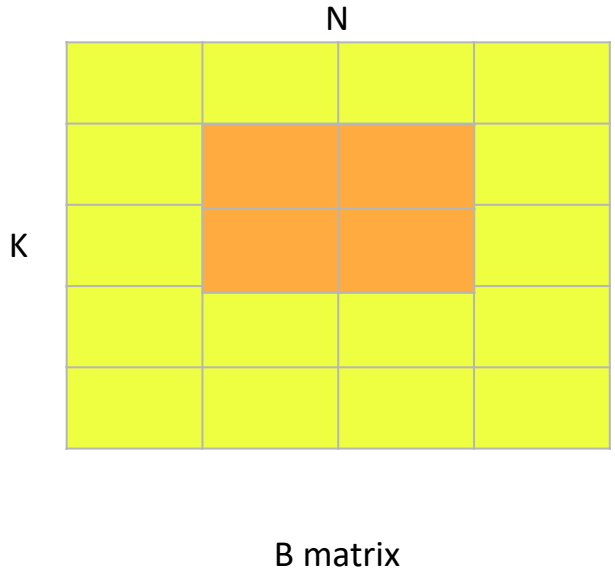
Stride

Stride

```
template<int Nr, int Mr, int Kc, int Nc>
void avx_regs(const double* ma, const double* b, double* c, int N, int K)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();

    for(int k = 0; k < Kc; ++k, b += N){
        const double* a = ma;

        int idx = 0;
        for(int i = 0; i < Mr; ++i, a += K){
            __m256d areg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

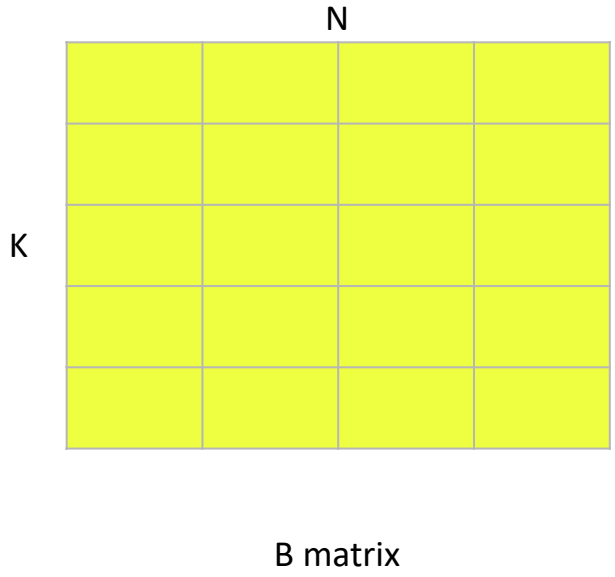
K, N are matrix dimensions.

Stride

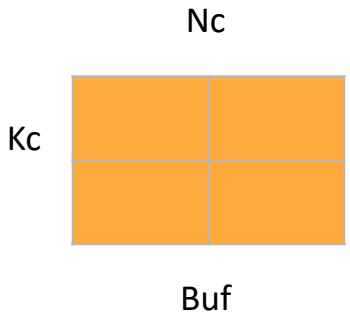


Pack the matrix to minimize stride.

Stride



Pack the matrix to minimize stride.



Stride

```
template<int Nr, int Mr, int Kc, int Nc>
void avx_regs(const double* ma, const double* b, double* c, int N, int K)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();


    for(int k = 0; k < Kc; ++k, b += N){
        const double* a = ma;

        int idx = 0;
        for(int i = 0; i < Mr; ++i, a += K){
            __m256d areg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

Stride

```
template<int Nr, int Mr, int Kc, int Nc>
void avx_regs_bpack(const double* ma, const double* b, double* c, int N, int K)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();

    for(int k = 0; k < Kc; ++k, b += Nc){
        const double* a = ma;

        int idx = 0;
        for(int i = 0; i < Mr; ++i, a += K){
            __m256d areg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

Packing matrix A had no impact.

Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%
3	+Tiling	3.9 s	3	38.9	12.2	8.3%
4	+Avx	3.158 s	1.23	48.13	15.1	10.2%
5	+Cache	2.709 s	1.165	56.1	17.6	11.95%
6	+Regs	1.407 s	1.92	108	33.95	23%
7	+Multithreading	0.466 s	3.02	326.1	102.52	69.5%
8	+Pack	0.398 s	1.17	381.9	120.03	81.38%

Matrix tile layout

Tile layout

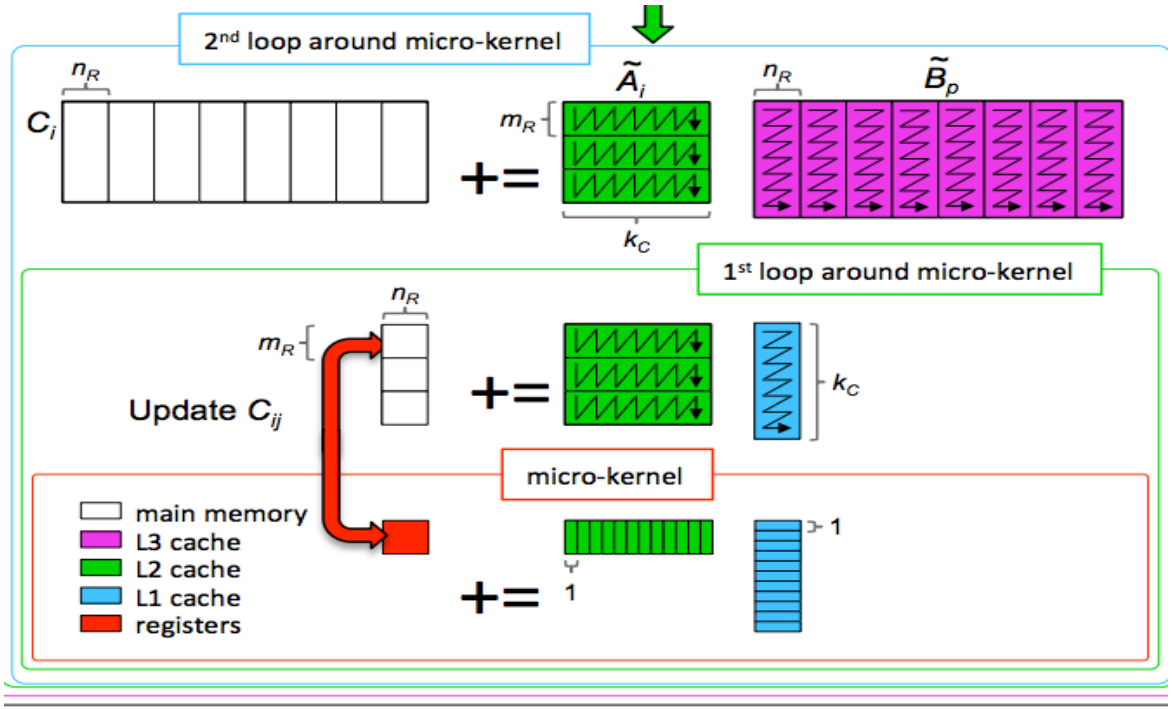
```
template<int Nr, int Mr, int Kc, int Nc>
void avx_regs_bpack(const double* ma, const double* b, double* c, int N, int K)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();

    for(int k = 0; k < Kc; ++k, b += Nc){
        const double* a = ma;

        int idx = 0;
        for(int i = 0; i < Mr; ++i, a += K){
            __m256d areg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

Tile layout

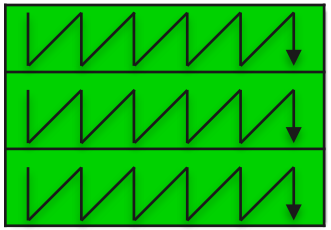
```
template<int Nr, int Mr, int Kc, int Nc>
void avx_regs_bpack(const double* ma, const double* b, double* c, int N, int K)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();

    for(int k = 0; k < Kc; ++k, b += Nc){
        const double* a = ma;

        int idx = 0;
        for(int i = 0; i < Mr; ++i, a += K){
            __m256d areg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

What if we make stride = 1?

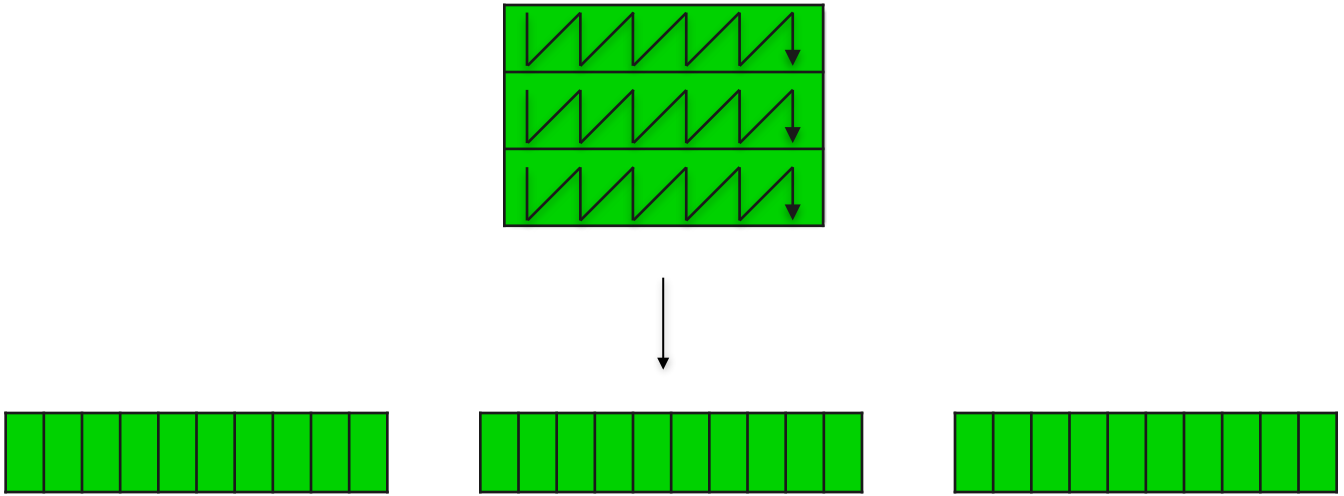
Tile layout


2nd loop around micro-kernel
n_R
C_i
$\tilde{A}_i$
m_R
k_C
$\tilde{B}_p$
1st loop around micro-kernel
Update C_{ij}
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

Tile layout



Tile layout



Tile layout

```
template<int Nr, int Mr, int Kc, int Nc>
void avx_regs_bpack(const double* ma, const double* b, double* c, int N, int K)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();

    for(int k = 0; k < Kc; ++k, b += Nc){
        const double* a = ma;
        int idx = 0;
        for(int i = 0; i < Mr; ++i, a += K){
            __m256d areg = _mm256_broadcast_sd(&a[k]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

Tile layout

```
template<int Nr, int Mr, int Kc>
void avx_regs_reordered(const double* a, const double* b, double* c, int N)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();

    for(int k = 0; k < Kc; ++k, b += Nr, a += Mr){

        int idx = 0;
        for(int i = 0; i < Mr; ++i){
            __m256d areg = _mm256_broadcast_sd(&a[i]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

Tile layout

```
template<int Nr, int Mr, int Kc>
void avx_regs_reordered(const double* a, const double* b, double* c, int N)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();

    for(int k = 0; k < Kc; ++k, b += Nr, a += Mr){

        int idx = 0;
        for(int i = 0; i < Mr; ++i){
            __m256d areg = _mm256_broadcast_sd(&a[i]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

Nr=12

Mr=4

Tile layout

```
template<int Nr, int Mr, int Kc>
void avx_regs_reordered(const double* a, const double* b, double* c, int N)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();

    for(int k = 0; k < Kc; ++k, b += Nr, a += Mr){

        int idx = 0;
        for(int i = 0; i < Mr; ++i){
            __m256d areg = _mm256_broadcast_sd(&a[i]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

```
vbroadcastsd (%rdi,%r11), %ymm12
vmovupd      (%rbx), %ymm13
vmovupd      0x20(%rbx), %ymm14
vmovupd      0x40(%rbx), %ymm15
vfmadd231pd  %ymm13, %ymm12, %ymm11
vfmadd231pd  %ymm12, %ymm14, %ymm10
vfmadd231pd  %ymm12, %ymm15, %ymm9
vbroadcastsd 0x8(%rdi,%r11), %ymm12
vfmadd231pd  %ymm13, %ymm12, %ymm8
vfmadd231pd  %ymm14, %ymm12, %ymm7
vfmadd231pd  %ymm12, %ymm15, %ymm6
vbroadcastsd 0x10(%rdi,%r11), %ymm12
vfmadd231pd  %ymm13, %ymm12, %ymm5
vfmadd231pd  %ymm14, %ymm12, %ymm4
vfmadd231pd  %ymm12, %ymm15, %ymm3
vbroadcastsd 0x18(%rdi,%r11), %ymm12
vfmadd231pd  %ymm13, %ymm12, %ymm2
vfmadd231pd  %ymm14, %ymm12, %ymm1
vfmadd231pd  %ymm15, %ymm12, %ymm0
addq         $0x20,%r11
addq         $0x60,%rbx
cmpl         $0xb40,%r11d
jne          3a00
```

Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%
3	+Tiling	3.9 s	3	38.9	12.2	8.3%
4	+Avx	3.158 s	1.23	48.13	15.1	10.2%
5	+Cache	2.709 s	1.165	56.1	17.6	11.95%
6	+Regs	1.407 s	1.92	108	33.95	23%
7	+Multithreading	0.466 s	3.02	326.1	102.52	69.5%
8	+Pack	0.398 s	1.17	381.9	120.03	81.38%
9	+Reorder	0.360 s	1.10	422.22	132.7	89.9%

SIMD

SIMD

```
template<int Nr, int Mr, int Kc>
void avx_regs_reordered(const double* a, const double* b, double* c, int N)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<__m256d, CREG_CNT> res;
    for(int idx = 0; idx < CREG_CNT; ++idx)
        res[idx] = _mm256_setzero_pd();

    for(int k = 0; k < Kc; ++k, b += Nr, a += Mr){

        int idx = 0;
        for(int i = 0; i < Mr; ++i){
            __m256d areg = _mm256_broadcast_sd(&a[i]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                res[idx] = _mm256_fmadd_pd(areg, _mm256_loadu_pd(&b[j]), res[idx]);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

SIMD

```
template<int Nr, int Mr, int Kc>
void avx_regs_reordered_simd(const double* a, const double* b, double* c, int N)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<simd_d, CREG_CNT> c_reg{};



    for(int k = 0; k < Kc; ++k, b += Nr, a += Mr){
        int idx = 0;
        for(int i = 0; i < Mr; ++i){
            simd_d a_reg(a[i]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                c_reg[idx] += a_reg * simd_d(&b[j], stdx::element_aligned);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

SIMD

```
template<int Nr, int Mr, int Kc>
void avx_regs_reordered_simd(const double* a, const double* b, double* c, int N)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<simd_d, CREG_CNT> c_reg{};



    for(int k = 0; k < Kc; ++k, b += Nr, a += Mr){
        int idx = 0;
        for(int i = 0; i < Mr; ++i){
            simd_d a_reg(a[i]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                c_reg[idx] += a_reg * simd_d(&b[j], stdx::element_aligned);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

Zero overhead

Loop unrolling

Loop unrolling

```
template<int Nr, int Mr, int Kc>
void avx_regs_reordered_simd(const double* a, const double* b, double* c, int N)
{
    constexpr int CREG_CNT{Mr * Nr / 4};
    std::array<simd_d, CREG_CNT> c_reg{};



    for(int k = 0; k < Kc; ++k, b += Nr, a += Mr){
        int idx = 0;
        for(int i = 0; i < Mr; ++i){
            simd_d a_reg(a[i]);
            for(int j = 0; j < Nr; j += 4, ++idx){
                c_reg[idx] += a_reg * simd_d(&b[j], stdx::element_aligned);
            }
        }
    }

    int idx = 0;
    for(int i = 0; i < Mr; ++i, c += N){
        for(int j = 0; j < Nr; j += 4, ++idx){
            load_inc_store_double(&c[j], res[idx]);
        }
    }
}
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr){
    int idx = 0;
    for(int i = 0; i < Mr; ++i){
        simd_d a_reg(a[i]);
        for(int j = 0; j < Nr; j += 4, ++idx){
            auto aligned = stdx::element_aligned;
            c_reg[idx] += a_reg * simd_d(&b[j], aligned);
        }
    }
}
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr){
    int idx = 0;
    for(int i = 0; i < Mr; ++i){
        simd_d a_reg(a[i]);
        for(int j = 0; j < Nr; j += 4, ++idx){
            auto aligned = stdx::element_aligned;
            c_reg[idx] += a_reg * simd_d(&b[j], aligned);
        }
    }
}
```

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b1(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    // ...
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b1(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    // ...
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b1(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    // ...
```

Pack (since C++11)

A pack is a C++ entity that defines one of the following:

- a parameter pack
  - template parameter pack
  - function parameter pack

Fold expressions (since C++17)

Reduces (folds) a pack over a binary operator.

Syntax

(*pack op* ...)	(1)
(... *op pack*)	(2)

Built-in comma operator

Comma expressions have the following form:

E1 , *E2*

In a comma expression `E1, E2`, the expression `E1` is evaluated, its result is discarded

Loop unrolling

```
void print(std::size_t x) {
    std::cout << "Value: " << x << '\n';
}
print(0); print(1); print(2); print(3);
```

Loop unrolling

```
void print(std::size_t x) {
    std::cout << "Value: " << x << '\n';
}
print(0); print(1); print(2); print(3);


template<std::size_t... Idx>
void callPrintNTimes() {
    (print(Idx), ...); // fold expression over comma operator
}

callPrintNTimes<0,1,2,3>();
```

Loop unrolling

```
void print(std::size_t x) {
    std::cout << "Value: " << x << '\n';
}
print(0); print(1); print(2); print(3);


template<std::size_t... Idx>
void callPrintNTimes(std::index_sequence<Idx...>) {
    (print(Idx), ...); // fold expression over comma operator
}

callPrintNTimes(std::make_index_sequence<4>{});
```

Loop unrolling

```
void print(std::size_t x) {
    std::cout << "Value: " << x << '\n';
}
print(0); print(1); print(2); print(3);


template<std::size_t... Idx>
void callPrintNTimes(std::index_sequence<Idx...>) {
    (print(Idx), ...); // fold expression over comma operator
}

callPrintNTimes(std::make_index_sequence<4>{});
```

```
template< class T, T N >
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */>;
```

```
template< std::size_t N >
using make_index_sequence = std::make_integer_sequence<std::size_t, N>;
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b2(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    r[2] += a0 * b2;

    a0 = simd_d(a[1]);
    r[3] += a0 * b0;
    r[4] += a0 * b1;
    r[5] += a0 * b2;
    // Mr times ...
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b2(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    r[2] += a0 * b2;

    a0 = simd_d(a[1]);
    r[3] += a0 * b0;
    r[4] += a0 * b1;
    r[5] += a0 * b2;
    // Mr times ...
```

```
template<typename T, size_t... I, size_t... J>
void compute_kernel(const T* a,
                    const T* b,
                    simd<T>* r,
                    std::index_sequence<I...>,
                    std::index_sequence<J...>)
{
    constexpr auto Nrs = sizeof...(J);
    constexpr auto seq =  std::make_index_sequence<Nrs>{};
    constexpr auto align = stdx::element_aligned;

    simd<T> bs[Nrs] = {simd<T>(&b[J * simd<T>::size()], align)...};
    (..., (compute_row(simd<T>(a[I]), bs, &r[I * Nrs], seq)));
}
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b2(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    r[2] += a0 * b2;

    a0 = simd_d(a[1]);
    r[3] += a0 * b0;
    r[4] += a0 * b1;
    r[5] += a0 * b2;
    // Mr times ...
```

```
template<typename T, size_t... I, size_t... J>
void compute_kernel(const T* a,
                    const T* b,
                    simd<T>* r,
                    std::index_sequence<I...>,
                    std::index_sequence<J...>)
{
    constexpr auto Nrs = sizeof...(J);
    constexpr auto seq =  std::make_index_sequence<Nrs>{};
    constexpr auto align = stdx::element_aligned;

    simd<T> bs[Nrs] = {simd<T>(&b[J * simd<T>::size()], align)...};
    (..., (compute_row(simd<T>(a[I]), bs, &r[I * Nrs], seq)));
}
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b2(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    r[2] += a0 * b2;

    a0 = simd_d(a[1]);
    r[3] += a0 * b0;
    r[4] += a0 * b1;
    r[5] += a0 * b2;
    // Mr times ...
```

```
template<typename T, size_t... I, size_t... J>
void compute_kernel(const T* a,
                    const T* b,
                    simd<T>* r,
                    std::index_sequence<I...>,
                    std::index_sequence<J...>)
{
    constexpr auto Nrs = sizeof...(J);
    constexpr auto seq =  std::make_index_sequence<Nrs>{};
    constexpr auto align = stdx::element_aligned;

    simd<T> bs[Nrs] = {simd<T>(&b[J * simd<T>::size()], align)...};
    (..., (compute_row(simd<T>(a[I]), bs, &r[I * Nrs], seq)));
}
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b2(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    r[2] += a0 * b2;

    a0 = simd_d(a[1]);
    r[3] += a0 * b0;
    r[4] += a0 * b1;
    r[5] += a0 * b2;
    // Mr times ...
```

```
template<typename T, size_t... I, size_t... J>
void compute_kernel(const T* a,
                    const T* b,
                    simd<T>* r,
                    std::index_sequence<I...>,
                    std::index_sequence<J...>)
{
    constexpr auto Nrs = sizeof...(J);
    constexpr auto seq =  std::make_index_sequence<Nrs>{};
    constexpr auto align = stdx::element_aligned;

    simd<T> bs[Nrs] = {simd<T>(&b[J * simd<T>::size()], align)...};
    (..., (compute_row(simd<T>(a[I]), bs, &r[I * Nrs], seq)));
}
```

```
template<typename T, std::size_t... J>
void compute_row(const simd<T>& a, simd<T>* b, simd<T>* r, std::index_sequence<J...>)
{
    (..., (r[J] += a * b[J]));
}
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b2(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    r[2] += a0 * b2;

    a0 = simd_d(a[1]);
    r[3] += a0 * b0;
    r[4] += a0 * b1;
    r[5] += a0 * b2;
    // Mr times ...
```

```
compute_kernel(a, b, r,
    std::make_index_sequence<Mr>{},
    std::make_index_sequence<Nrs>{});
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b2(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    r[2] += a0 * b2;

    a0 = simd_d(a[1]);
    r[3] += a0 * b0;
    r[4] += a0 * b1;
    r[5] += a0 * b2;
    // Mr times ...
```

```
compute_kernel_impl(a, b, r,
    std::make_index_sequence<Mr>{},
    std::make_index_sequence<Nrs>{});

template<int Mr, int Nrs, typename T>
void compute_kernel(const T* a,
                    const T* b,
                    simd<T>* r)
{
    compute_kernel_impl(a, b, r,
        std::make_index_sequence<Mr>{},
        std::make_index_sequence<Nrs>{});
}
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    simd_d b0(&b[0], stdx::element_aligned);
    simd_d b1(&b[4], stdx::element_aligned);
    simd_d b2(&b[8], stdx::element_aligned);

    simd_d a0(a[0]);
    r[0] += a0 * b0;
    r[1] += a0 * b1;
    r[2] += a0 * b2;

    a0 = simd_d(a[1]);
    r[3] += a0 * b0;
    r[4] += a0 * b1;
    r[5] += a0 * b2;
    // Mr times ...
```

```
compute_kernel_impl(a, b, r,
    std::make_index_sequence<Mr>{},
    std::make_index_sequence<Nrs>{});

compute_kernel<Mr,Nrs>(a, b, r);
```

Loop unrolling

```
for(int k = 0; k < Kc; ++k, b += Nr, a += Mr)
{
    compute_kernel<Mr,Nrs>(a, b, r);
}
```

Loop unrolling

```
load_inc_store_double(&c[0], r[0]);
load_inc_store_double(&c[4], r[1]);
load_inc_store_double(&c[8], r[2]);
c += N;

load_inc_store_double(&c[0], r[3]);
load_inc_store_double(&c[4], r[4]);
load_inc_store_double(&c[8], r[5]);
c += N;
// ...
```

Loop unrolling

```
load_inc_store_double(&c[0], r[0]);
load_inc_store_double(&c[4], r[1]);
load_inc_store_double(&c[8], r[2]);
c += N;

load_inc_store_double(&c[0], r[3]);
load_inc_store_double(&c[4], r[4]);
load_inc_store_double(&c[8], r[5]);
c += N;
// ...
```

```
template<std::size_t RowIdx, typename T, int WIDTH, std::size_t... I>
void store_row(T* c, simd<T, WIDTH>* r, std::index_sequence<I...>)
{
    (..., (load_inc_store_double(&c[I * WIDTH], r[RowIdx * sizeof...(I) + I])));
}
```

Loop unrolling

```
load_inc_store_double(&c[0], r[0]);
load_inc_store_double(&c[4], r[1]);
load_inc_store_double(&c[8], r[2]);
c += N;

load_inc_store_double(&c[0], r[3]);
load_inc_store_double(&c[4], r[4]);
load_inc_store_double(&c[8], r[5]);
c += N;
// ...
```

```
template<std::size_t RowIdx, typename T, int WIDTH, std::size_t... I>
void store_row(T* c, simd<T, WIDTH>* r, std::index_sequence<I...>)
{
    (..., (load_inc_store_double(&c[I * WIDTH], r[RowIdx * sizeof...(I) + I])));
}

template<int Nrs, typename T, int WIDTH, std::size_t... RowIndices>
void store_kernel(T* c,
                  simd<T, WIDTH>*  r,
                  int              N,
                  std::index_sequence<RowIndices...>)
{
    constexpr auto seq = std::make_index_sequence<Nrs>{};
    (..., (store_row<RowIndices>(c, r, seq), c += N));
}
```

Loop unrolling

```
load_inc_store_double(&c[0], r[0]);
load_inc_store_double(&c[4], r[1]);
load_inc_store_double(&c[8], r[2]);
c += N;

load_inc_store_double(&c[0], r[3]);
load_inc_store_double(&c[4], r[4]);
load_inc_store_double(&c[8], r[5]);
c += N;
// ...
```

```
store_kernel<Mr, Nrs>(c, r, N);
```

Loop unrolling

```
template<int Nr, int Mr, int Kc, std::floating_point T>
void cpp_packed_kernel(const T* a, const T* b, T* c, int N)
    requires(Nr % 256_bits == 0)
{
    constexpr int Nrs{Nr / 256_bits};
    simd<T, 256_bits> r[Nrs * Mr] = {};
    for (int k = 0; k < Kc; ++k, b += Nr, a += Mr)
        compute_kernel<Mr,Nrs>(a, b, r);

    store_kernel<Mr, Nrs>(c, r, N);
}
```

Loop unrolling

```
7.30 | 3a00: ┌─→vbroadcastsd (%rdi,%r11), %ymm12
2.76 |        |  vmovupd      (%rbx), %ymm13
6.45 |        |  vmovupd      0x20(%rbx), %ymm14
3.40 |        |  vmovupd      0x40(%rbx), %ymm15
7.78 |        |  vfmadd231pd  %ymm13, %ymm12, %ymm11
2.17 |        |  vfmadd231pd  %ymm12, %ymm14, %ymm10
5.54 |        |  vfmadd231pd  %ymm12, %ymm15, %ymm9
1.15 |        |  vbroadcastsd 0x8(%rdi,%r11), %ymm12
5.52 |        |  vfmadd231pd  %ymm13, %ymm12, %ymm8
1.39 |        |  vfmadd231pd  %ymm14, %ymm12, %ymm7
5.50 |        |  vfmadd231pd  %ymm12, %ymm15, %ymm6
1.09 |        |  vbroadcastsd 0x10(%rdi,%r11), %ymm12
```

Auto unrolling

```
6.84 | 3a90: ┌─→vmovupd      (%r15), %ymm12
2.95 |       |  vmovupd      0x20(%r15), %ymm13
7.01 |       |  vmovupd      0x40(%r15), %ymm14
2.66 |       |  vbroadcastsd (%r8,%rbx), %ymm15
5.97 |       |  vfmadd231pd  %ymm12, %ymm15, %ymm9
2.26 |       |  vfmadd231pd  %ymm13, %ymm15, %ymm11
6.40 |       |  vfmadd231pd  %ymm15, %ymm14, %ymm10
1.44 |       |  vbroadcastsd 0x8(%r8,%rbx), %ymm15
5.48 |       |  vfmadd231pd  %ymm12, %ymm15, %ymm8
1.20 |       |  vfmadd231pd  %ymm13, %ymm15, %ymm7
5.98 |       |  vfmadd231pd  %ymm15, %ymm14, %ymm6
1.42 |       |  vbroadcastsd 0x10(%r8,%rbx), %ymm15
```

Manual unrolling

Loop unrolling

```
7.30 | 3a00: ┌─→vbroadcastsd (%rdi,%r11), %ymm12
2.76 |        |  vmovupd      (%rbx), %ymm13
6.45 |        |  vmovupd      0x20(%rbx), %ymm14
3.40 |        |  vmovupd      0x40(%rbx), %ymm15
7.78 |        |  vfmadd231pd  %ymm13, %ymm12, %ymm11
2.17 |        |  vfmadd231pd  %ymm12, %ymm14, %ymm10
5.54 |        |  vfmadd231pd  %ymm12, %ymm15, %ymm9
1.15 |        |  vbroadcastsd 0x8(%rdi,%r11), %ymm12
5.52 |        |  vfmadd231pd  %ymm13, %ymm12, %ymm8
1.39 |        |  vfmadd231pd  %ymm14, %ymm12, %ymm7
5.50 |        |  vfmadd231pd  %ymm12, %ymm15, %ymm6
1.09 |        |  vbroadcastsd 0x10(%rdi,%r11), %ymm12
```

Auto unrolling

```
6.84 | 3a90: ┌─→vmovupd      (%r15), %ymm12
2.95 |       |  vmovupd      0x20(%r15), %ymm13
7.01 |       |  vmovupd      0x40(%r15), %ymm14
2.66 |       |  vbroadcastsd (%r8,%rbx), %ymm15
5.97 |       |  vfmadd231pd  %ymm12, %ymm15, %ymm9
2.26 |       |  vfmadd231pd  %ymm13, %ymm15, %ymm11
6.40 |       |  vfmadd231pd  %ymm15, %ymm14, %ymm10
1.44 |       |  vbroadcastsd 0x8(%r8,%rbx), %ymm15
5.48 |       |  vfmadd231pd  %ymm12, %ymm15, %ymm8
1.20 |       |  vfmadd231pd  %ymm13, %ymm15, %ymm7
5.98 |       |  vfmadd231pd  %ymm15, %ymm14, %ymm6
1.42 |       |  vbroadcastsd 0x10(%r8,%rbx), %ymm15
```

Manual unrolling

Loop unrolling

```
7.30 | 3a00: ┌─→vbroadcastsd (%rdi,%r11), %ymm12
2.76 |        |  vmovupd     (%rbx), %ymm13
6.45 |        |  vmovupd     0x20(%rbx), %ymm14
3.40 |        |  vmovupd     0x40(%rbx), %ymm15
7.78 |        |  vfmadd231pd %ymm13, %ymm12, %ymm11
2.17 |        |  vfmadd231pd %ymm12, %ymm14, %ymm10
5.54 |        |  vfmadd231pd %ymm12, %ymm15, %ymm9
1.15 |        |  vbroadcastsd 0x8(%rdi,%r11), %ymm12
5.52 |        |  vfmadd231pd %ymm13, %ymm12, %ymm8
1.39 |        |  vfmadd231pd %ymm14, %ymm12, %ymm7
5.50 |        |  vfmadd231pd %ymm12, %ymm15, %ymm6
1.09 |        |  vbroadcastsd 0x10(%rdi,%r11), %ymm12
```

Auto unrolling

```
6.84 | 3a90: ┌─→vmovupd     (%r15), %ymm12
2.95 |        |  vmovupd     0x20(%r15), %ymm13
7.01 |        |  vmovupd     0x40(%r15), %ymm14
2.66 |        |  vbroadcastsd (%r8,%rbx), %ymm15
5.97 |        |  vfmadd231pd %ymm12, %ymm15, %ymm9
2.26 |        |  vfmadd231pd %ymm13, %ymm15, %ymm11
6.40 |        |  vfmadd231pd %ymm15, %ymm14, %ymm10
1.44 |        |  vbroadcastsd 0x8(%r8,%rbx), %ymm15
5.48 |        |  vfmadd231pd %ymm12, %ymm15, %ymm8
1.20 |        |  vfmadd231pd %ymm13, %ymm15, %ymm7
5.98 |        |  vfmadd231pd %ymm15, %ymm14, %ymm6
1.42 |        |  vbroadcastsd 0x10(%r8,%rbx), %ymm15
```

Manual unrolling

Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%
3	+Tiling	3.9 s	3	38.9	12.2	8.3%
4	+Avx	3.158 s	1.23	48.13	15.1	10.2%
5	+Cache	2.709 s	1.165	56.1	17.6	11.95%
6	+Regs	1.407 s	1.92	108	33.95	23%
7	+Multithreading	0.466 s	3.02	326.1	102.52	69.5%
8	+Pack	0.398 s	1.17	381.9	120.03	81.38%
9	+Reorder	0.360 s	1.10	422.22	132.7	89.9%
10	+Unroll	0.350 s	1.02	434.28	136.5	92.7%

Loop unrolling

```
7.30 | 3a00: ┌─→vbroadcastsd (%rdi,%r11), %ymm12
2.76 |        |  vmovupd     (%rbx), %ymm13
6.45 |        |  vmovupd     0x20(%rbx), %ymm14
3.40 |        |  vmovupd     0x40(%rbx), %ymm15
7.78 |        |  vfmadd231pd %ymm13, %ymm12, %ymm11
2.17 |        |  vfmadd231pd %ymm12, %ymm14, %ymm10
5.54 |        |  vfmadd231pd %ymm12, %ymm15, %ymm9
1.15 |        |  vbroadcastsd 0x8(%rdi,%r11), %ymm12
5.52 |        |  vfmadd231pd %ymm13, %ymm12, %ymm8
1.39 |        |  vfmadd231pd %ymm14, %ymm12, %ymm7
5.50 |        |  vfmadd231pd %ymm12, %ymm15, %ymm6
1.09 |        |  vbroadcastsd 0x10(%rdi,%r11), %ymm12
```

Auto unrolling

```
6.84 | 3a90: ┌─→vmovupd     (%r15), %ymm12
2.95 |       |  vmovupd     0x20(%r15), %ymm13
7.01 |       |  vmovupd     0x40(%r15), %ymm14
2.66 |       |  vbroadcastsd (%r8,%rbx), %ymm15
5.97 |       |  vfmadd231pd %ymm12, %ymm15, %ymm9
2.26 |       |  vfmadd231pd %ymm13, %ymm15, %ymm11
6.40 |       |  vfmadd231pd %ymm15, %ymm14, %ymm10
1.44 |       |  vbroadcastsd 0x8(%r8,%rbx), %ymm15
5.48 |       |  vfmadd231pd %ymm12, %ymm15, %ymm8
1.20 |       |  vfmadd231pd %ymm13, %ymm15, %ymm7
5.98 |       |  vfmadd231pd %ymm15, %ymm14, %ymm6
1.42 |       |  vbroadcastsd 0x10(%r8,%rbx), %ymm15
```

Manual unrolling

Prefetching

Prefetching

```
void _mm_prefetch(const void *p, int hint);
```

Prefetching

```
void _mm_prefetch(const void *p, int hint);

    _mm_prefetch(c, _MM_HINT_NTA);
```

Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%
3	+Tiling	3.9 s	3	38.9	12.2	8.3%
4	+Avx	3.158 s	1.23	48.13	15.1	10.2%
5	+Cache	2.709 s	1.165	56.1	17.6	11.95%
6	+Regs	1.407 s	1.92	108	33.95	23%
7	+Multithreading	0.466 s	3.02	326.1	102.52	69.5%
8	+Pack	0.398 s	1.17	381.9	120.03	81.38%
9	+Reorder	0.360 s	1.10	422.22	132.7	89.9%
10	+Unroll + prefetch	0.336s	1.07	452.4	142.2	96.6%

OpenBlas

Version	Implementation	Execution time	Relative speedup	Absolute speedup	GFLOPS	Percent of peak performance
1	Naive	152 s	NA	NA	0.314	0.2%
2	+Order	12 s	12.6	12.6	3.98	2.7%
3	+Tiling	3.9 s	3	38.9	12.2	8.3%
4	+Avx	3.158 s	1.23	48.13	15.1	10.2%
5	+Cache	2.709 s	1.165	56.1	17.6	11.95%
6	+Regs	1.407 s	1.92	108	33.95	23%
7	+Multithreading	0.466 s	3.02	326.1	102.52	69.5%
8	+Pack	0.398 s	1.17	381.9	120.03	81.38%
9	+Reorder	0.360 s	1.10	422.22	132.7	89.9%
10	+Unroll + prefetch	0.336s	1.07	452.4	142.2	96.6%
11	OpenBlas	0.310 s	1.08	490.322	154.1	104%

Why it is hard to reason about the code performance?

Performance Reasoning

```
 6.84 3a90:  vmovupd      (%r15), %ymm12
 2.95        vmovupd      0x20(%r15), %ymm13
 7.01        vmovupd      0x40(%r15), %ymm14
 2.66        vbroadcastsd (%r8,%rbx), %ymm15
 5.97        vfmadd231pd  %ymm12, %ymm15, %ymm9
 2.26        vfmadd231pd  %ymm13, %ymm15, %ymm11
 6.40        vfmadd231pd  %ymm15, %ymm14, %ymm10
 1.44        vbroadcastsd 0x8(%r8,%rbx), %ymm15
 5.48        vfmadd231pd  %ymm12, %ymm15, %ymm8
 1.20        vfmadd231pd  %ymm13, %ymm15, %ymm7
 5.98        vfmadd231pd  %ymm15, %ymm14, %ymm6
 1.42        vbroadcastsd 0x10(%r8,%rbx), %ymm15
 4.57        vfmadd231pd  %ymm12, %ymm15, %ymm3
 1.11        vfmadd231pd  %ymm13, %ymm15, %ymm4
 6.37        vfmadd231pd  %ymm15, %ymm14, %ymm5
 1.29        vbroadcastsd 0x18(%r8,%rbx), %ymm15
 6.74        vfmadd231pd  %ymm12, %ymm15, %ymm2
 1.86        vfmadd231pd  %ymm13, %ymm15, %ymm1
 5.59        vfmadd231pd  %ymm14, %ymm15, %ymm0
 1.05        addq         $0x20,%rbx
 4.69        addq         $0x60,%r15
 0.01        cmpl         $0xb40,%ebx
 1.81        jne          3a90
```

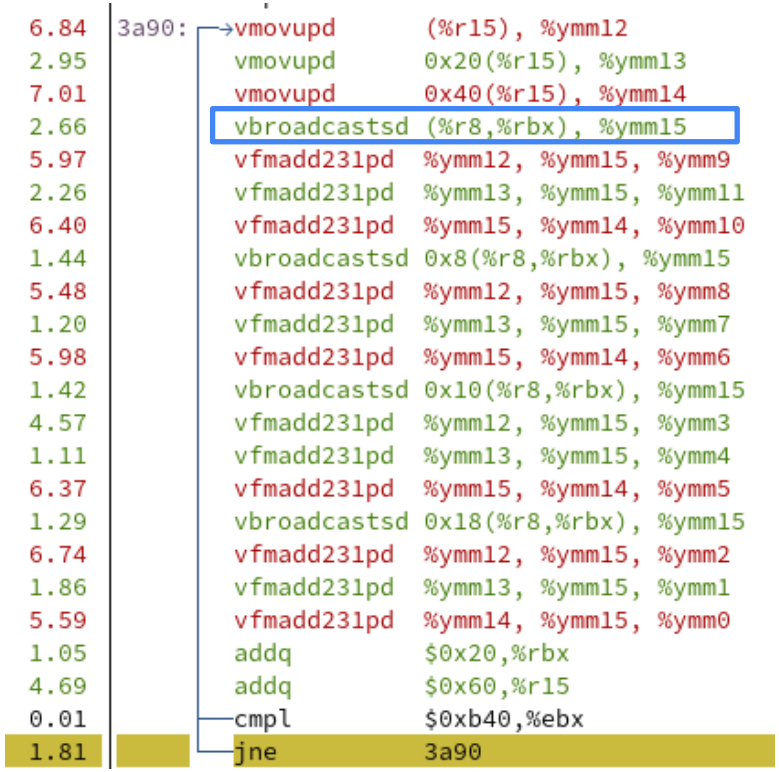
- There’s no point in analyzing C++ code.

Performance Reasoning

```
 6.84 3a90:  vmovupd      (%r15), %ymm12
 2.95        vmovupd      0x20(%r15), %ymm13
 7.01        vmovupd      0x40(%r15), %ymm14
 2.66        vbroadcastsd (%r8,%rbx), %ymm15
 5.97        vfmadd231pd  %ymm12, %ymm15, %ymm9
 2.26        vfmadd231pd  %ymm13, %ymm15, %ymm11
 6.40        vfmadd231pd  %ymm15, %ymm14, %ymm10
 1.44        vbroadcastsd 0x8(%r8,%rbx), %ymm15
 5.48        vfmadd231pd  %ymm12, %ymm15, %ymm8
 1.20        vfmadd231pd  %ymm13, %ymm15, %ymm7
 5.98        vfmadd231pd  %ymm15, %ymm14, %ymm6
 1.42        vbroadcastsd 0x10(%r8,%rbx), %ymm15
 4.57        vfmadd231pd  %ymm12, %ymm15, %ymm3
 1.11        vfmadd231pd  %ymm13, %ymm15, %ymm4
 6.37        vfmadd231pd  %ymm15, %ymm14, %ymm5
 1.29        vbroadcastsd 0x18(%r8,%rbx), %ymm15
 6.74        vfmadd231pd  %ymm12, %ymm15, %ymm2
 1.86        vfmadd231pd  %ymm13, %ymm15, %ymm1
 5.59        vfmadd231pd  %ymm14, %ymm15, %ymm0
 1.05        addq         $0x20,%rbx
 4.69        addq         $0x60,%r15
 0.01        cmpl         $0xb40,%ebx
 1.81        jne          3a90
```

- There’s no point in analyzing C++ code.
- Analyzing instructions for slowness doesn’t lead to the same conclusions as analyzing functions in C++.

Performance Reasoning



- There’s no point in analyzing C++ code.
- Analyzing instructions for slowness doesn’t lead to the same conclusions as analyzing functions in C++.
- Instruction level parallelism

Performance Reasoning

https://tavianator.com/2025/shlx.html

Let's try initializing `rcx` differently:

```
 .LOOP:
-        mov rcx, 1
+        mov ecx, 1
```

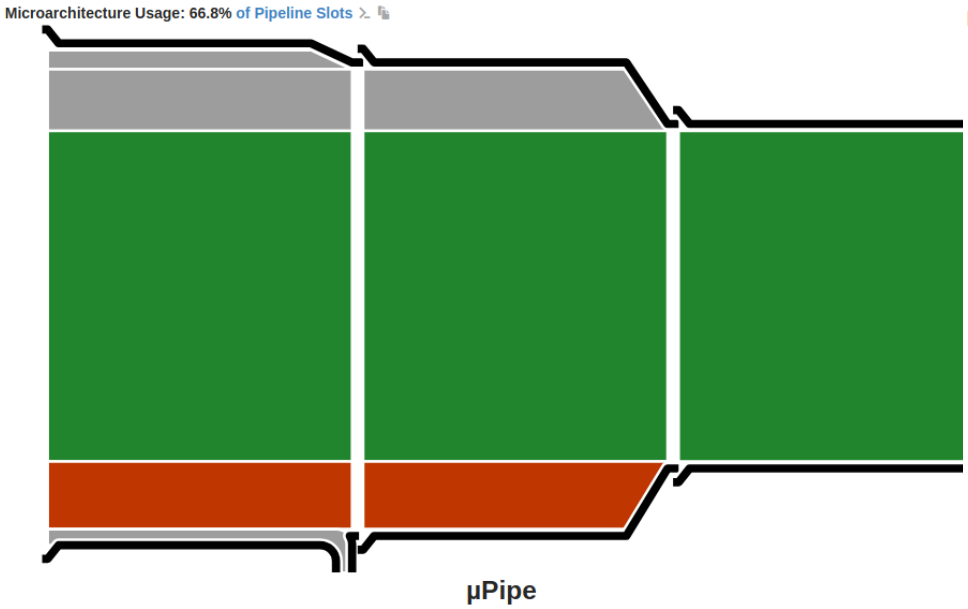
`ecx` is the 32-bit low half of the 64-bit `rcx` register. On x86-64, writing a 32-bit register implicitly sets the upper half of the corresponding 64-bit register to zero. So these two instructions should behave identically. And yet:

```
Performance counter stats for './shlx':

       100,321,870      cpu_core/cycles:u/
       100,155,867      cpu_core/instructions:u/         #    1.00  insn per cycle
```

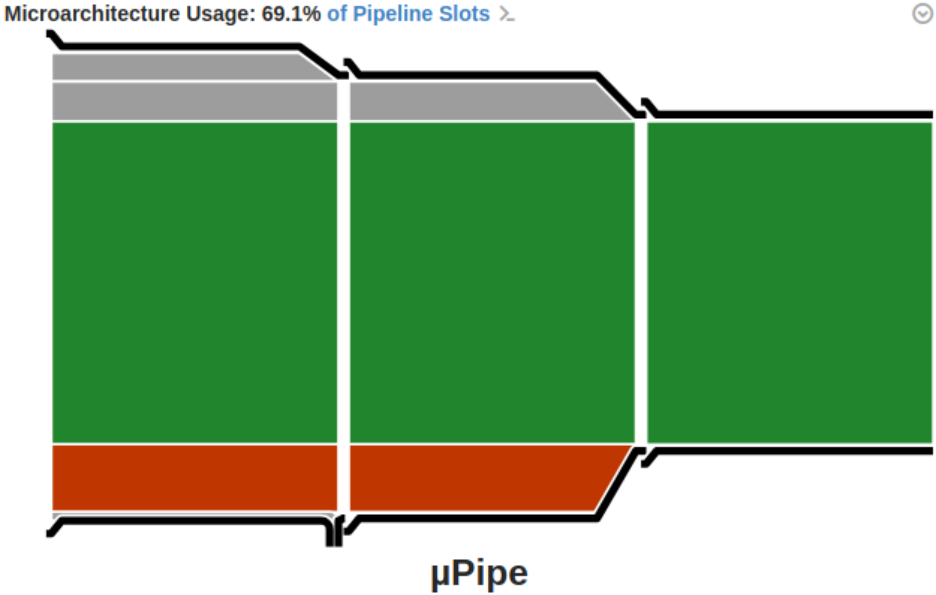
It seems like `shlx` performs differently depending on how the shift count register is initialized. If you

CPU Load Insights

CPU Load Insights



My GEMM



OpenBlas

CPU Load Insights

```
278,670,430,786      instructions              #    2.31  insn per cycle           (69.21%)
120,474,604,222      cycles                    #    2.761 GHz                      (69.21%)
 15,421,445,969      branches                  #  353.467 M/sec                    (69.16%)
     30,942,171      branch-misses             #    0.20% of all branches        (69.17%)
 81,268,780,802      L1-dcache-loads           #    1.863 G/sec                    (69.19%)
  6,950,039,094      L1-dcache-load-misses     #    8.55% of all L1-dcache accesses  (69.22%)
    945,509,752      LLC-loads                 #   21.672 M/sec                    (69.26%)
     62,717,697      LLC-load-misses           #    6.63% of all LL-cache accesses  (69.27%)
```

My GEMM

```
268,093,386,717      instructions              #    2.53  insn per cycle           (69.24%)
105,758,934,675      cycles                    #    2.762 GHz                      (69.25%)
  5,162,809,742      branches                  #  134.846 M/sec                    (69.24%)
      9,843,236      branch-misses             #    0.19% of all branches        (69.23%)
102,783,357,004      L1-dcache-loads           #    2.685 G/sec                    (69.22%)
  9,755,373,829      L1-dcache-load-misses     #    9.49% of all L1-dcache accesses  (69.22%)
    235,889,330      LLC-loads                 #    6.161 M/sec                    (69.22%)
     35,233,587      LLC-load-misses           #   14.94% of all LL-cache accesses  (69.23%)
```

OpenBlas

CPU Load Insights

```
278,670,430,786      instructions              #    2.31  insn per cycle           (69.21%)
120,474,604,222      cycles                    #    2.761 GHz                      (69.21%)
 15,421,445,969      branches                  #  353.467 M/sec                    (69.16%)
     30,942,171      branch-misses             #    0.20% of all branches        (69.17%)
 81,268,780,802      L1-dcache-loads           #    1.863 G/sec                    (69.19%)
  6,950,039,094      L1-dcache-load-misses     #    8.55% of all L1-dcache accesses  (69.22%)
    945,509,752      LLC-loads                 #   21.672 M/sec                    (69.26%)
     62,717,697      LLC-load-misses           #    6.63% of all LL-cache accesses  (69.27%)
```

My GEMM

```
268,093,386,717      instructions              #    2.53  insn per cycle           (69.24%)
105,758,934,675      cycles                    #    2.762 GHz                      (69.25%)
  5,162,809,742      branches                  #  134.846 M/sec                    (69.24%)
      9,843,236      branch-misses             #    0.19% of all branches        (69.23%)
102,783,357,004      L1-dcache-loads           #    2.685 G/sec                    (69.22%)
  9,755,373,829      L1-dcache-load-misses     #    9.49% of all L1-dcache accesses  (69.22%)
    235,889,330      LLC-loads                 #    6.161 M/sec                    (69.22%)
     35,233,587      LLC-load-misses           #   14.94% of all LL-cache accesses  (69.23%)
```

OpenBlas

CPU Load Insights

```
278,670,430,786      instructions              #    2.31  insn per cycle           (69.21%)
120,474,604,222      cycles                    #    2.761 GHz                      (69.21%)
 15,421,445,969      branches                  #  353.467 M/sec                    (69.16%)
     30,942,171      branch-misses             #    0.20% of all branches        (69.17%)
 81,268,780,802      L1-dcache-loads           #    1.863 G/sec                    (69.19%)
  6,950,039,094      L1-dcache-load-misses     #    8.55% of all L1-dcache accesses  (69.22%)
    945,509,752      LLC-loads                 #   21.672 M/sec                    (69.26%)
     62,717,697      LLC-load-misses           #    6.63% of all LL-cache accesses  (69.27%)
```

My GEMM

```
268,093,386,717      instructions              #    2.53  insn per cycle           (69.24%)
105,758,934,675      cycles                    #    2.762 GHz                      (69.25%)
  5,162,809,742      branches                  #  134.846 M/sec                    (69.24%)
      9,843,236      branch-misses             #    0.19% of all branches        (69.23%)
102,783,357,004      L1-dcache-loads           #    2.685 G/sec                    (69.22%)
  9,755,373,829      L1-dcache-load-misses     #    9.49% of all L1-dcache accesses  (69.22%)
    235,889,330      LLC-loads                 #    6.161 M/sec                    (69.22%)
     35,233,587      LLC-load-misses           #   14.94% of all LL-cache accesses  (69.23%)
```

OpenBlas

What numbers tell us?

CPU Load Insights

```
 6.84 3a90: ┌─→vmovupd      (%r15), %ymm12
 2.95       │  vmovupd      0x20(%r15), %ymm13
 7.01       │  vmovupd      0x40(%r15), %ymm14
 2.66       │  vbroadcastsd (%r8,%rbx), %ymm15
 5.97       │  vfmadd231pd  %ymm12, %ymm15, %ymm9
 2.26       │  vfmadd231pd  %ymm13, %ymm15, %ymm11
 6.40       │  vfmadd231pd  %ymm15, %ymm14, %ymm10
 1.44       │  vbroadcastsd 0x8(%r8,%rbx), %ymm15
 5.48       │  vfmadd231pd  %ymm12, %ymm15, %ymm8
 1.20       │  vfmadd231pd  %ymm13, %ymm15, %ymm7
 5.98       │  vfmadd231pd  %ymm15, %ymm14, %ymm6
 1.42       │  vbroadcastsd 0x10(%r8,%rbx), %ymm15
 4.57       │  vfmadd231pd  %ymm12, %ymm15, %ymm3
 1.11       │  vfmadd231pd  %ymm13, %ymm15, %ymm4
 6.37       │  vfmadd231pd  %ymm15, %ymm14, %ymm5
 1.29       │  vbroadcastsd 0x18(%r8,%rbx), %ymm15
 6.74       │  vfmadd231pd  %ymm12, %ymm15, %ymm2
 1.86       │  vfmadd231pd  %ymm13, %ymm15, %ymm1
 5.59       │  vfmadd231pd  %ymm14, %ymm15, %ymm0
 1.05       │  addq         $0x20,%rbx
 4.69       │  addq         $0x60,%r15
 0.01       │  cmpl         $0xb40,%ebx
 1.81       └─ jne          3a90
```

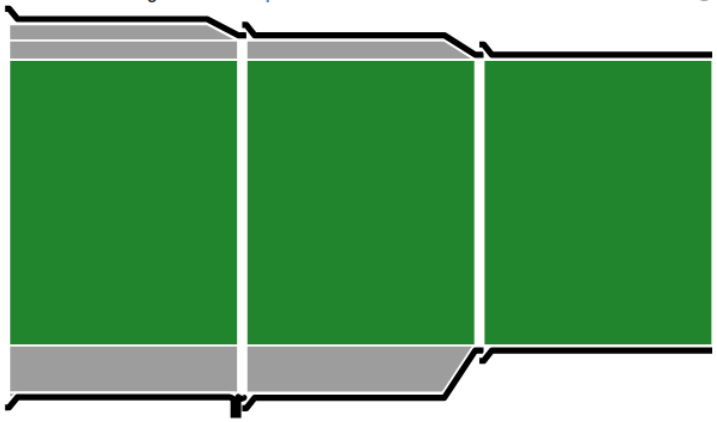
std::simd

```
 0.02       vbroadcastsd 0xb00(%rbx), %ymm8
 0.06       vmovupd      0x60(%rsp), %ymm9
 0.01       vfmadd231pd  %ymm9, %ymm8, %ymm7
 0.05       vmovupd      0xd0(%rsp), %ymm10
            vfmadd231pd  %ymm10, %ymm8, %ymm6
 0.07       vmovupd      0xb0(%rsp), %ymm11
            vfmadd231pd  %ymm8, %ymm11, %ymm5
 0.10       vbroadcastsd 0xb08(%rbx), %ymm8
 0.01       vfmadd231pd  %ymm9, %ymm8, %ymm4
 0.08       vfmadd231pd  %ymm10, %ymm8, %ymm3
            vfmadd231pd  %ymm8, %ymm11, %ymm2
 0.06       vbroadcastsd 0xb10(%rbx), %ymm8
 0.01       vmovups      %ymm8, 0x110(%rsp)
 0.05       vbroadcastsd 0xb20(%rbx), %ymm9
 0.01       vmovupd      0xf0(%rsp), %ymm8
 0.05       vfmadd231pd  %ymm8, %ymm9, %ymm7
            vmovupd      0x270(%rsp), %ymm10
 0.07       vfmadd231pd  %ymm10, %ymm9, %ymm6
 0.02       vmovupd      0x130(%rsp), %ymm11
 0.10       vfmadd231pd  %ymm9, %ymm11, %ymm5
 0.01       vbroadcastsd 0xb18(%rbx), %ymm9
 1.40       vaddpd       (%rdi,%r14,8), %ymm7, %ymm7
 0.16       vmovapd      %ymm7, (%rdi,%r14,8)
```

std::simd + k loop unrolling

CPU Load Insights

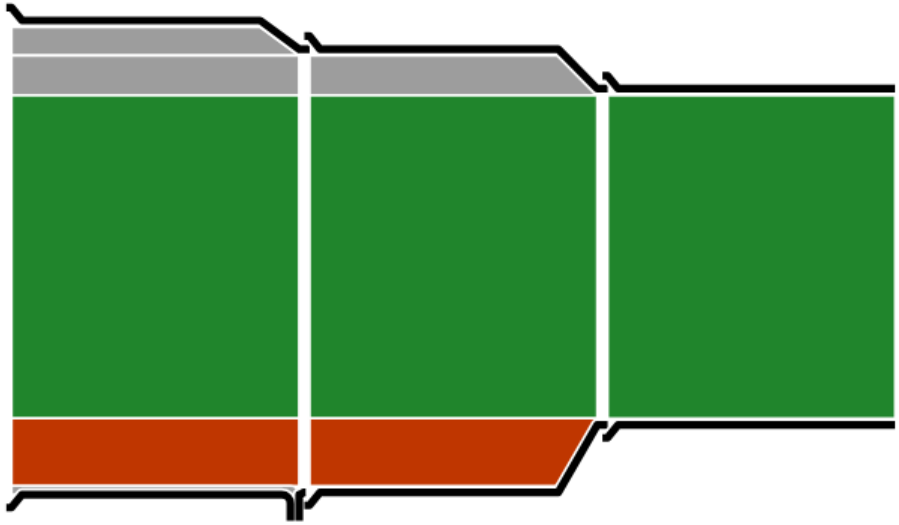
Microarchitecture Usage: 76.5% of Pipeline Slots



µPipe

Retiring:	76.5%	of Pipeline Slots
Light Operations:	80.3%	of Pipeline Slots
FP Arithmetic:	0.0%	of uOps
Memory Operations:	35.5%	of Pipeline Slots
Fused Instructions:	9.7%	of Pipeline Slots
Non Fused Branches:	0.0%	of Pipeline Slots
Nop Instructions:	0.7%	of Pipeline Slots
Other:	34.6%	of Pipeline Slots
Heavy Operations:	0.0%	of Pipeline Slots
Front-End Bound:	4.4%	of Pipeline Slots
Bad Speculation:	1.2%	of Pipeline Slots
Back-End Bound:	17.9%	of Pipeline Slots

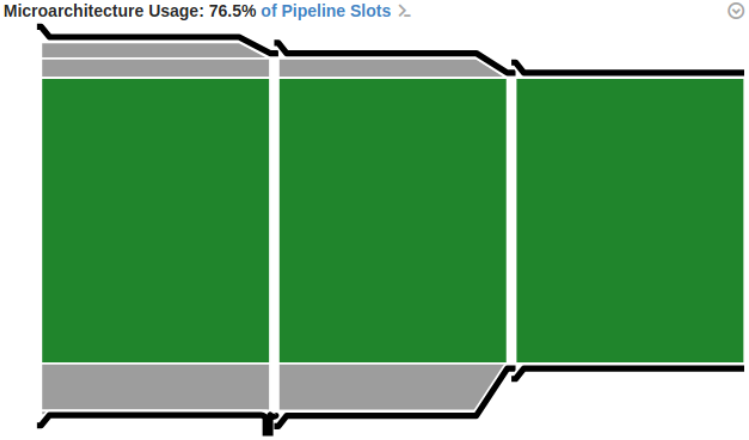
Microarchitecture Usage: 69.1% of Pipeline Slots



µPipe

Retiring:	69.1%	of Pipeline Slots
Front-End Bound:	6.1%	of Pipeline Slots
Bad Speculation:	1.9%	of Pipeline Slots
Back-End Bound:	22.9%	of Pipeline Slots
Memory Bound:	8.4%	of Pipeline Slots
Core Bound:	14.4%	of Pipeline Slots

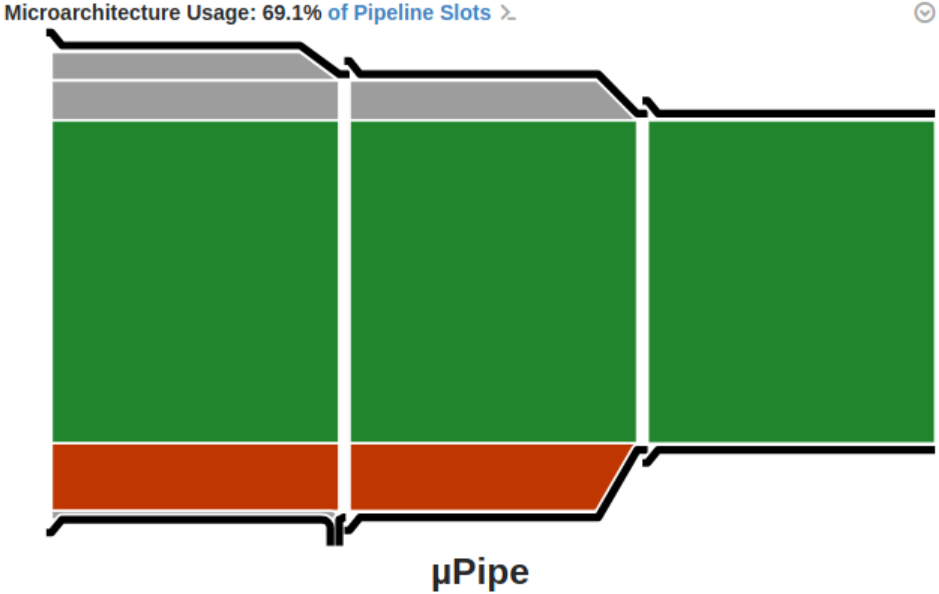
CPU Load Insights



µPipe

Retiring:	76.5%	of Pipeline Slots
Light Operations:	80.3%	of Pipeline Slots
FP Arithmetic:	0.0%	of uOps
Memory Operations:	35.5%	of Pipeline Slots
Fused Instructions:	9.7%	of Pipeline Slots
Non Fused Branches:	0.0%	of Pipeline Slots
Nop Instructions:	0.7%	of Pipeline Slots
Other:	34.6%	of Pipeline Slots
Heavy Operations:	0.0%	of Pipeline Slots
Front-End Bound:	4.4%	of Pipeline Slots
Bad Speculation:	1.2%	of Pipeline Slots
Back-End Bound:	17.9%	of Pipeline Slots

Simd(128-bit)



µPipe

Retiring:	69.1%	of Pipeline Slots
Front-End Bound:	6.1%	of Pipeline Slots
Bad Speculation:	1.9%	of Pipeline Slots
Back-End Bound:	22.9%	of Pipeline Slots
Memory Bound:	8.4%	of Pipeline Slots
Core Bound:	14.4%	of Pipeline Slots

OpenBlas

CPU Load Insights

```
278,670,430,786      instructions              #    2.31  insn per cycle           (69.21%)
120,474,604,222      cycles                    #    2.761 GHz                      (69.21%)
 15,421,445,969      branches                  #  353.467 M/sec                    (69.16%)
     30,942,171      branch-misses             #    0.20% of all branches        (69.17%)
 81,268,780,802      L1-dcache-loads           #    1.863 G/sec                    (69.19%)
  6,950,039,094      L1-dcache-load-misses     #    8.55% of all L1-dcache accesses  (69.22%)
    945,509,752      LLC-loads                 #   21.672 M/sec                    (69.26%)
     62,717,697      LLC-load-misses           #    6.63% of all LL-cache accesses  (69.27%)
```

My GEMM

```
268,093,386,717      instructions              #    2.53  insn per cycle           (69.24%)
105,758,934,675      cycles                    #    2.762 GHz                      (69.25%)
  5,162,809,742      branches                  #  134.846 M/sec                    (69.24%)
      9,843,236      branch-misses             #    0.19% of all branches        (69.23%)
102,783,357,004      L1-dcache-loads           #    2.685 G/sec                    (69.22%)
  9,755,373,829      L1-dcache-load-misses     #    9.49% of all L1-dcache accesses  (69.22%)
    235,889,330      LLC-loads                 #    6.161 M/sec                    (69.22%)
     35,233,587      LLC-load-misses           #   14.94% of all LL-cache accesses  (69.23%)
```

OpenBlas

How effective is C++ for matrix multiplication?

Assembly: Where the FLOPs Really Happen

All kernels are written in assembly

Assembly: Where the FLOPs Really Happen

All kernels are written in assembly

- **You don’t want to struggle against the compiler.**

Assembly: Where the FLOPs Really Happen

All kernels are written in assembly

- **You don’t want to struggle against the compiler.**
- **You don’t want to depend on specific compilers or versions.**

Assembly: Where the FLOPs Really Happen

All kernels are written in assembly

- **You don’t want to struggle against the compiler.**
- **You don’t want to depend on specific compilers or versions.**
- **You want to be able to contribute fixes over an adequate period.**

Assembly: Where the FLOPs Really Happen

All kernels are written in assembly

- **You don’t want to struggle against the compiler.**
- **You don’t want to depend on specific compilers or versions.**
- **You want to be able to contribute fixes over an adequate period.**

The performance of various implementations varies by up to 40% between Clang and GCC.
(NOT applicable for the final version)

But

C++: May the FLOPs Be With You

- **Mature ecosystem:**
  - Kokkos, BLAS, CUDA, SYCL, etc.

C++: May the FLOPs Be With You

- **Mature ecosystem:**
  - Kokkos, BLAS, CUDA, SYCL, etc.
- **Zero-cost abstractions with templates and constexpr**
  - Write generic, readable code that compiles down to highly optimized kernels.

C++: May the FLOPs Be With You

- **Mature ecosystem:**
  - Kokkos, BLAS, CUDA, SYCL, etc.
- **Zero-cost abstractions with templates and constexpr**
  - Write generic, readable code that compiles down to highly optimized kernels.
- **Chip-Ready**
  - Preferred for new architectures like RISC-V — ideal for hardware-aware tuning.

C++: May the FLOPs Be With You

- **Mature ecosystem:**
  - Kokkos, BLAS, CUDA, SYCL, etc.
- **Zero-cost abstractions with templates and constexpr**
  - Write generic, readable code that compiles down to highly optimized kernels.
- **Chip-Ready**
  - Preferred for new architectures like RISC-V — ideal for hardware-aware tuning.
- **More Power on the Horizon**
  - std::mdspan std::simd reflection std::linalg std::executors

C++: May the FLOPs Be With You

- **Mature ecosystem:**
  - Kokkos, BLAS, CUDA, SYCL, etc.
- **Zero-cost abstractions with templates and constexpr**
  - Write generic, readable code that compiles down to highly optimized kernels.
- **Chip-Ready**
  - Preferred for new architectures like RISC-V — ideal for hardware-aware tuning.
- **More Power on the Horizon**
  - std::mdspan std::simd reflection std::linalg std::executors
- **Powered by a Strong and Active Community**

Alternatives

Alternatives

MLIR

MLIR (**Multi-Level Intermediate Representation**) is a unifying software framework for compiler development.

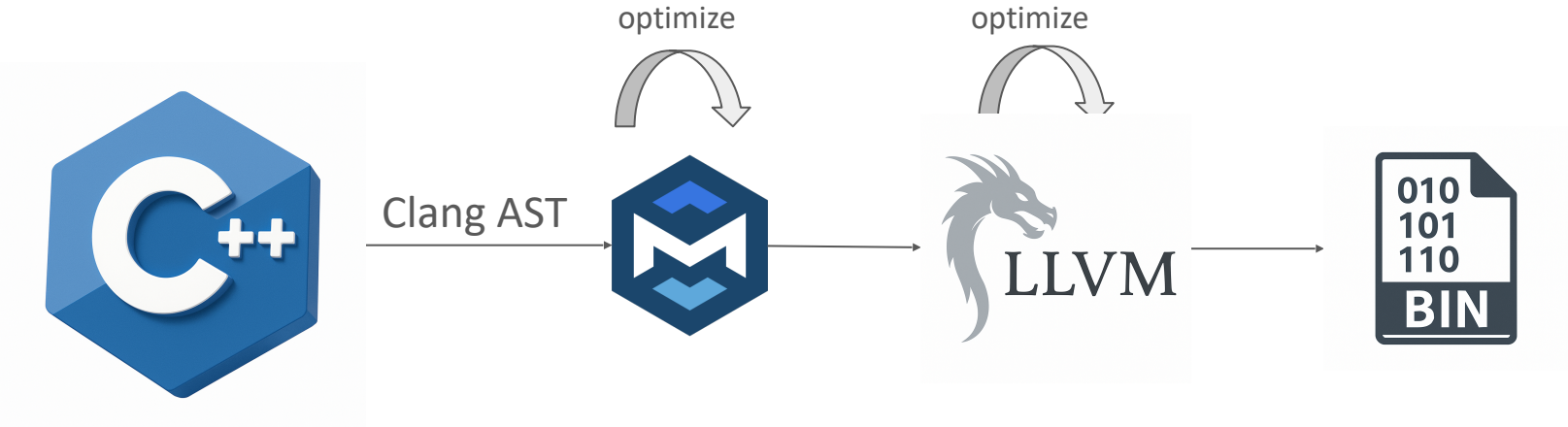
Alternatives

MLIR

```
func.func @matmul(%A: tensor<MxKxf32>, %B: tensor<KxNxf32>) -> tensor<MxNxf32> {
  %C = linalg.matmul ins(%A, %B : tensor<MxKxf32>, tensor<KxNxf32>) -> tensor<MxNxf32>
  return %C : tensor<MxNxf32>
}
```

Alternatives

Polygeist



Alternatives

Mojo

Alternatives

Mojo

```
fn matmul_unrolled[mode: Int](inout C: Matrix, A: Matrix, B: Matrix):
    @parameter
    fn calc_row(m: Int):
        @parameter
        fn calc_tile[tile_x: Int, tile_y: Int](x: Int, y: Int):
            @parameter
            for _k in range(tile_y):
                var k = _k + y
                @parameter
                fn dot[nelts: Int](n: Int):
                    C.store(m, n + x, C.load[nelts](m, n + x) + A[m, k] * B.load[nelts](k, n + x))

                vectorize[
                    dot, nelts, size=tile_x, unroll_factor = tile_x // nelts
                ]()

        tile[calc_tile, tile_n, tile_k](C.cols, B.rows)

    parallelize[calc_row](C.rows, num_workers)
```

Alternatives

Mojo - 70 % of peak performance

Alternatives

Mojo - 70 % of peak performance

LLM - 70 % of peak performance

Future plans

- Complete work on CPU-side matrix multiplication algorithms
- Repeat the same exercise using the GPU.
- Repeat the same exercise using the AI chips.

Thank you!

For more information, contact

Aliaksei Sala

Lead Software Engineer

LinkedIn: https://www.linkedin.com/in/aliaksei-sala/