SIMD Seminar

Yang Supei, Yang Jiekun

CUHK(SZ) Supercomputing Group

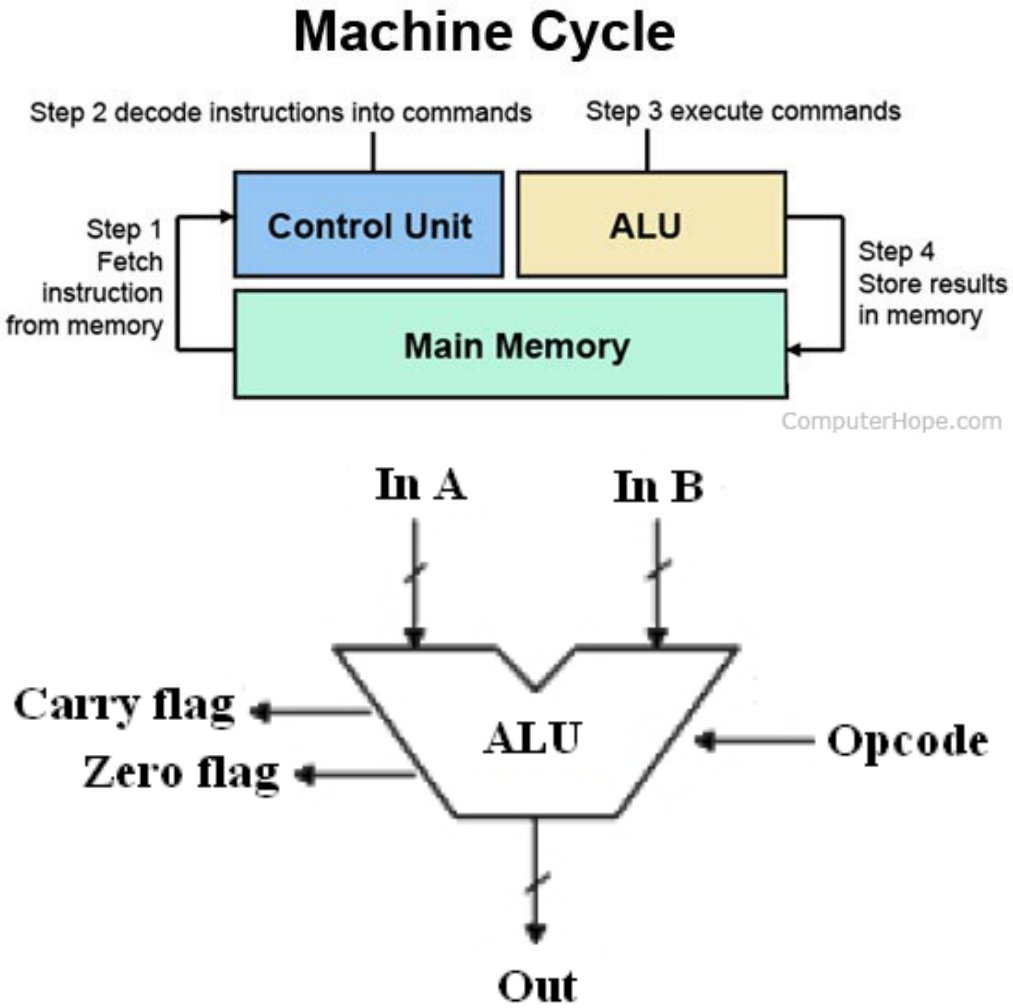
October 18, 2020

Overview

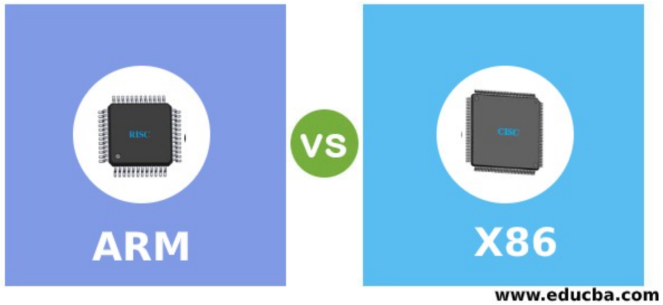
Introduction to SIMD instructions

Test Programs

Introduction to SIMD



Instruction Set



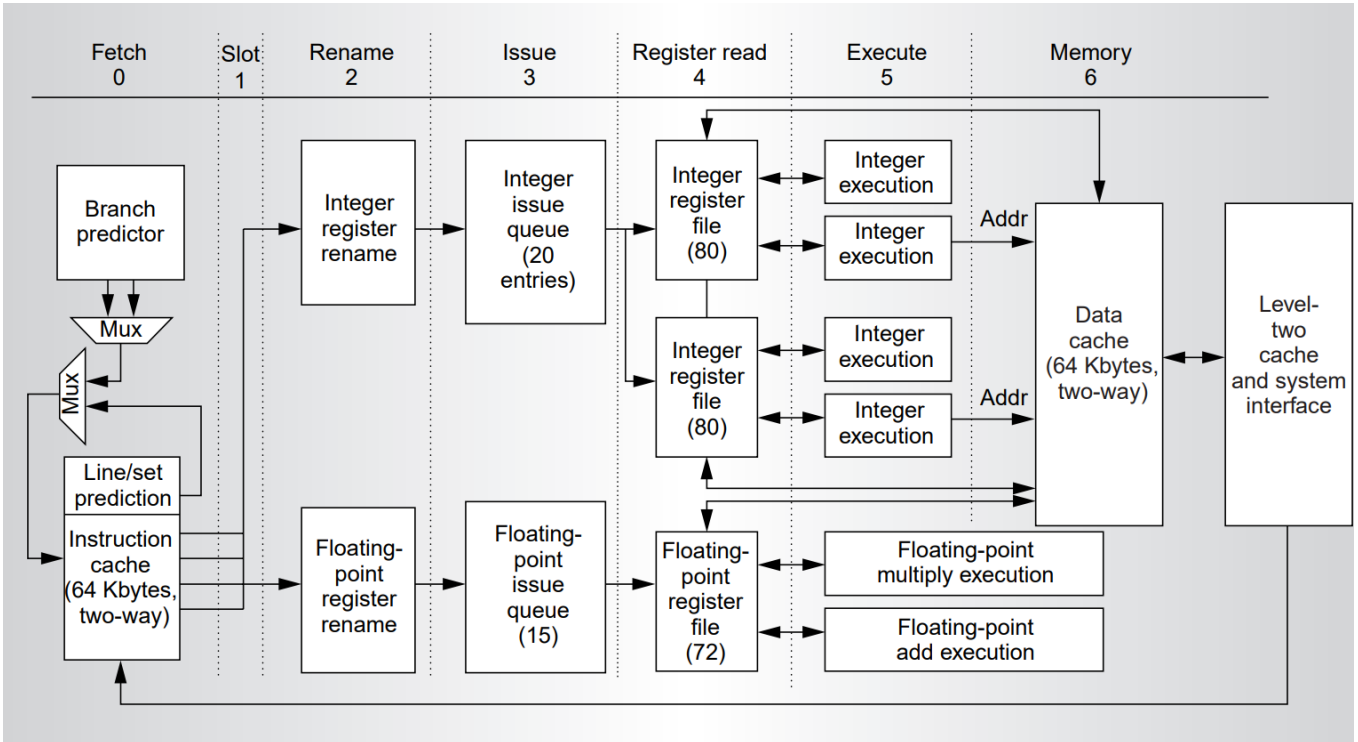
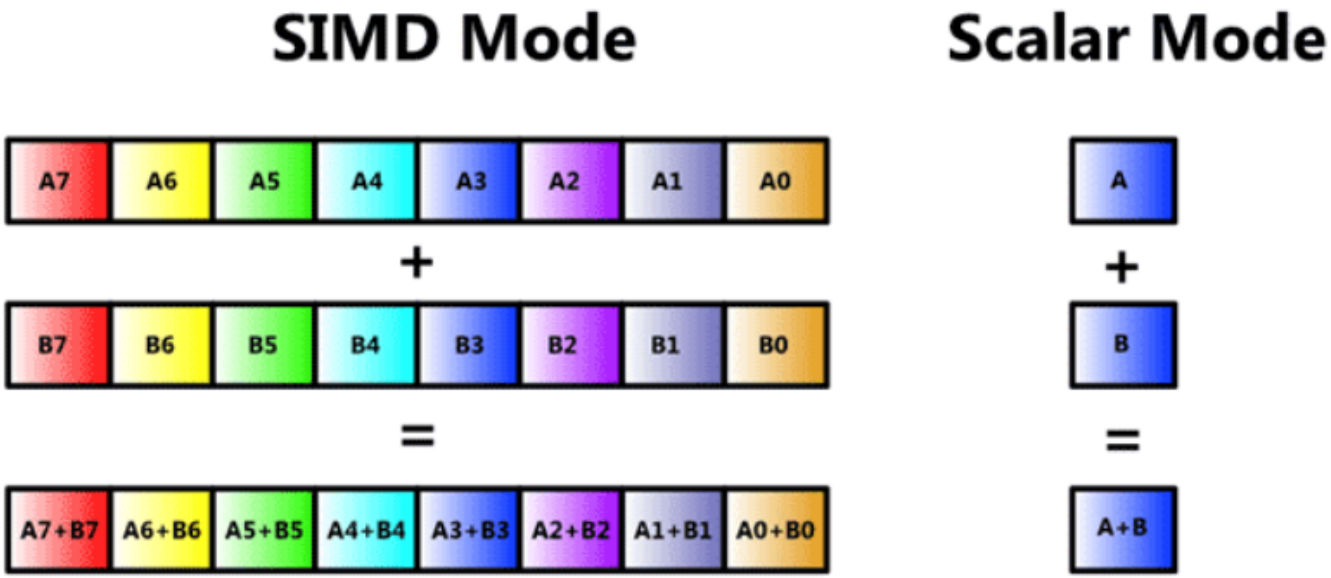


Figure 2. Stages of the Alpha 21264 instruction pipeline.

Introduction to SIMD

SIMD(Single Instruction Multiple Data)



Instruction Set

- MMX(Matrix Math eXtensions):
  MM0-MM7, 64-bit reg, integer only
- SSE(Streaming SIMD Extensions):
  XMM0-XMM7(support x86-64 by adding XMM8-XMM15)
  128-bit reg, floating-point support

Instruction Set (cont.)

- AVX(Advanced Vector Extensions)
  YMM0-YMM15, 256-bit reg
- AVX2, add FMA operation
- AVX-512, 512-bit extensions to AVX2

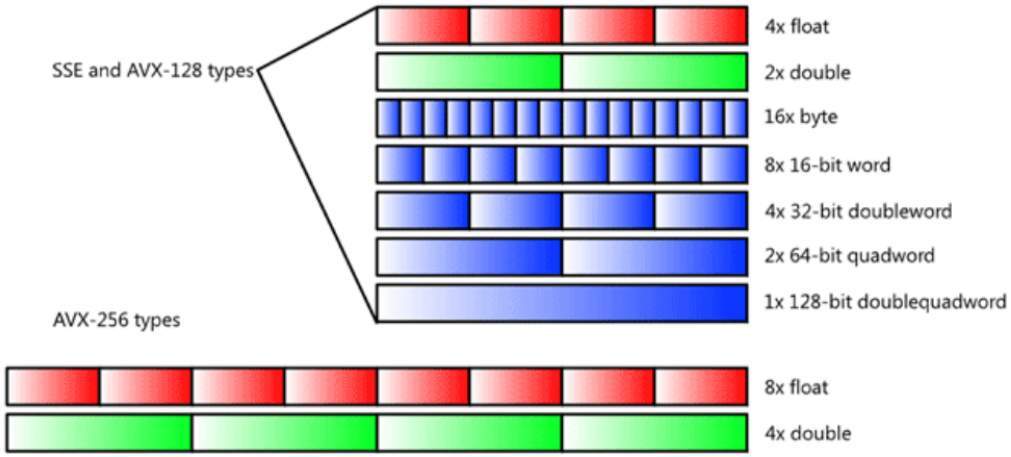


Figure 1: Datatype

Intel Intrinsics

- AVX header: <immintrin.h>
- function definition
  _mm<bit_width>_<name>_<data_type>
  __m256 _mm256_set1_ps (float a),
  Broadcast a single-precision (32-bit) floating-point value a to 8-element vector
  __m256 _mm256_add_ps (__m256 a, __m256 b)
  Add 8 single-precision (32-bit) floating-point elements in a and b.

Intel Intrinsics (cont.)

- (load/store/set)
- Arithmetic
- Comparison
- Logical Operation

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Vector-Scalar Multiplication

```
__m256 scalar = _mm256_set1_ps(c);
for (int i = 0; i < size; i += 8) {
    vec = _mm256_loadu_ps(a + i);
    vec = _mm256_mul_ps(vec, scalar);
    _mm256_storeu_ps(a + i, vec);
}
```

Figure 2: AVX version

```
for (int i = 0; i < size; i++) {
    a[i] = a[i] * c;
}
```

Figure 3: Non-AVX version

Performance

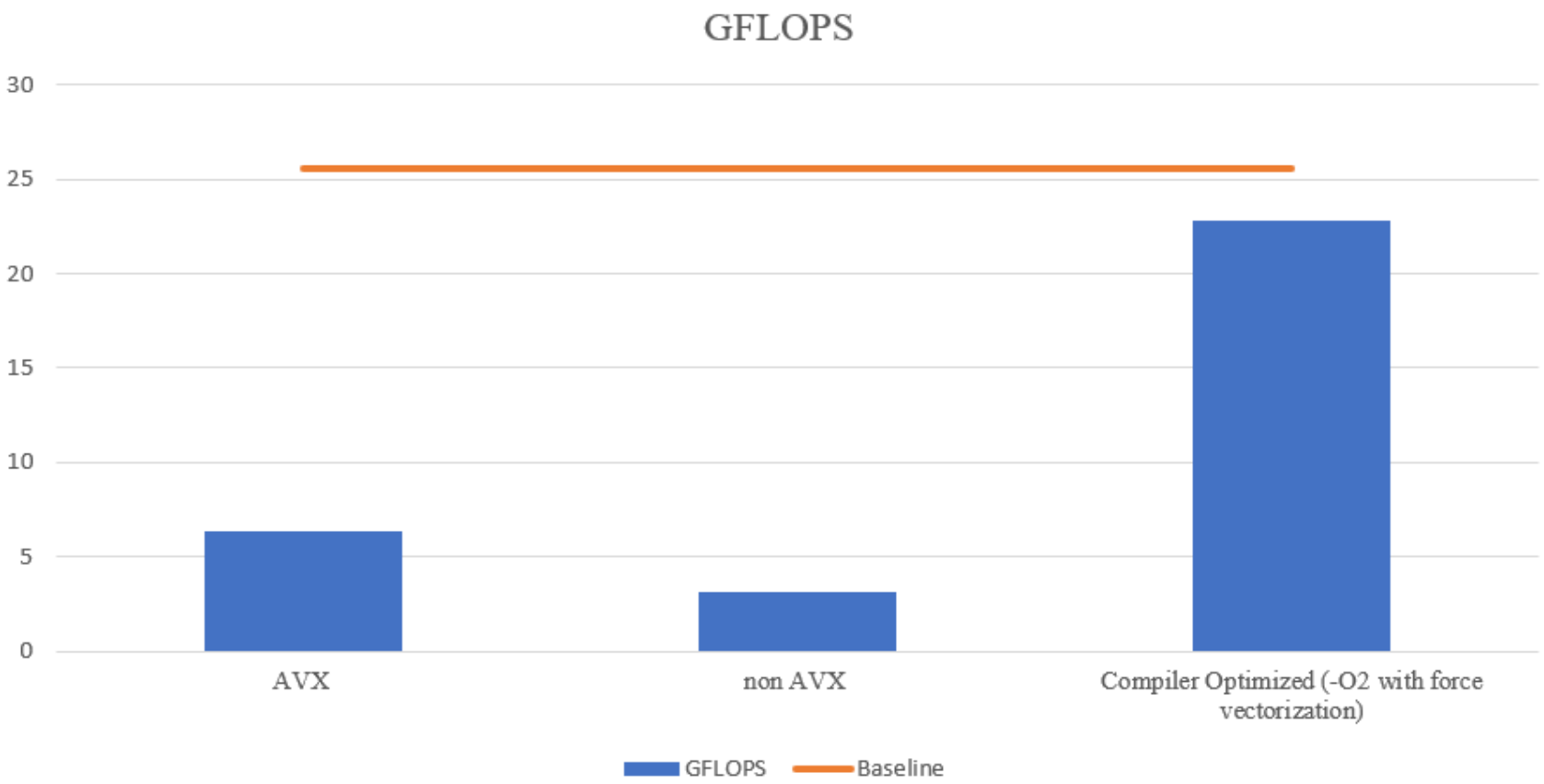


Figure 4: AVX Performance

Further Optimization

```
vmovups    %ymm0, (,%rax,4)
addq       $8, %rax
cmpq       $1000, %rax
jl         ..B1.2          # Pr
```

Original assembly

```
vmovups    %ymm0, (,%rax,4)
vmovups    %ymm0, 32(,%rax,4)
vmovups    %ymm0, 64(,%rax,4)
vmovups    %ymm0, 96(,%rax,4)
addq       $32, %rax
cmpq       $992, %rax
jb         ..B1.2          # Prob 82%
```

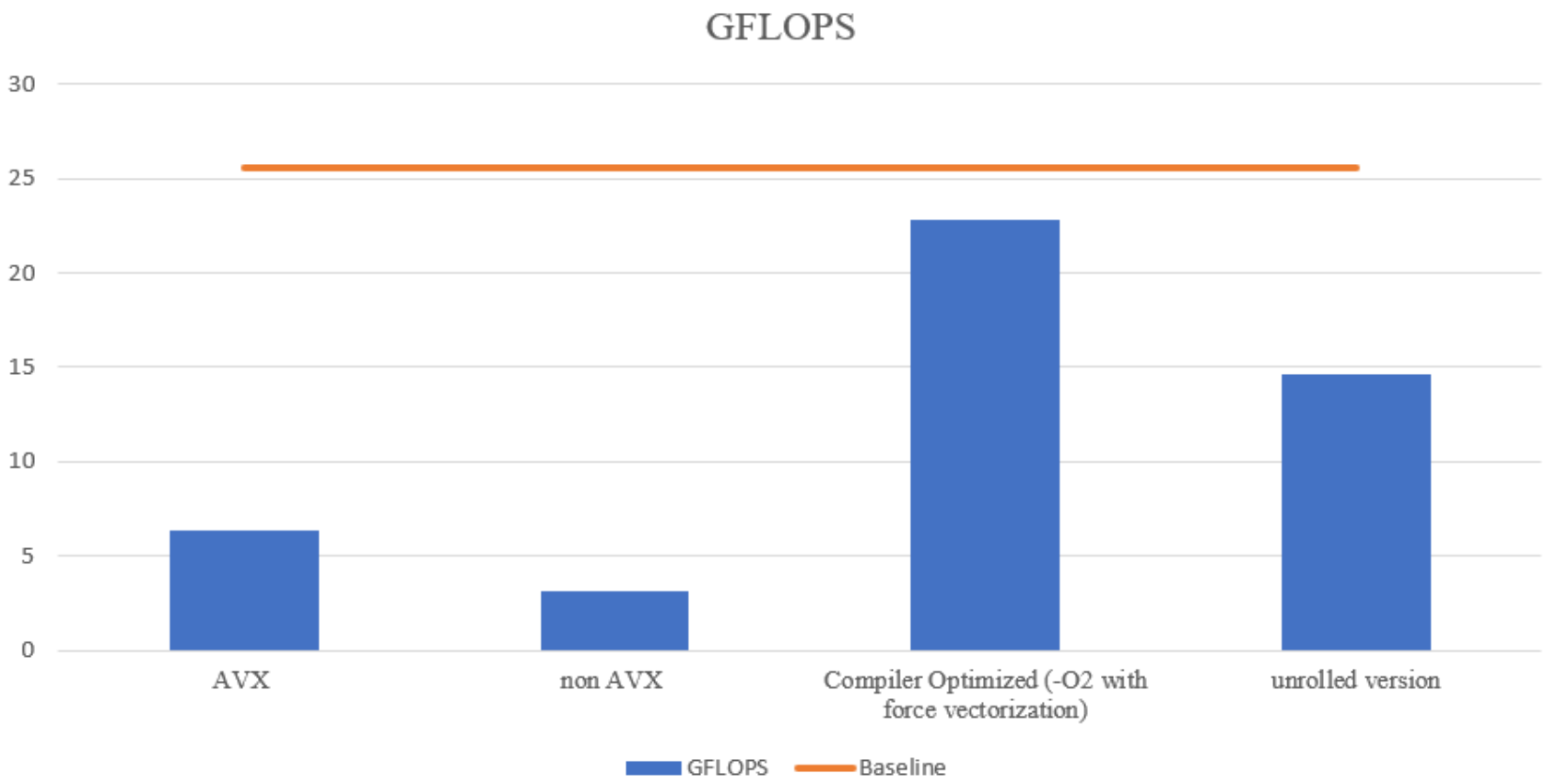
Compiler optimized assembly

```
#pragma unroll(4)
for (int i = 0; i < size; i += 8) {
    vec = _mm256_loadu_ps(a + i);
    vec = _mm256_mul_ps(vec, scalar);
    _mm256_storeu_ps(a + i, vec);
}
```

```
movslq     %eax, %rax
incb       %dl
vmovups    %ymm0, (,%rax,4)
vmovups    %ymm0, 32(,%rax,4)
vmovups    %ymm0, 64(,%rax,4)
vmovups    %ymm0, 96(,%rax,4)
addl       $32, %eax
cmpb       $31, %dl
jb         ..B1.5          # Prob 27%
```

New version assembly

Further Optimization



Vector-Vector Multiplication

```
for(int i=0;i<size;i+=8){
    vec1 = _mm256_loadu_ps(a+i);
    vec2 = _mm256_loadu_ps(b+i);
    sum = _mm256_fmadd_ps(vec1,vec2,sum);
```

Figure 5: AVX version

```
double dot(const float* a, const float* b, int size){
    double product = 0;
    for(int i=0;i<size;i++){
        product+=a[i]*b[i];
    }
    return product;
```

Figure 6: Non-AVX version

Add Reduce

Add all elements in "Sum" up

```
float fhsum(__m256 x){
    // hiQuad = ( x7, x6, x5, x4 )
    const __m128 hiQuad = _mm256_extractf128_ps(x, 1);
    // loQuad = ( x3, x2, x1, x0 )
    const __m128 loQuad = _mm256_castps256_ps128(x);
    // sumQuad = ( x3 + x7, x2 + x6, x1 + x5, x0 + x4 )
    const __m128 sumQuad = _mm_add_ps(loQuad, hiQuad);
    // loDual = ( -, -, x1 + x5, x0 + x4 )
    const __m128 loDual = sumQuad;
    // hiDual = ( -, -, x3 + x7, x2 + x6 )
    const __m128 hiDual = _mm_movehl_ps(sumQuad, sumQuad);
    // sumDual = ( -, -, x1 + x3 + x5 + x7, x0 + x2 + x4 + x6 )
    const __m128 sumDual = _mm_add_ps(loDual, hiDual);
    // lo = ( -, -, -, x0 + x2 + x4 + x6 )
    const __m128 lo = sumDual;
    // hi = ( -, -, -, x1 + x3 + x5 + x7 )
    const __m128 hi = _mm_shuffle_ps(sumDual, sumDual, 0x1);
    // sum = ( -, -, -, x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7 )
    const __m128 sum = _mm_add_ss(lo, hi);
    return _mm_cvtss_f32(sum);
}
```

Figure 7: Add Reduce

Performance

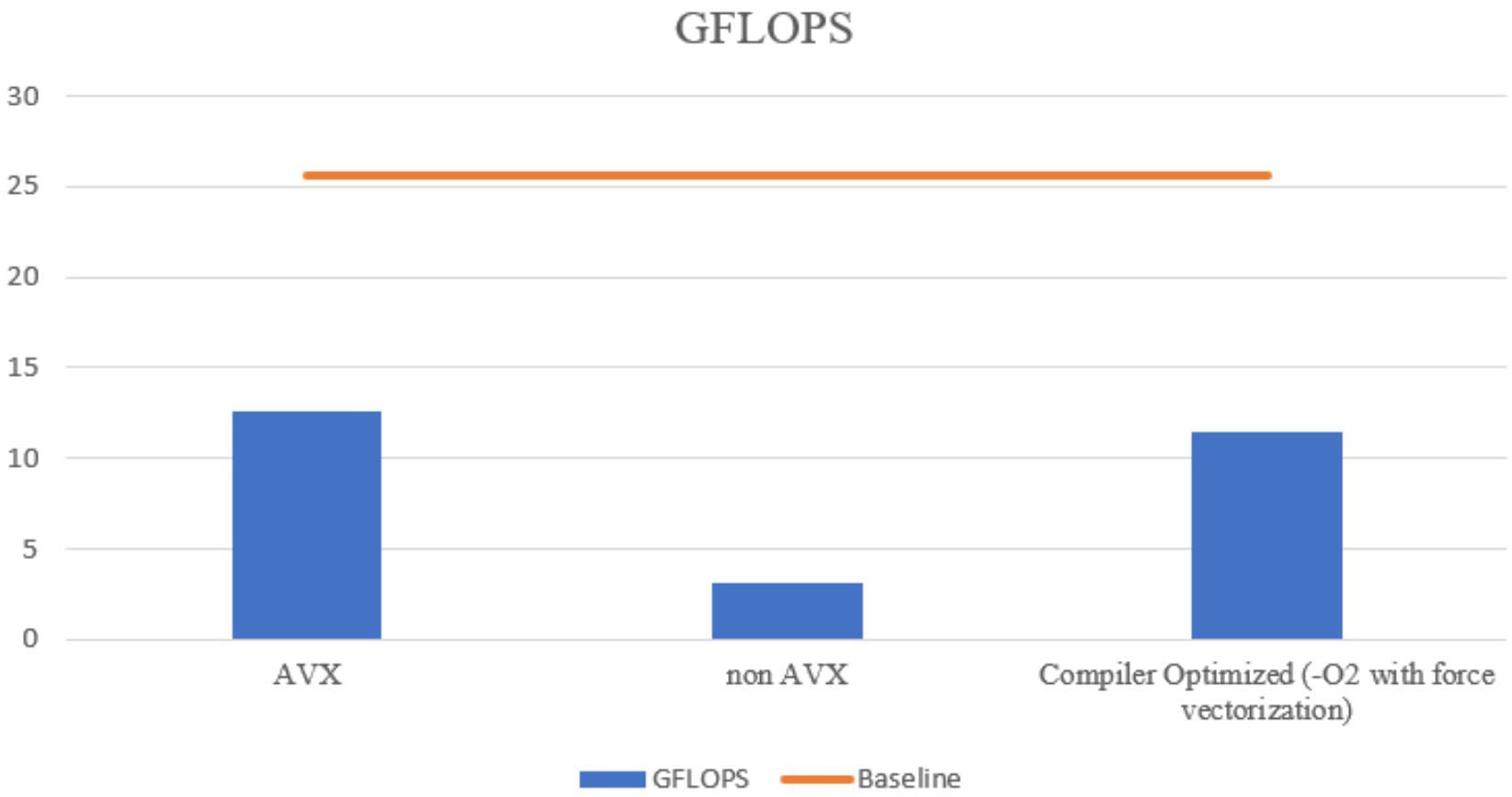


Figure 8: AVX Performance

Vector-Matrix Multiplication

```
float* _avx_mul(float* m, float* v,int size){
    float* product = (float*) malloc(size*sizeof(float));
    __m256 sum = _mm256_setzero_ps();
    for(int i=0;i<size;i++){
        for(int j=0;j<size;j+=8){
            __m256 vec1 = _mm256_loadu_ps(m+j+i*size);
            __m256 vec2 = _mm256_loadu_ps(v+j);
            sum = _mm256_fmadd_ps(vec1,vec2,sum);
        }
            product[i] = fhsum(sum);
    }
    return product;
```

Figure 9: AVX version

```
float* mul(float* m, float* v, int size){
    float* product = (float*) malloc(size*sizeof(float));
    for(int i=0;i<size;i++){
        for(int j=0;j<size;j++){
            product[i] = m[i*size+j]+v[j];
        }
    }
    return product;
```

Figure 10: Non-AVX version

Performance

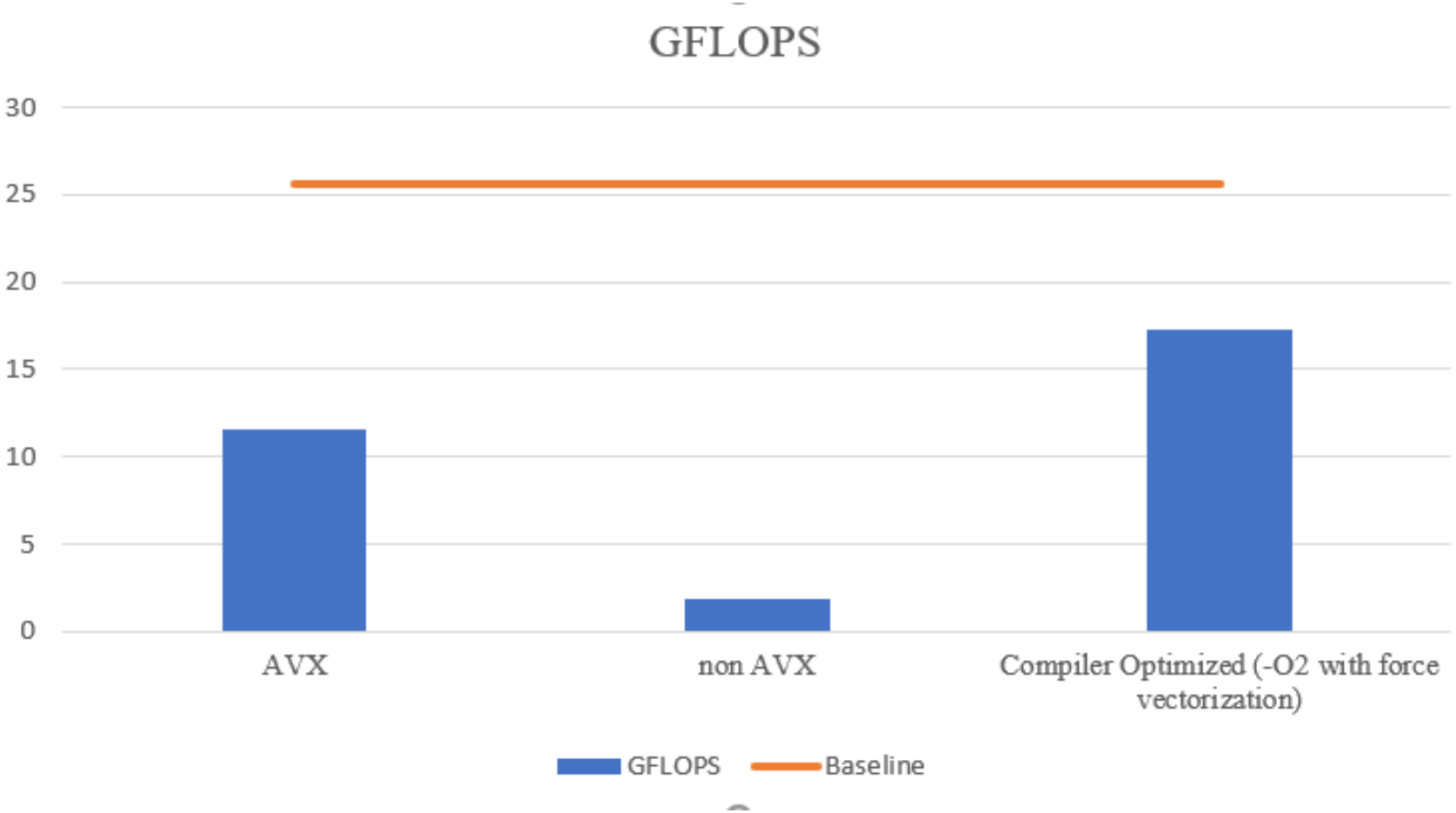


Figure 11: AVX Performance

Multi-Core Processing using MPI

```
MPI_Init(&argc,&argv);

int my_rank,p_num;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p_num);

a = (float*) malloc(MAX_N*sizeof(float));
init_vec(a,MAX_N);
start = MPI_Wtime();
for(int i = 0; i < ITER; i++ ) _avx_mul(a,1.1,MAX_N);

MPI_Barrier(MPI_COMM_WORLD);
if(my_rank == 0){
    time_cost = MPI_Wtime() - start;
    Gflops = MAX_N * ITER * p_num;
    printf("MPI time cost:%.5lf Gflops:%.2lf \n",time_cost,  Gflops / (1000000000*time_cost));
    free(a);
```

Multi-Core Processing using MPI

[Modify Frequency Info]

Mode	Base	Turbo Frequency/Active Cores									
		1	2	3	4	5	6	7	8	9	10
Normal	2,200MHz	3,200MHz	3,200MHz	3,000MHz	3,000MHz	2,900MHz	2,900MHz	2,900MHz	2,900MHz	2,700MHz	2,700MHz
AVX2	1,900MHz	3,000MHz	3,000MHz	2,800MHz	2,800MHz	2,500MHz	2,500MHz	2,500MHz	2,500MHz	2,300MHz	2,300MHz
AVX512	1,200MHz	2,000MHz	2,000MHz	1,800MHz	1,800MHz	1,600MHz	1,600MHz	1,600MHz	1,600MHz	1,500MHz	1,500MHz