بهبود کارایی برنامه های چندرسانه‌ای با استفاده از برنامه نویسی SIMD

حسین امیری

استاد راهنما: دکتر اسد الله شاه بهرامی

Performance Improvement of Multimedia Applications using SIMD Programming

Hossein Amiri

Supervisor: Asadollah Shahbahrami

Overview

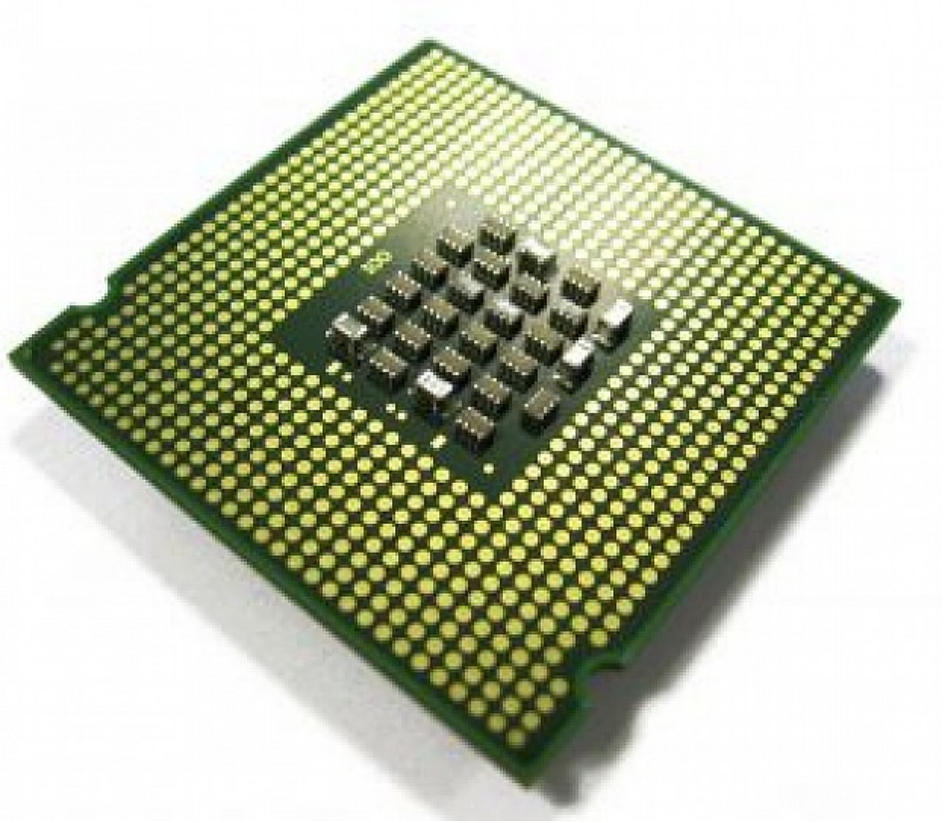
- Introduction
- Multimedia Principles
- SIMD Principles
- Programming Approaches
- Proposed Framework
- Implementations
- Experimental Results
- Conclusions

Introduction(1/3)

- Multimedia content such as video, image, audio, interactive content has been considerably popular

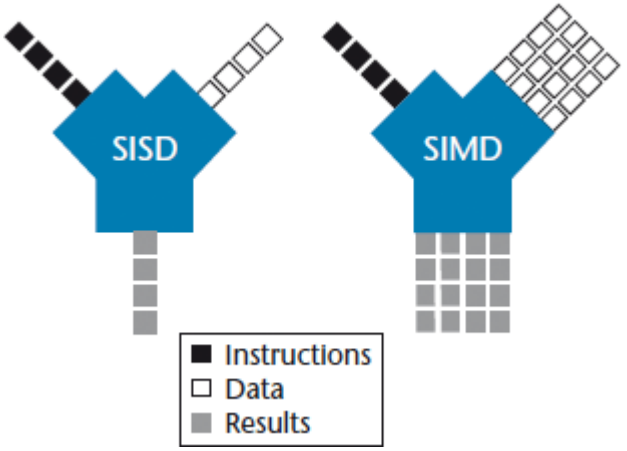
Introduction(2/3)

- Multimedia data is very large
- The larger data production, the more processing power requirement



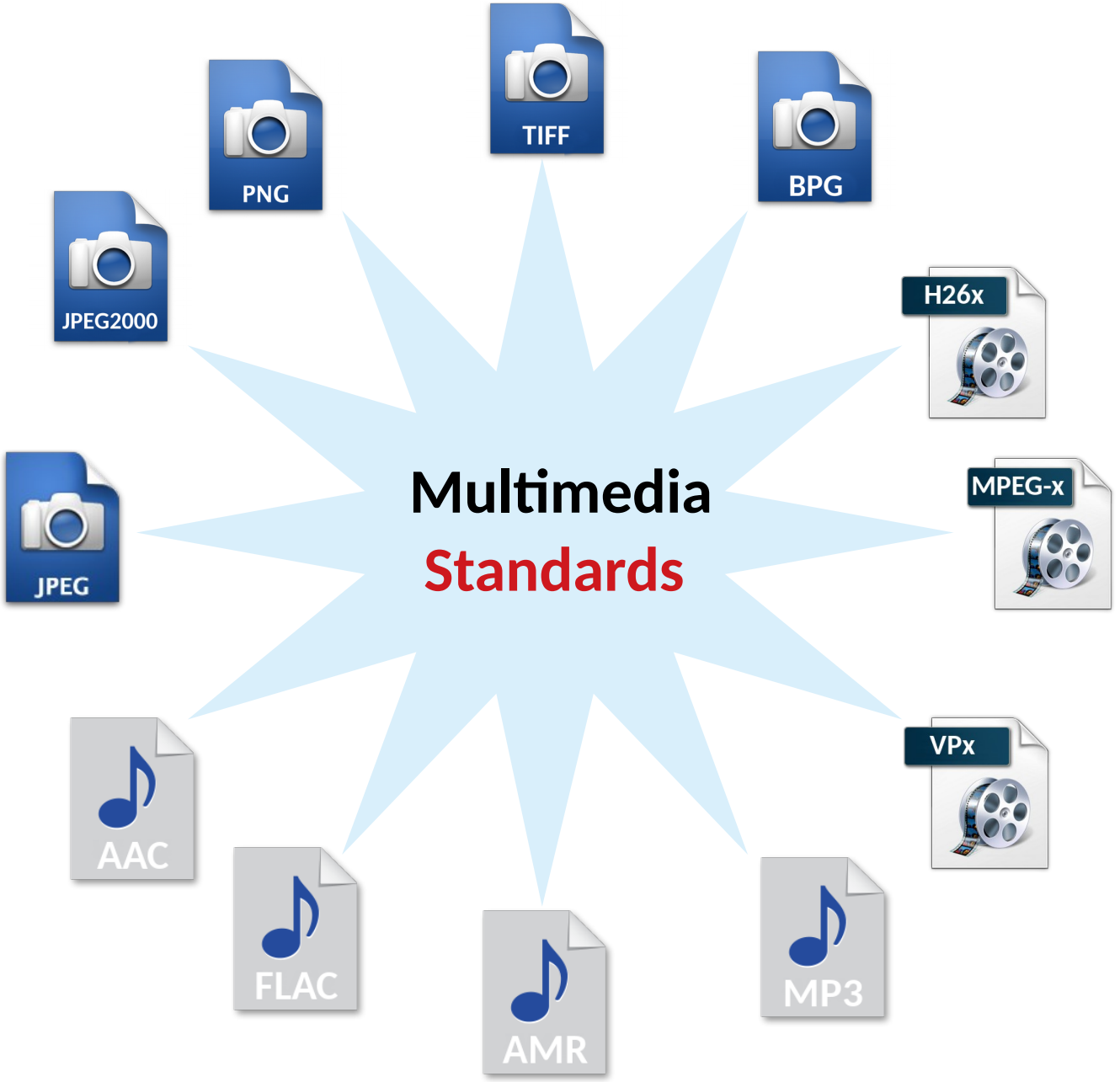
Introduction(3/3)

- Single Instruction and Multiple Data (SIMD)



Multimedia Principles(1/4)

Multimedia Principles(2/4)



Multimedia Principles(3/4)

Kernels:

- Matrix Operation:
  - Image, video and signal processing
- 2-D Convolution:
  - Image filtering such as sharpen, blur, edge detection.
- FIR filter:
  - Audio filtering in AMR standard
- Motion Estimations:
  - Image compression

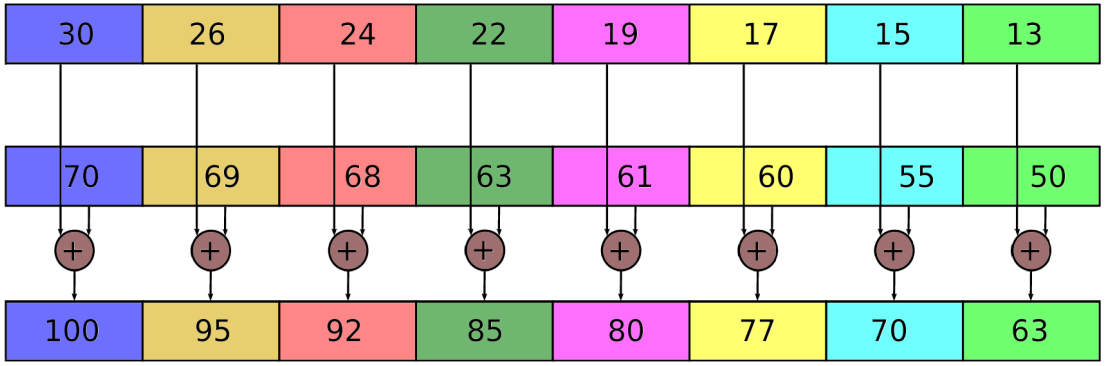
Multimedia Principles(4/4)

Multimedia features:

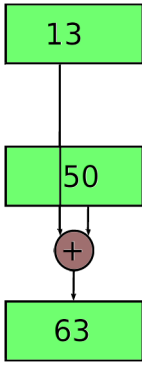
- The same operations on different data item
- Operations are largely independent
- Small data types

SIMD Principles(1/5)

Introduction:



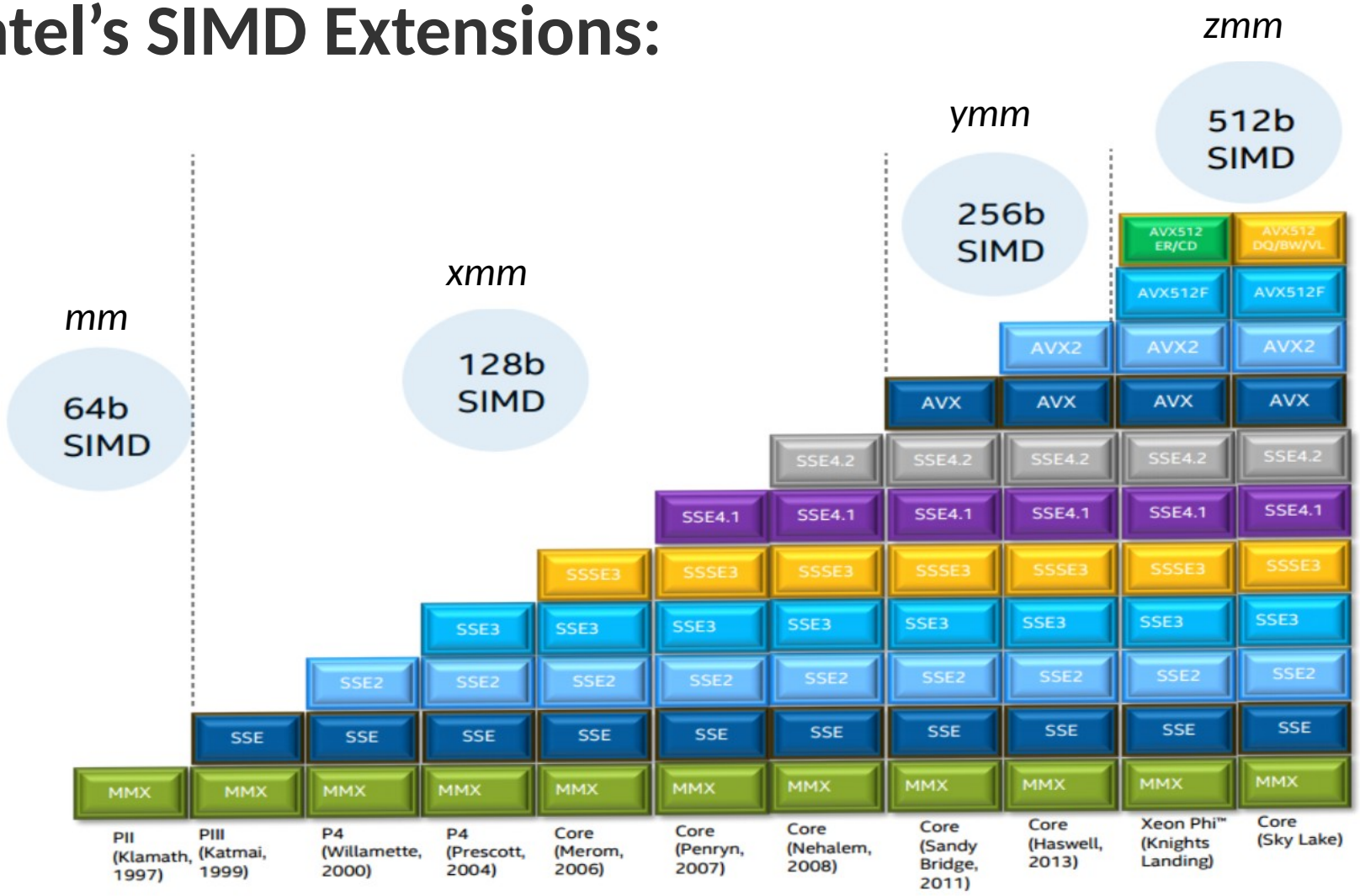
A simple SIMD operation (providing 8-way parallelism)



A simple scalar operation

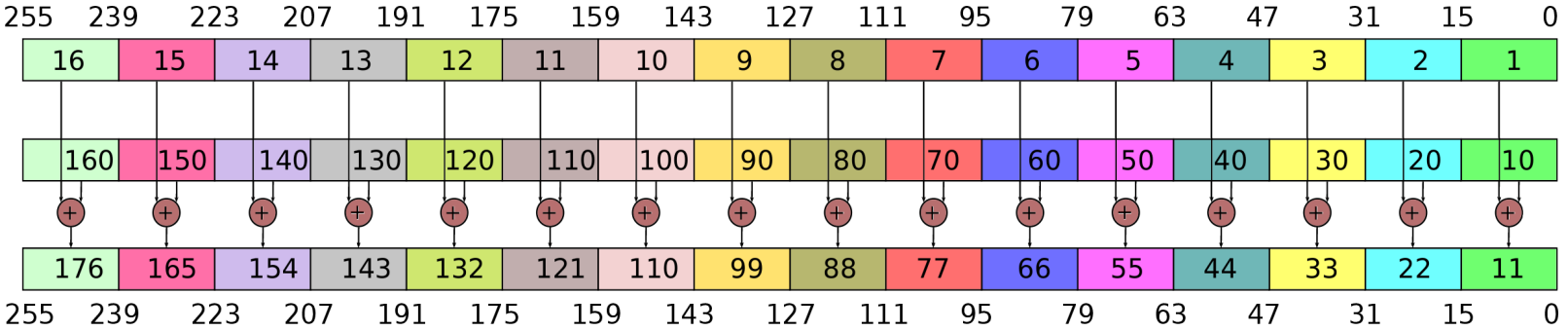
SIMD Principles(2/5)

Intel's SIMD Extensions:



SIMD Principles(3/5)

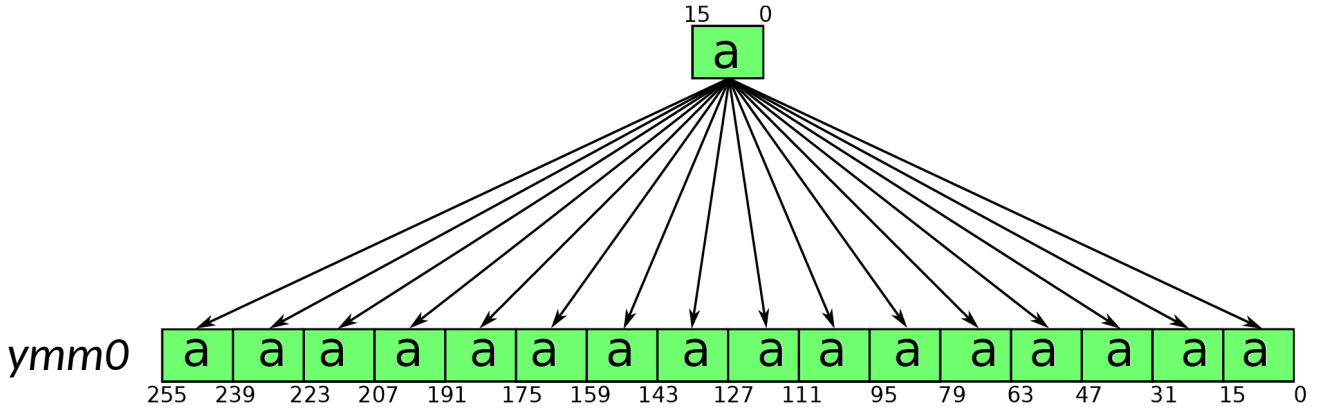
- There are many 16-bit instructions in AVX2 instruction set architecture
- It provides 16-way parallelism (256/16=16)



Using 16 short integers in a 256-bit vector register provides 16-way parallelism

SIMD Principles(4/5)

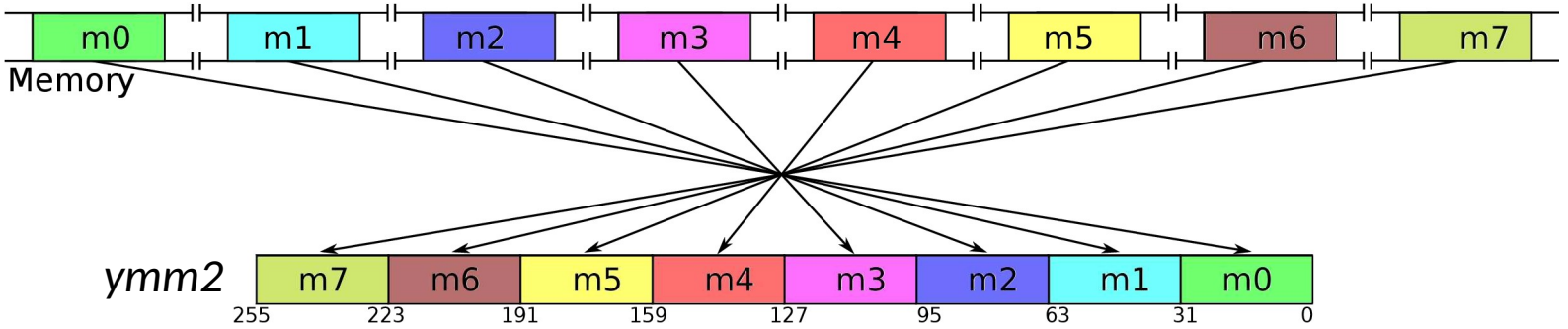
- Broadcast is a new instruction
  - *vpbroadcastw mem(%rip), %ymm0*



Broadcast a short value to all 16 elements of a vector register

SIMD Principles(5/5)

- Gather is a new instruction
  - *vpgatherdd %ymm1, (%rdx, %ymm0, 4), %ymm2*



Gather 8x32-bit values to a vector register

Programming Approaches(1/3)

Intrinsic Programming Model (IPM) :

- Assembly style SIMD codes in a high-level language

```
//Scalar implementation
for(i=0; i<size; i++)
    C[i] = A[i] + B[i];

//Vectorized code using IPM
for(i=0; i<size; i+=8){
    __m256 vec_a = _mm256_load_ps(&A[i]);
    __m256 vec_b = _mm256_load_ps(&B[i]);
    __m256 vec_c = _mm256_add_ps(vec_a, vec_b);
    __mm256_store_ps(&C[i], vec_c);
}
```

Programming Approaches(2/3)

Open Multi-Processing (OpenMP) :

- Compilers must support the OpenMP Application Programming Interface (API)

```
#pragma omp simd
for(i=0; i<size; i++)
    C[i] = A[i] + B[i];
```

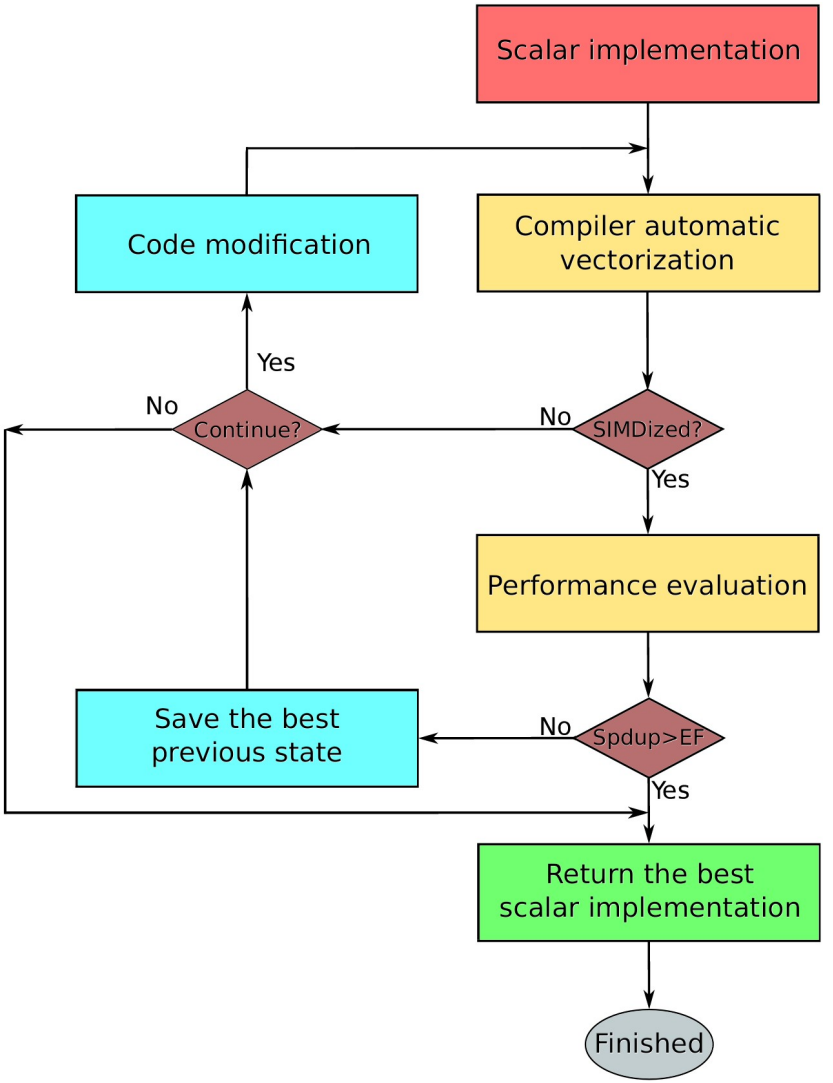
Programming Approaches(3/3)

Compiler's Automatic Vectorization (CAV) :

- Compilers can exploit the SIMD instructions automatically
  - *icc/gcc/clang -O3 code.c*

Proposed Framework

- *Source code is important*
- *Proposed framework improves the compiler's behavior*
- *Current state-of-the-art compilers need a similar approach*



Implementations

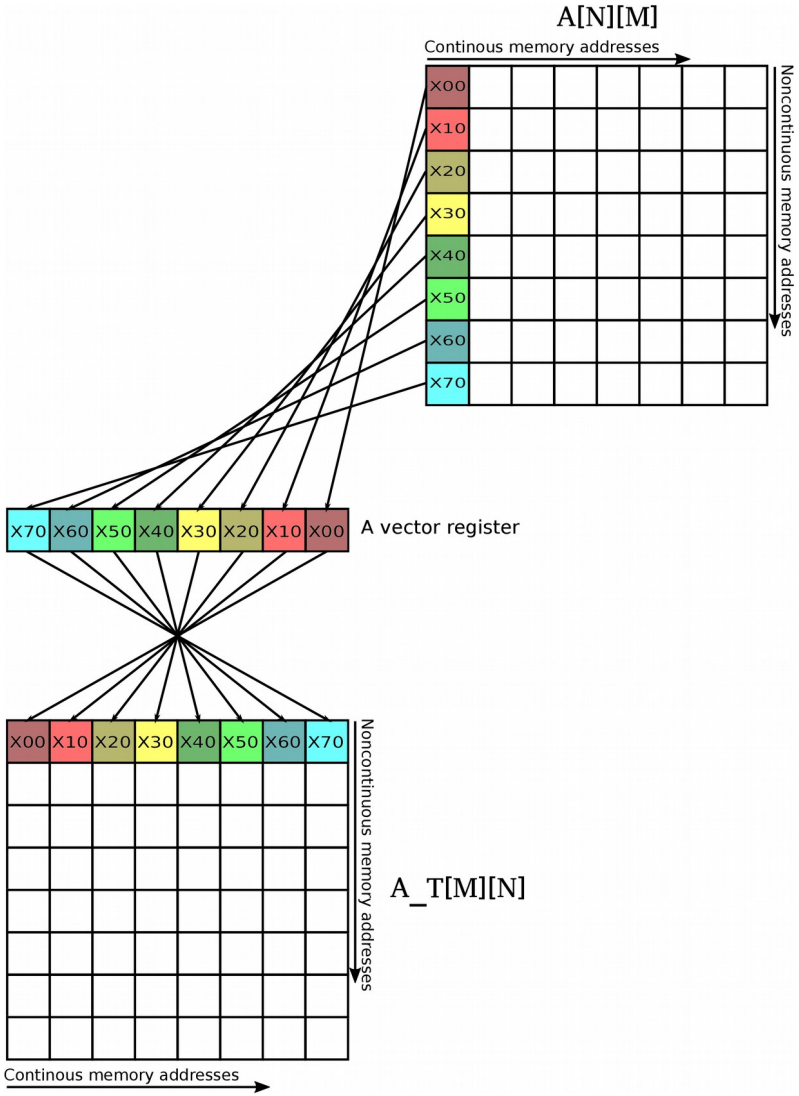
Implemented Kernels using IPM

Matrix-Matrix Addition	(ADD)	جمع ماتریس
Matrix Transposition	(TRA)	ترانهاده ماتریس
Matrix-Matrix Multiplication	(MUL)	ضرب ماتریس
Sum of Absolute Differences	(SAD)	مجموع قدرمطلق تفاضل‌ها
Sum of Squared Differences	(SSD)	مجموع مربع تفاضل‌ها
Finite Impulse Response Filter	(FIR)	فیلتر FIR
2-D Convolution	(CON)	پیچش دوبعدی

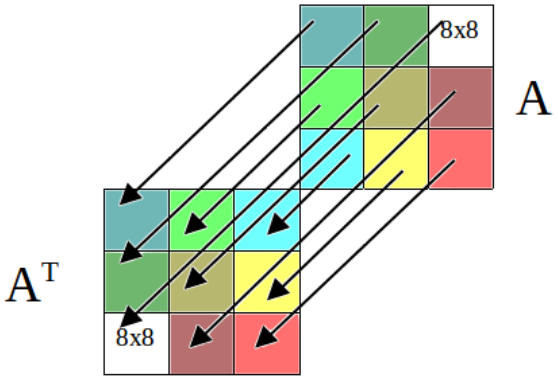
Implementations – TRA (1/2)

```
//Scalar implementation
//A[N][M], A_T[M][N]
for(i = 0 ; i < N ; i++)
    for(j = 0 ; j < M ; j++)
        A_T[j][i] = A[i][j];

//IPM implementation
int index [8]={0, N, N*2, N*3, N*4, N*5, N*6, N*7};
__m256i vec_index = _mm256_load_si256((__m256i *) &index[0]);

for(i=0; i<N; i+=8)
    for(j=0; j<M; j+=8){
        //loading from columns
        vec_x0 = _mm256_i32gather_epi32 ((int const *)&A[i][j+0], vec_index, 4);
         ...
        vec_x7 = _mm256_i32gather_epi32 ((int const *)&A[i][j+7], vec_index, 4);
        //storing to the rows
        _mm256_store_si256((__m256i *)&A_T[j+0][i], vec_x0);
        ...
        _mm256_store_si256((__m256i *)&A_T[j+7][i], vec_x7);
  }
```

Implementations – TRA (2/2)



Gathering data from non-continuous and storing to continuous addresses



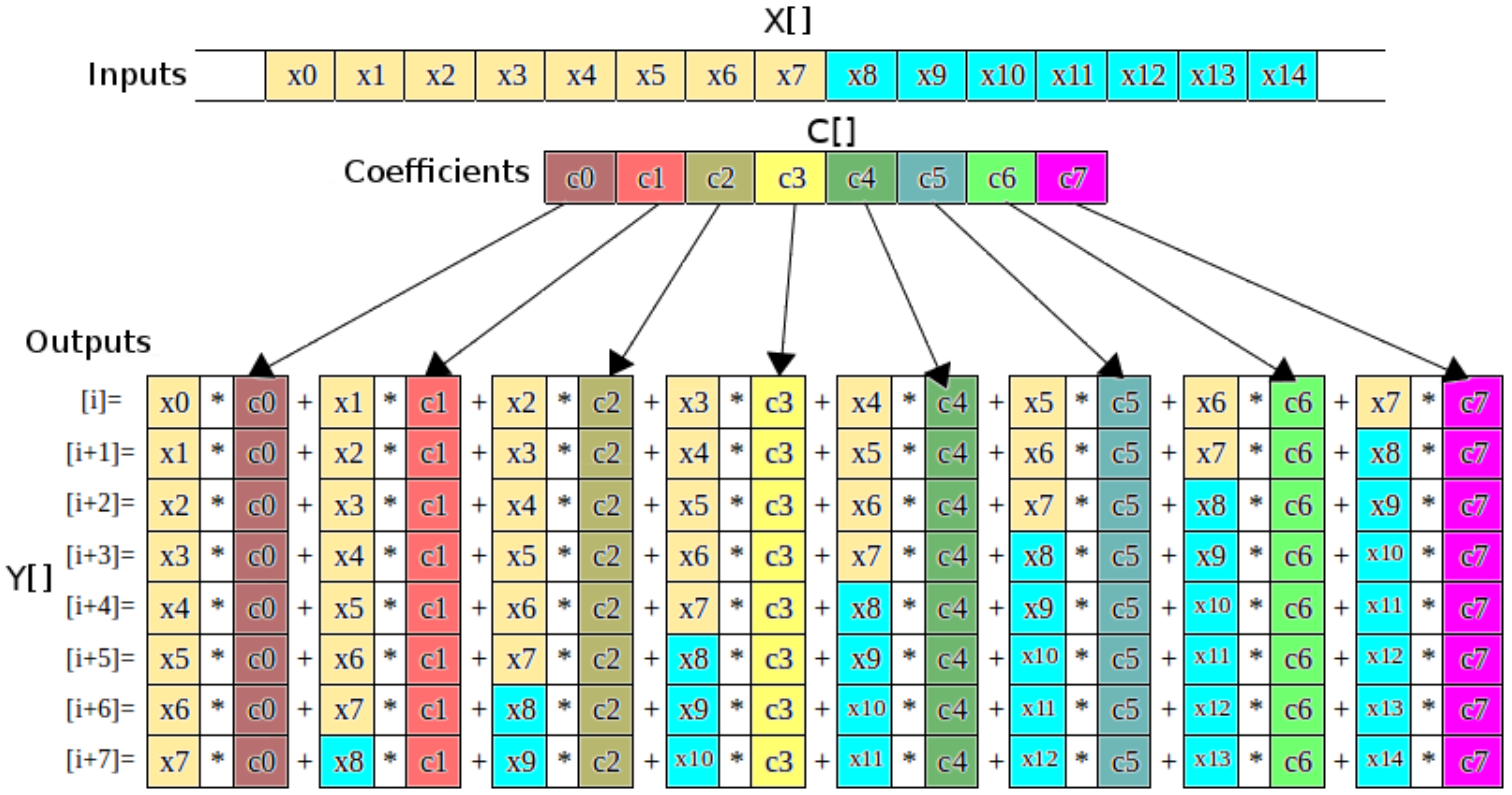
Transposing the 8x8 blocks to corresponding places

Implementations – FIR (1/2)

```
//Scalar implementation
//X[N], C[M], Y[N]
for(i = 0 ; i < N ; i++)
    for(j = 0 ; j < M ; j++)
        Y[i] += X[i+j] * C[j];
```

```
//IPM implementation
for(i=0; i < N; i += 8){
    vec_x = _mm256_load_si256 ( (__m256i *) &X[i]);
    vec_c = _mm256_set1_epi32 (C[0]);
    vec_result = _mm256_mullo_epi32 ( vec_c , vec_x);
    for(j=1; j < M; j++){
        vec_x = _mm256_loadu_si256( (__m256i *)&X[i+j]);
        vec_c = _mm256_set1_epi32 (C[j]);;
        vec_t  = _mm256_mullo_epi32( vec_c, vec_x);
        vec_result  = _mm256_add_epi32( vec_t, vec_result);
    }
    _mm256_store_si256((__m256i *)&Y[i], vec_result);
}
```

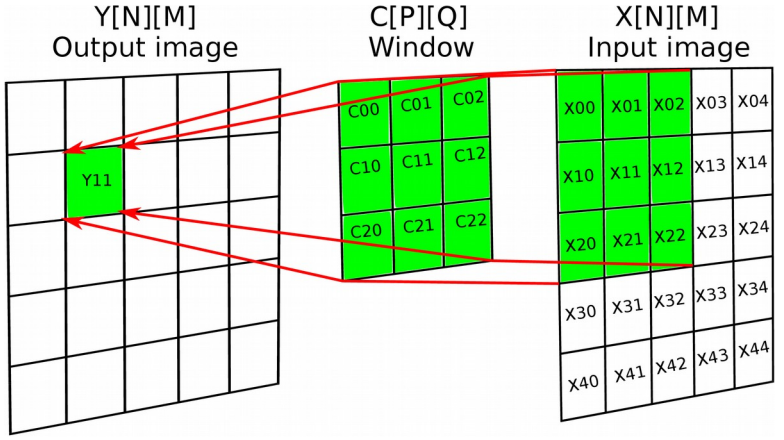
Implementations – FIR (2/2)



Mapping strategy of FIR filter to AVX2 instructions

Implementations – CON (1/6)

```
//Scalar implementation
//X[N ][M ], C[P ][Q]
for(i = P/2; i < N − P/2; i++)
    for(j = Q/2; j < M − Q/2; j++){
        temp = 0;
        for(k = 0; k < P ; k++)
            for(l = 0; l < Q; l++)
                temp+ = C[k][l]) ∗ X[i − (P/2) + k][j − (Q/2) + l];
        Y[i][j] = ((temp/div) + offset);
 }
```

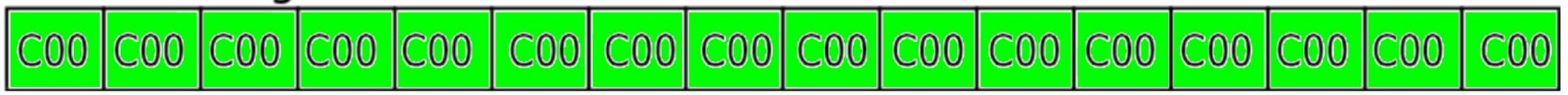


2D Convolution (P=Q=3)

Implementations – CON (2/6)

- Broadcast of Coefficients (IPM-BC)

A vector register

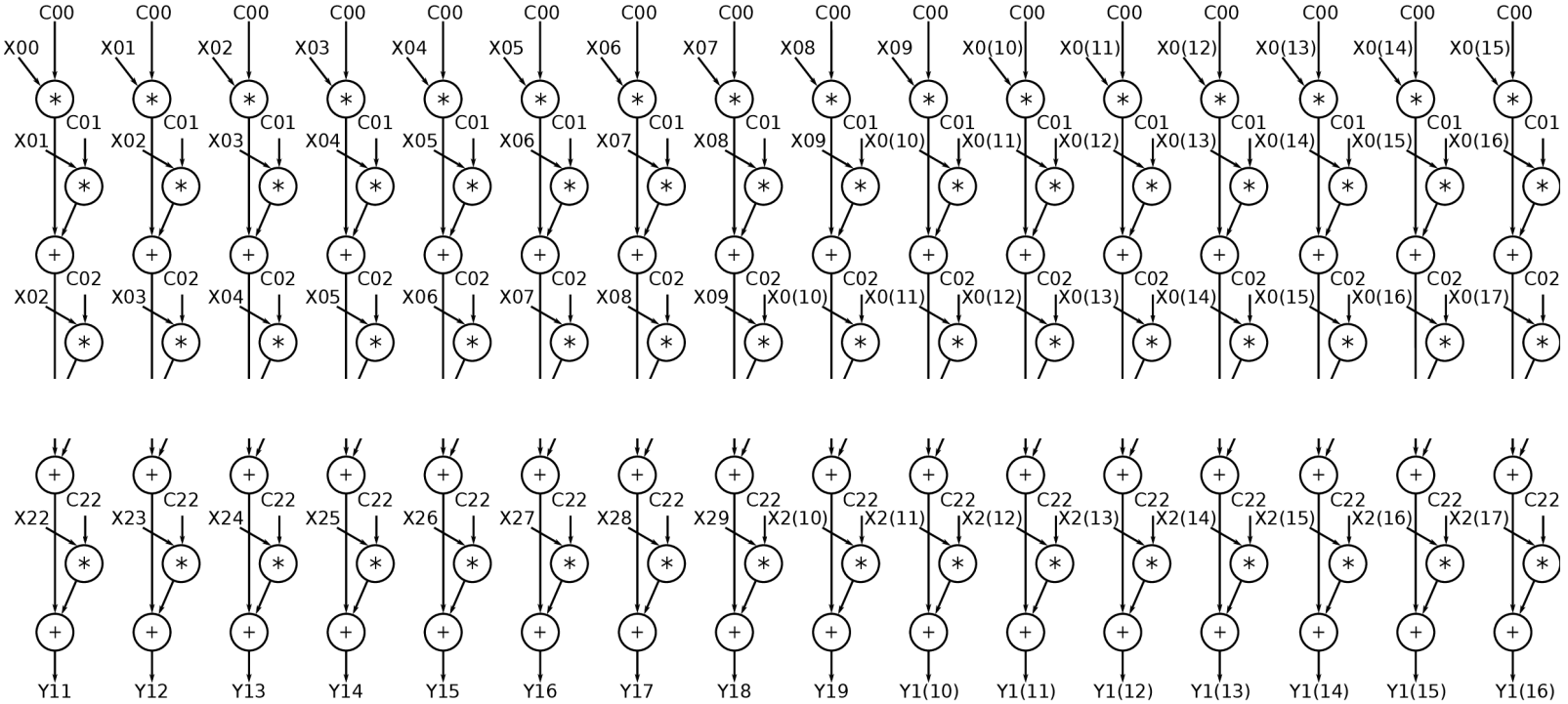


- Repetition of Coefficients (IPM-RC)

A vector register

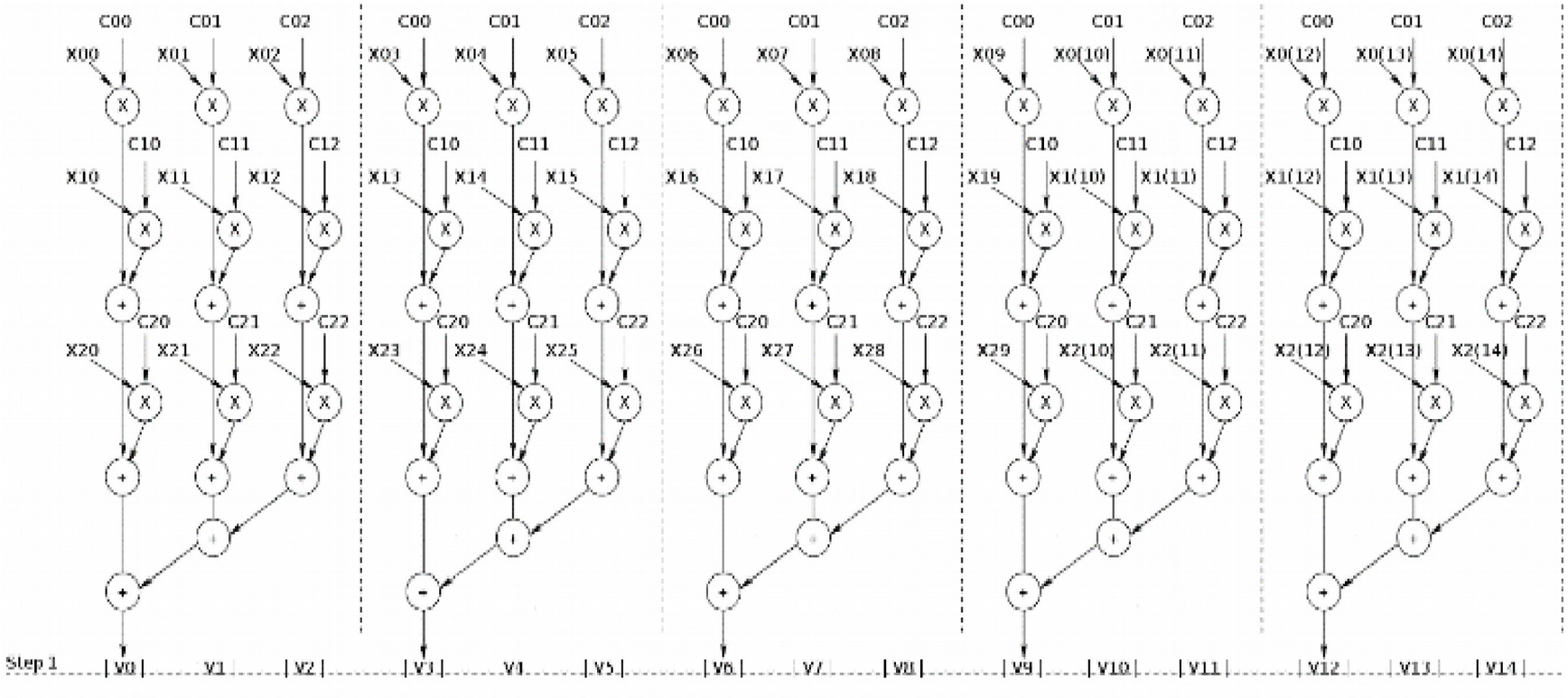
Implementations – CON (3/6)

- First method: Broadcast of Coefficients



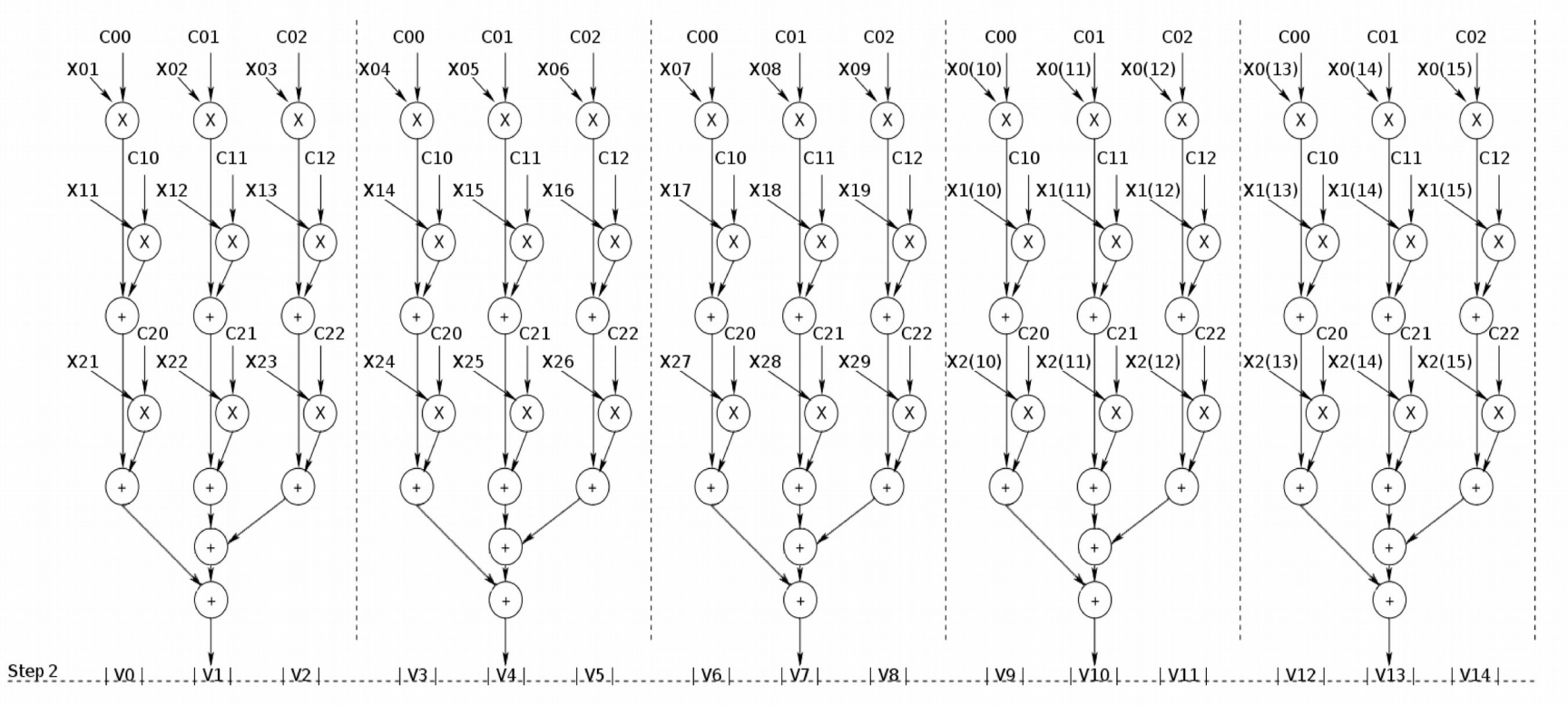
Implementations – CON (4/6)

- Second method: Repetition of Coefficients



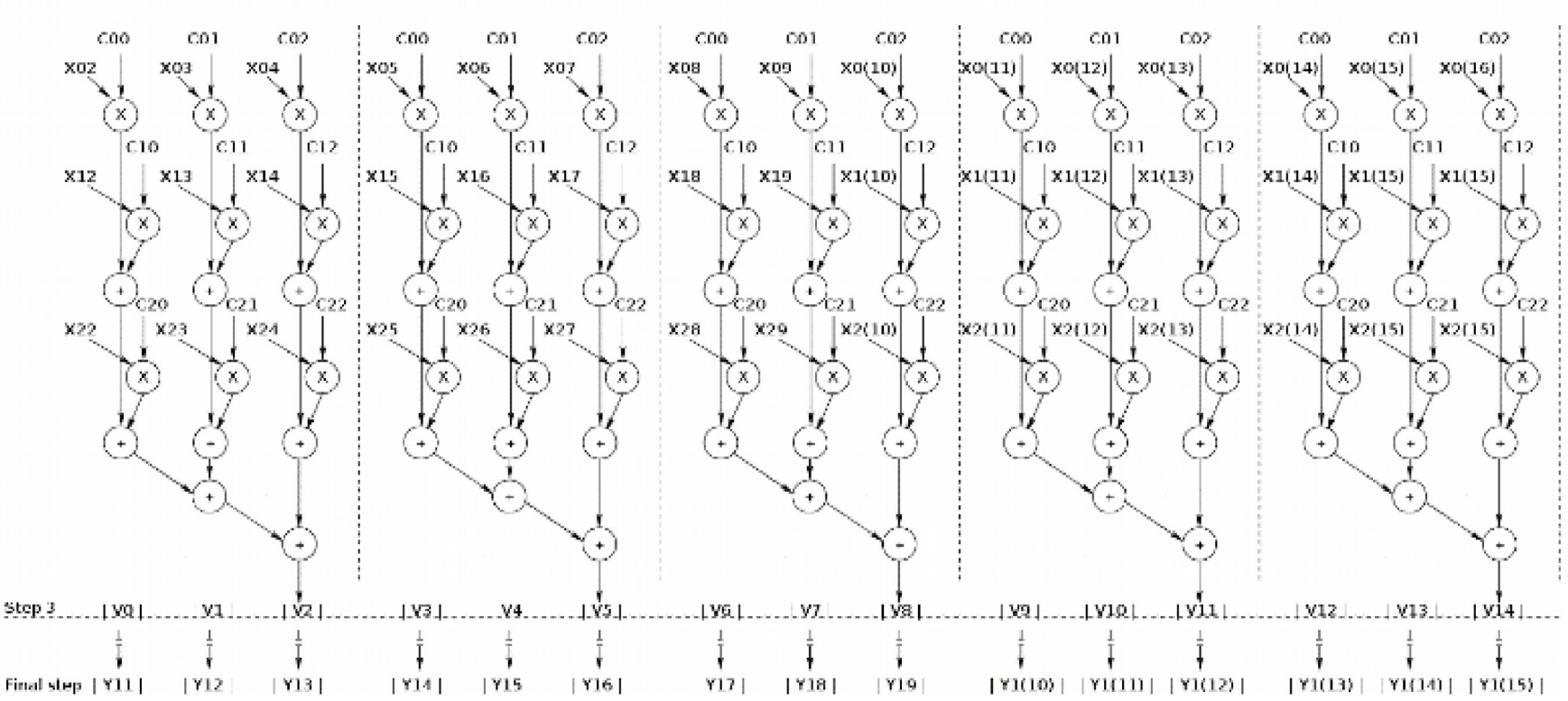
Implementations – CON (5/6)

- Second method: Repetition of Coefficients



Implementations – CON (6/6)

- Second method: Repetition of Coefficients

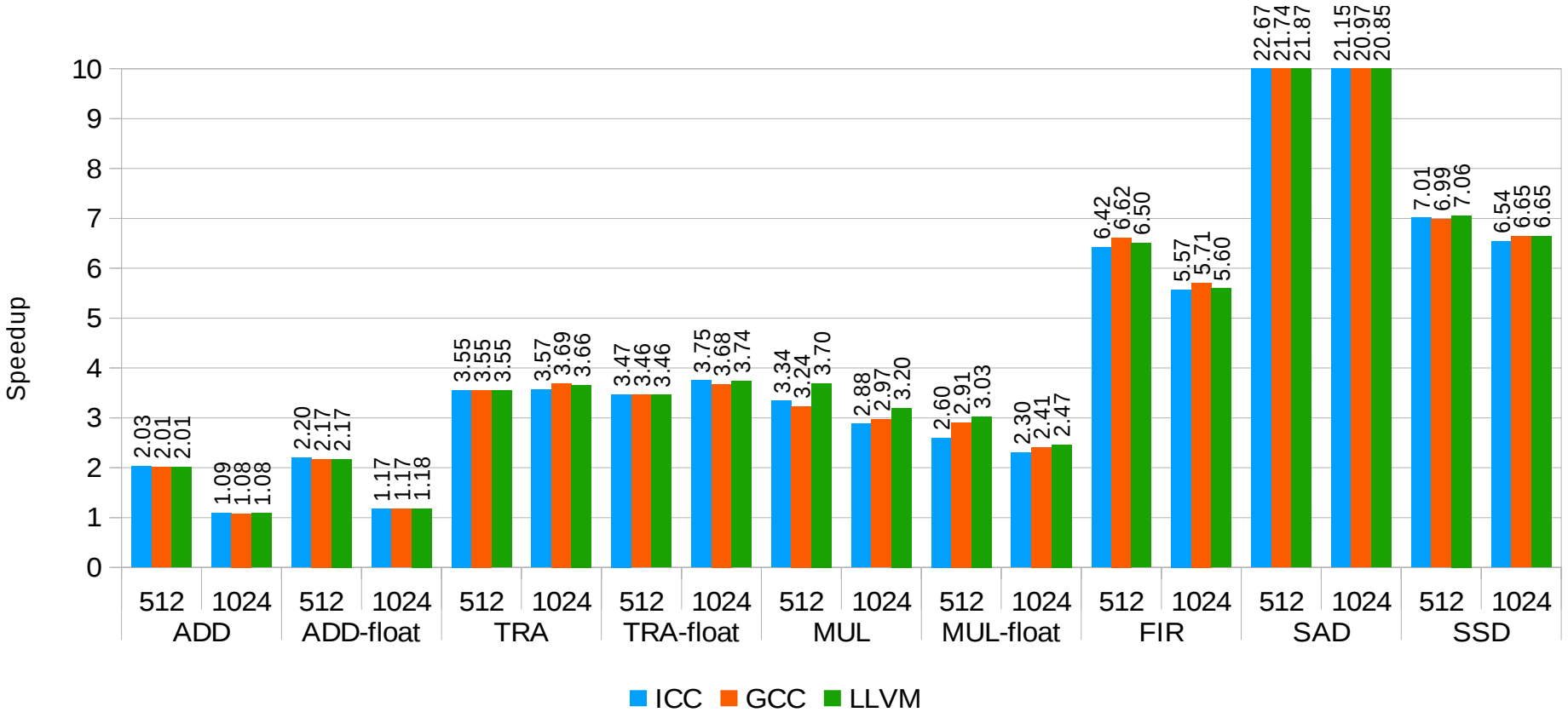


Environment Setup

Platform Specification

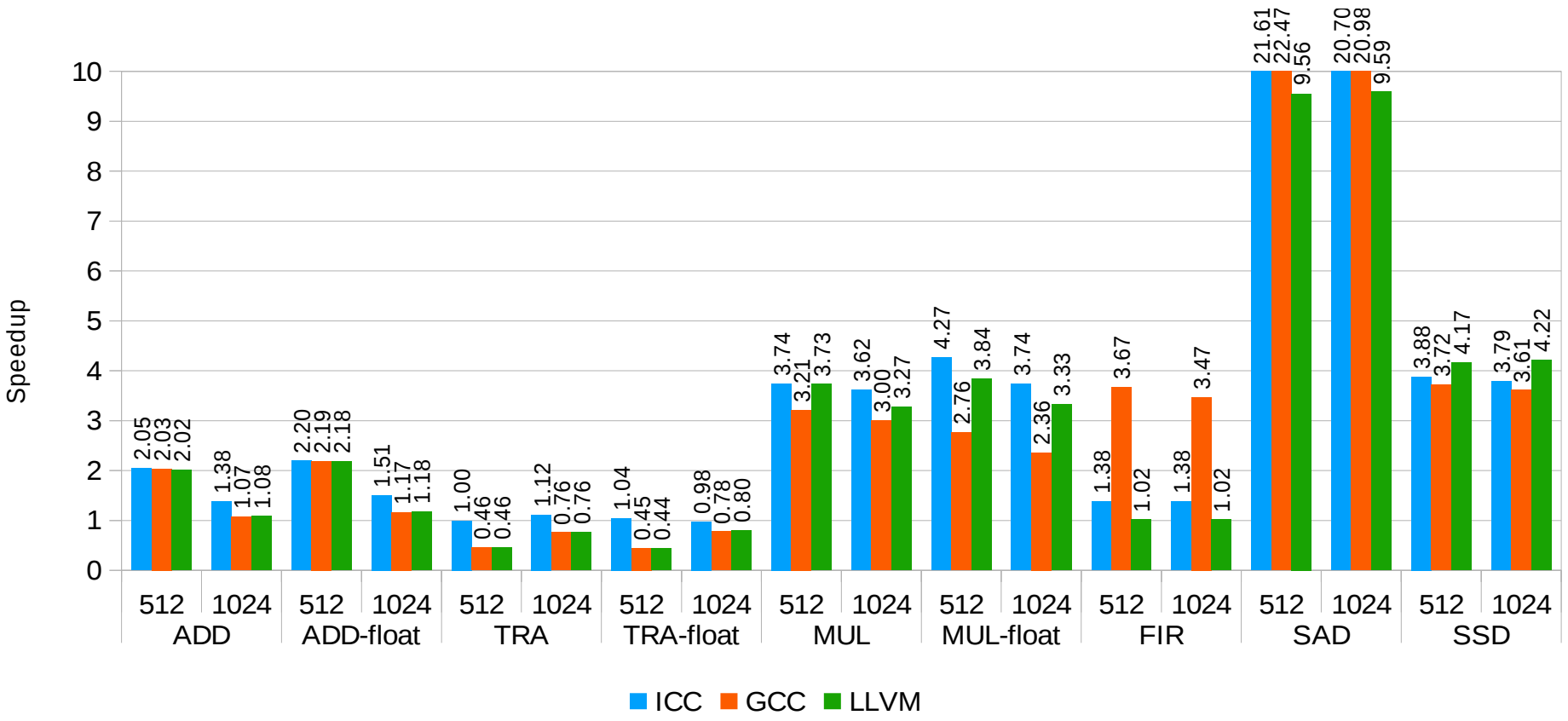
CPU	Intel Corei7- 6700HQ
Register width	Maximum 256 bits
L1 Data Cache	32KB, 8-way set associative, the fastest latency: 4 cycles, 2x32B load + 1x32B store
L2 Cache	256 KB,4-way set associative, the fastest latency: 12 cycles
L3 Cache	Up to 2 MB per core, up to 16-ways, the fastest latency: 44 cycles
Operating system	Fedora 27, 64-bit
Compilers	ICC 18.0.1, GCC 7.2.1 and LLVM 5.0.1
Programming tools	C and x86 Intrinsics (x86intrin.h)
Disable vectorizing	icc -O3 -no-vec
	gcc -O3 -fno-tree-vectorize -fno-tree-slp-vectorize
	clang -O3 -fno-vectorize -fno-slp-vectorize

Experimental Results (1/18)



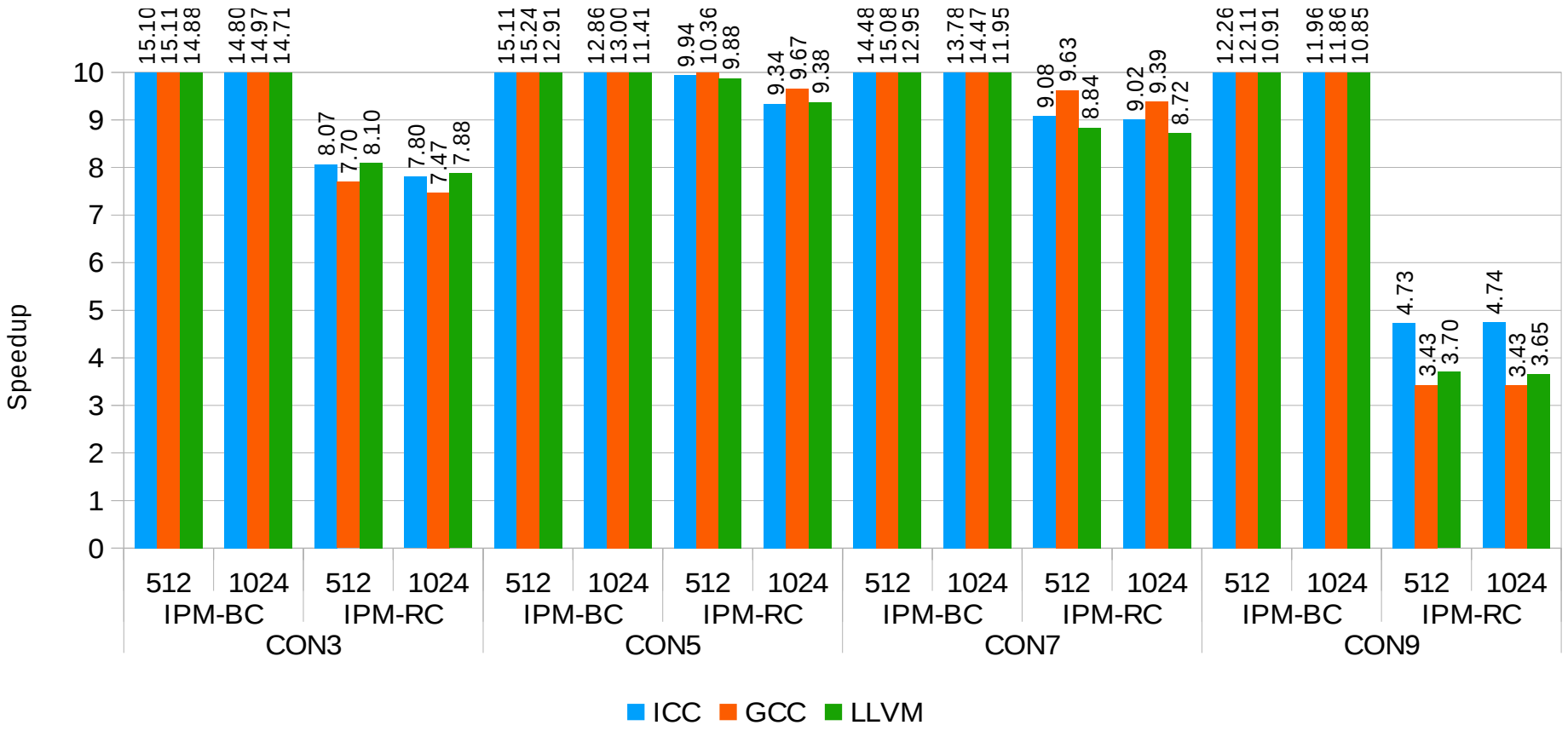
Performance Improvement of IPM-AVX2 over Scalar Implementation (Single-core)

Experimental Results (2/18)



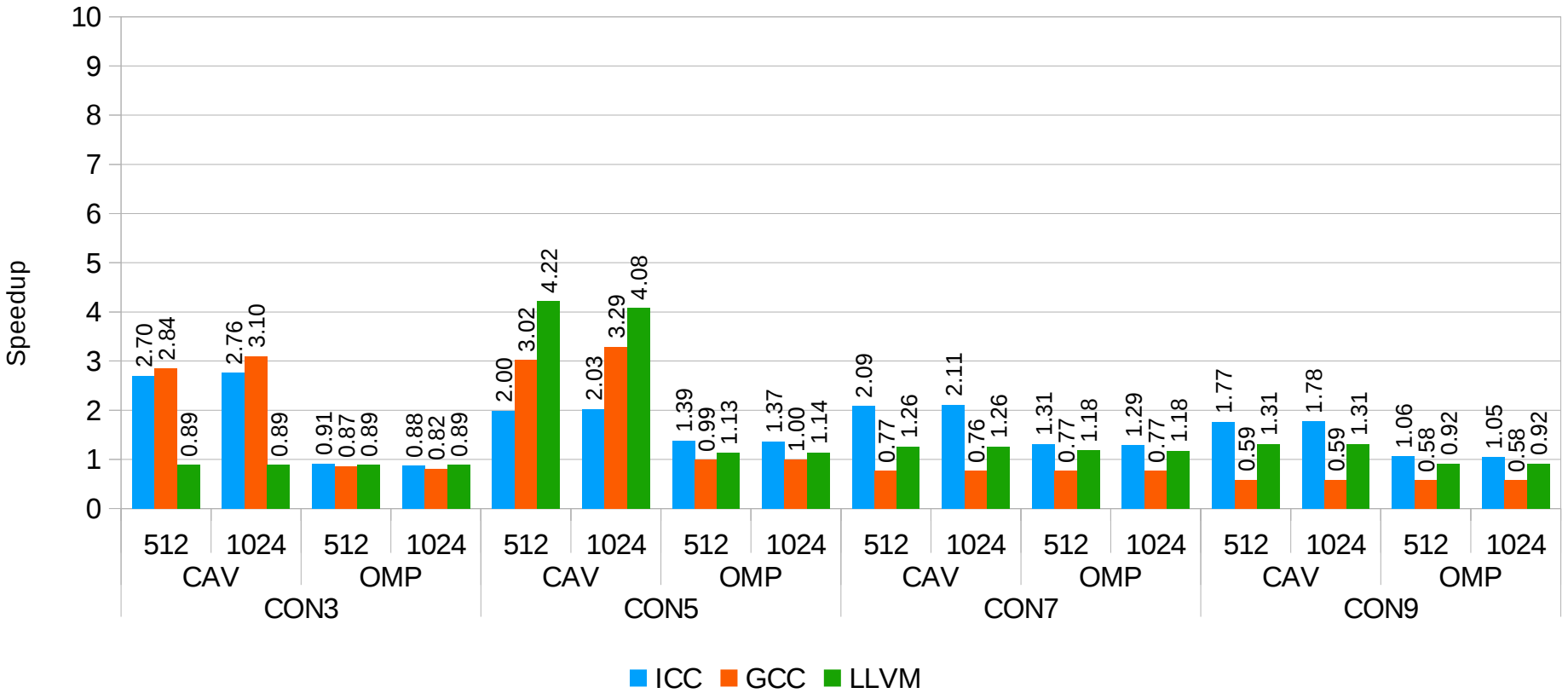
Performance Improvement of CAV-AVX2 over Scalar Implementation (Single-core)

Experimental Results (3/18)



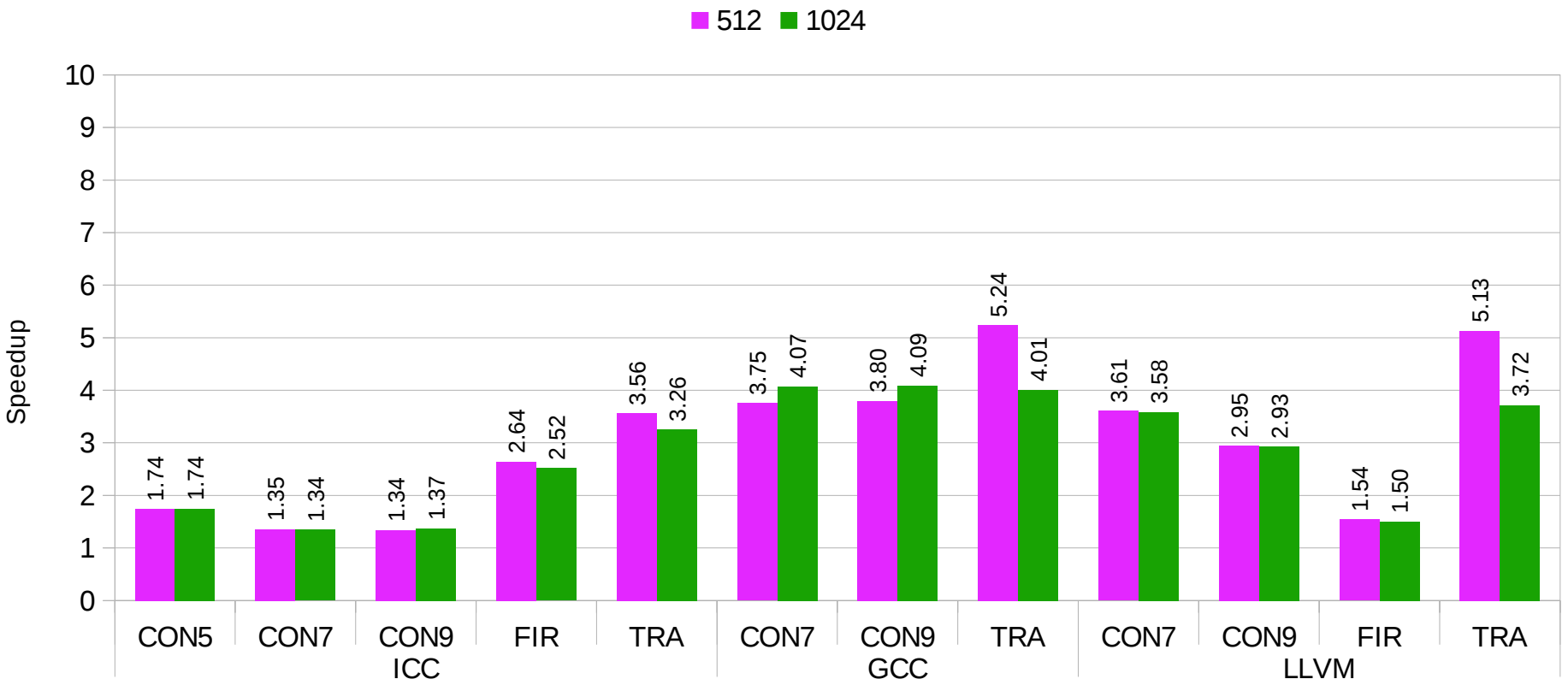
Performance Comparison of Explicit Implementation of Convolution Matrix over Scalar Implementation (Single-core)

Experimental Results (4/18)



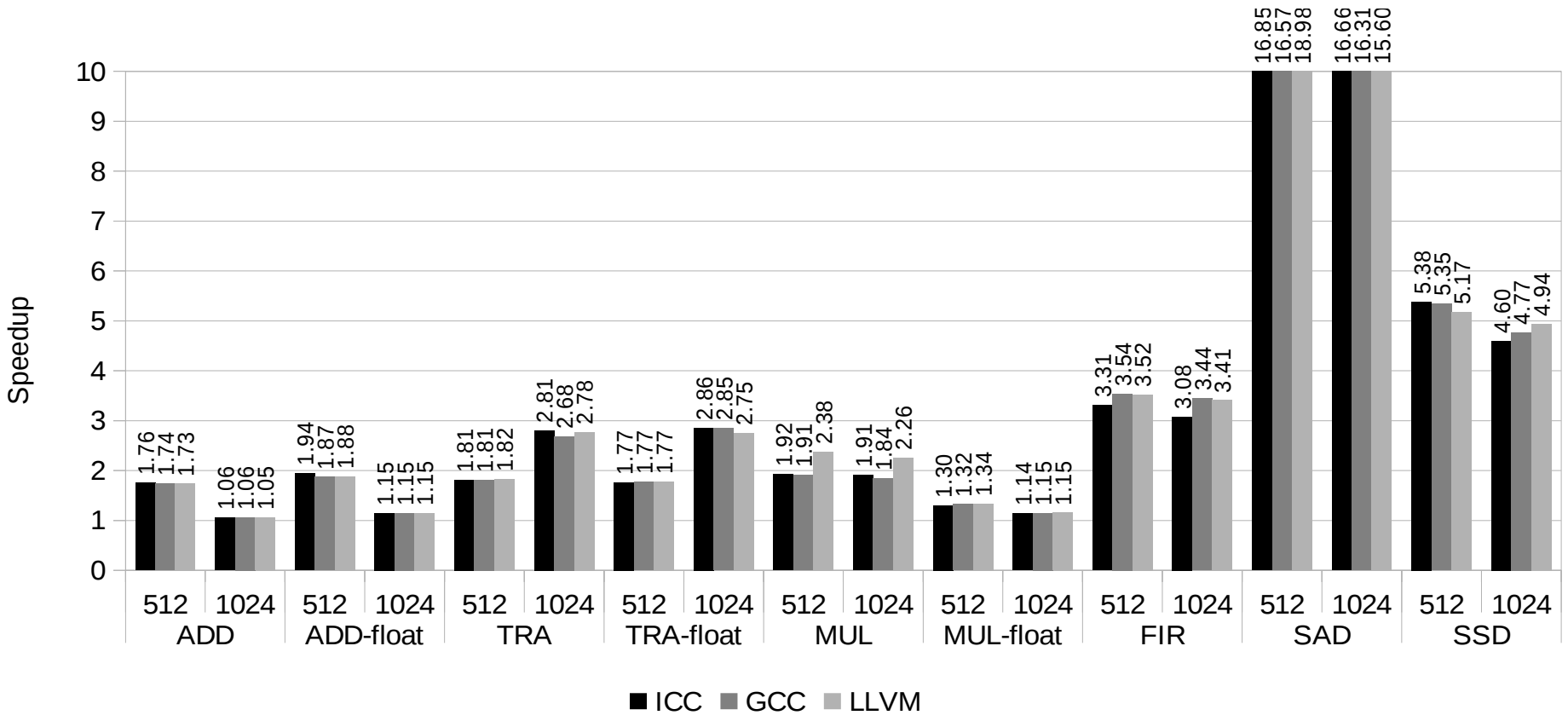
Performance Comparison of Implicit Implementation of Convolution Matrix over Scalar Implementation (Single-core)

Experimental Results (5/18)



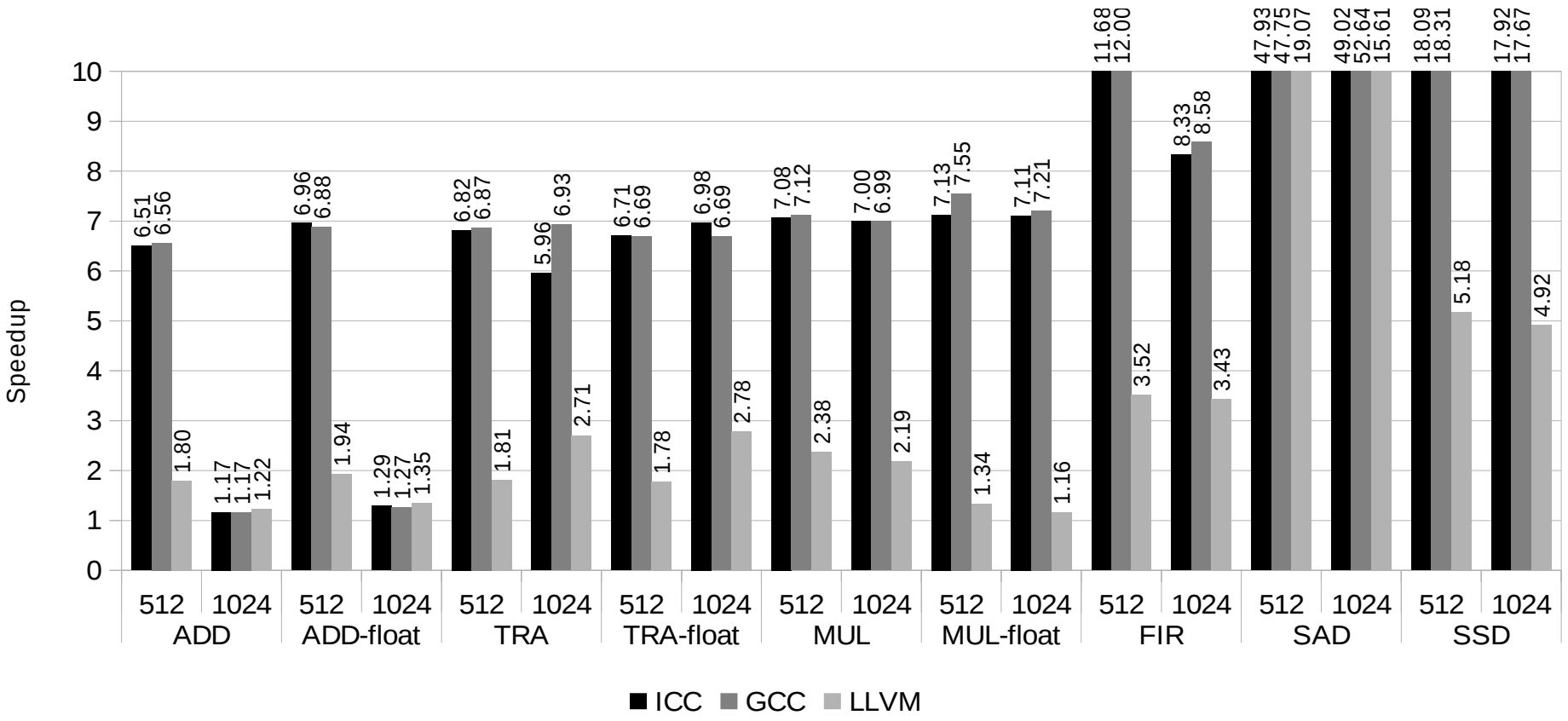
Performance Comparison of Proposed Framework over state-of-the-art CAVs

Experimental Results (6/18)



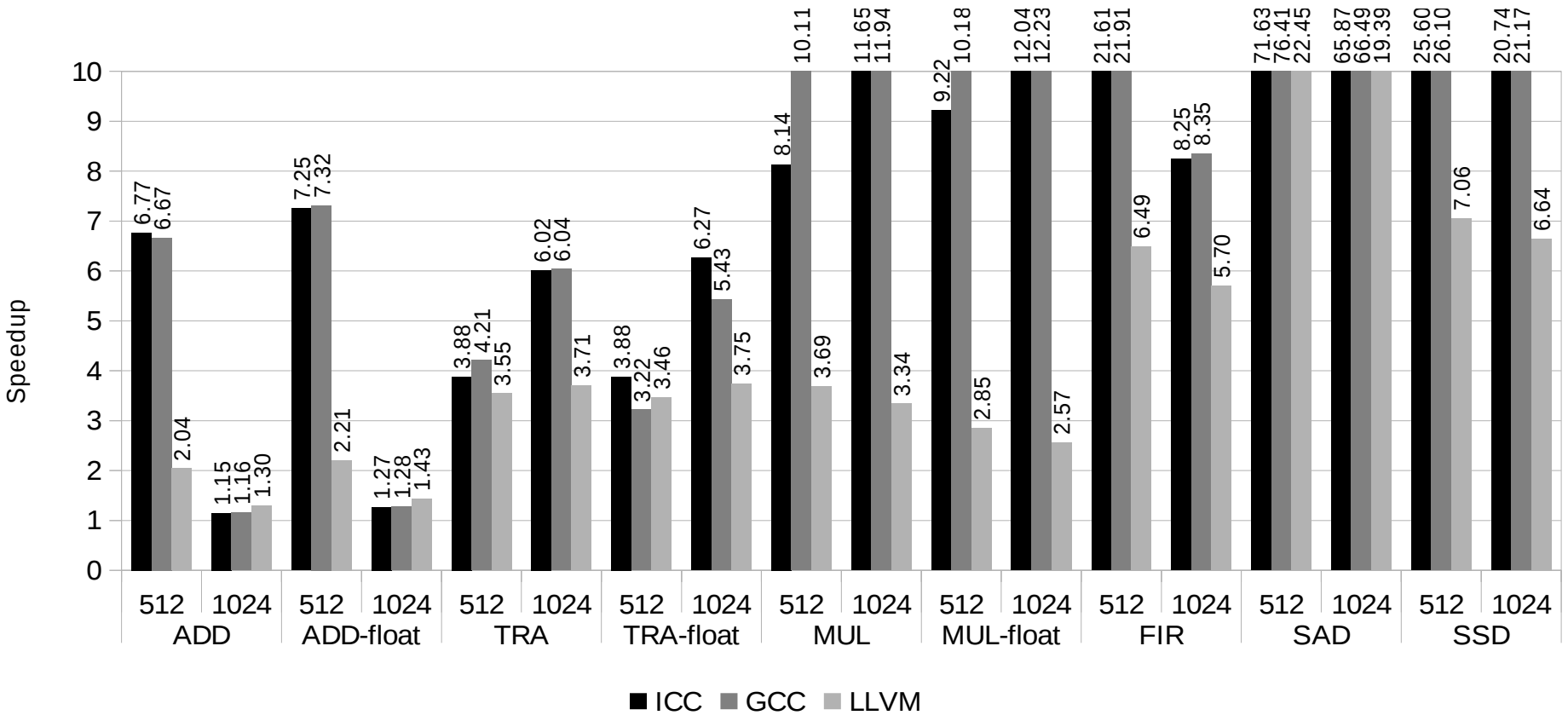
Performance Improvement of IPM-SSE4 over Scalar Implementation (Single-core)

Experimental Results (7/18)



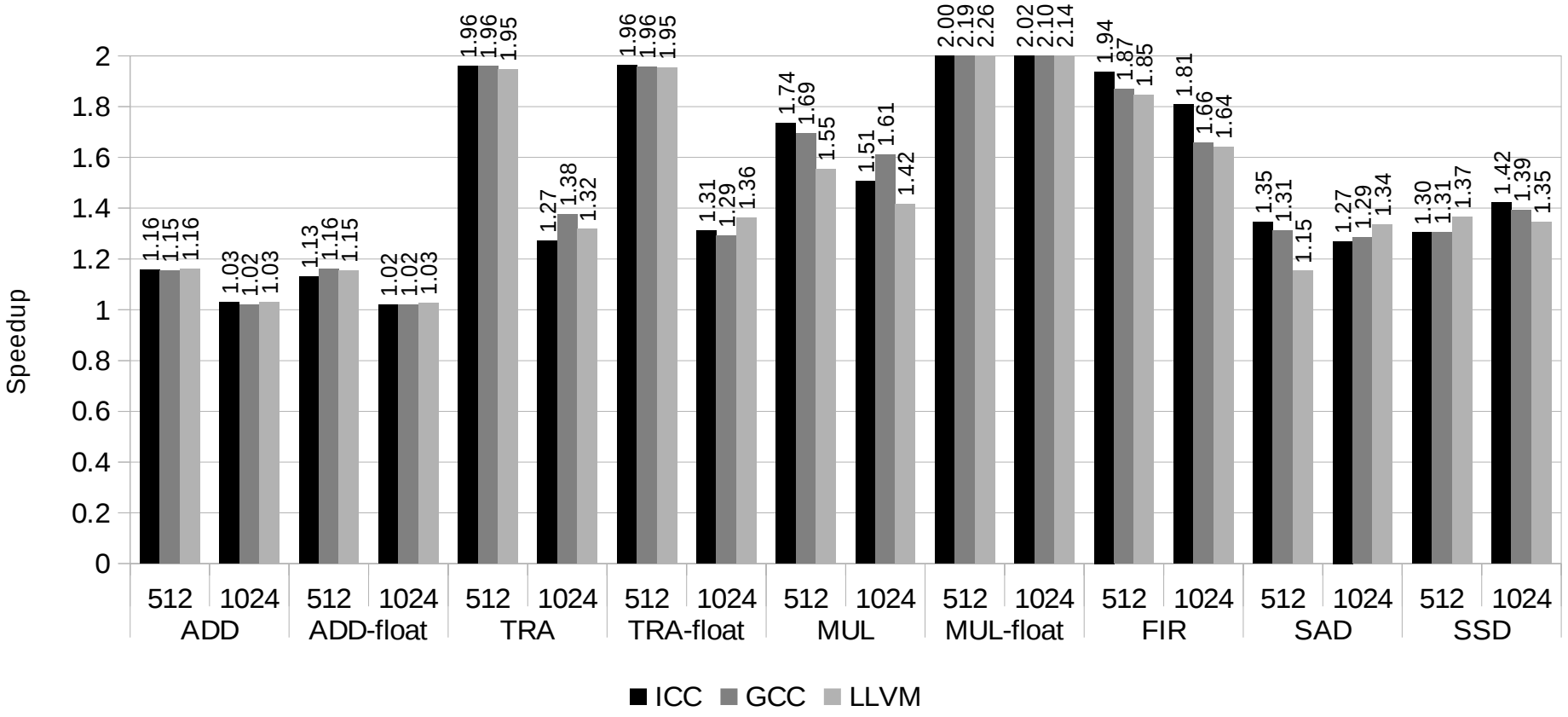
Performance Improvement of IPM-SSE4 over Scalar Implementation (Multi-core)

Experimental Results (8/18)



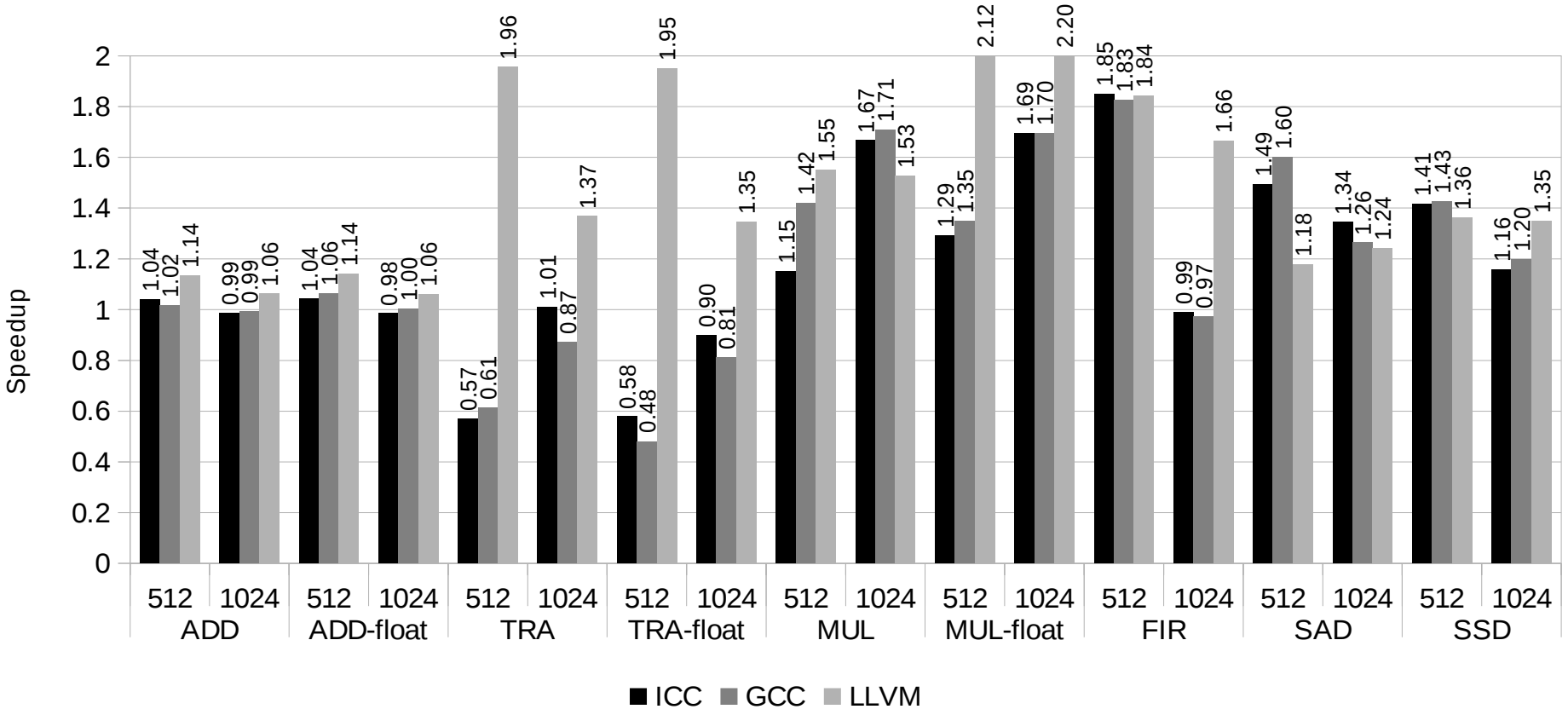
Performance Improvement of IPM-AVX2 over Scalar Implementation (Multi-core)

Experimental Results (9/18)



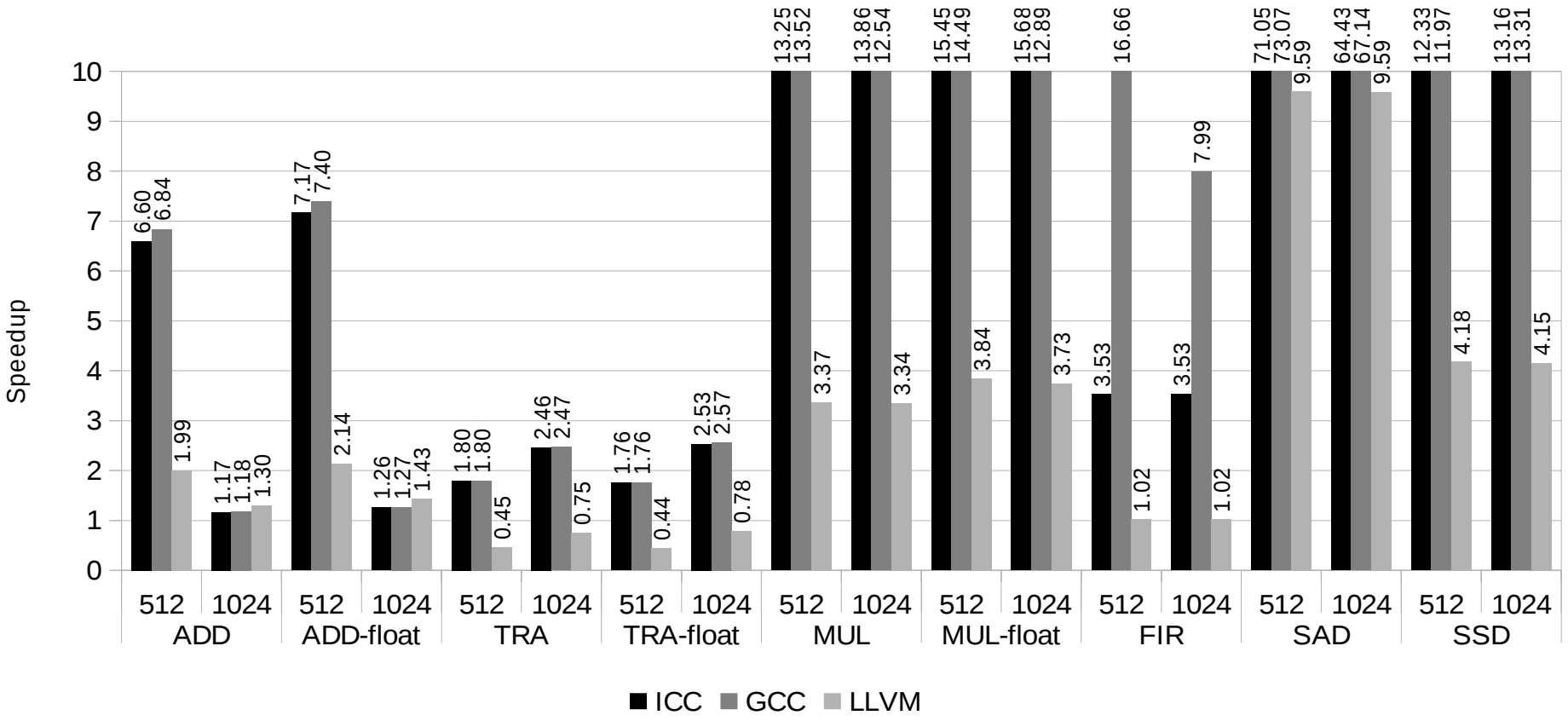
Performance Improvement of IPM-AVX2 over IPM-SSE4 (Single-core)

Experimental Results (10/18)



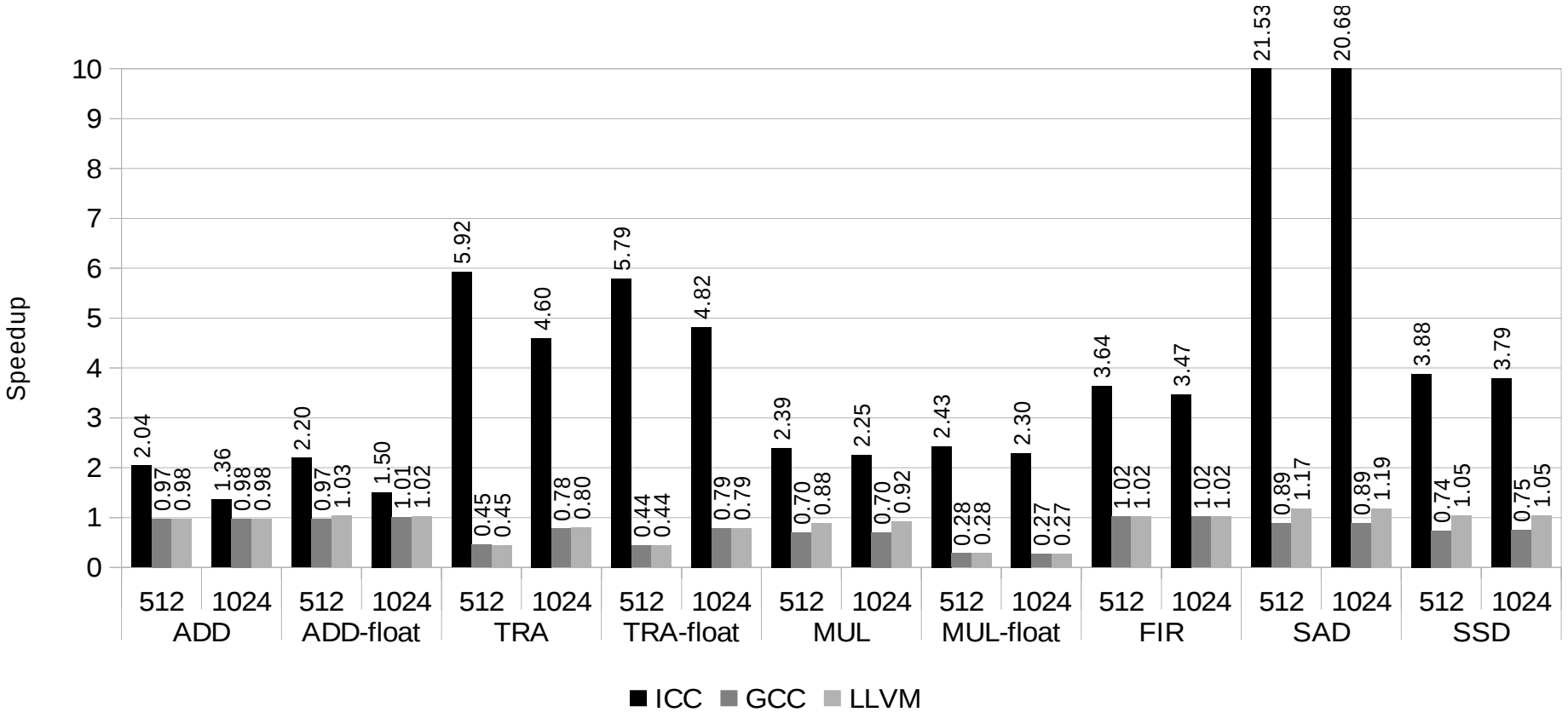
Performance Improvement of IPM-AVX2 over IPM-SSE4 (Multi-core)

Experimental Results (11/18)



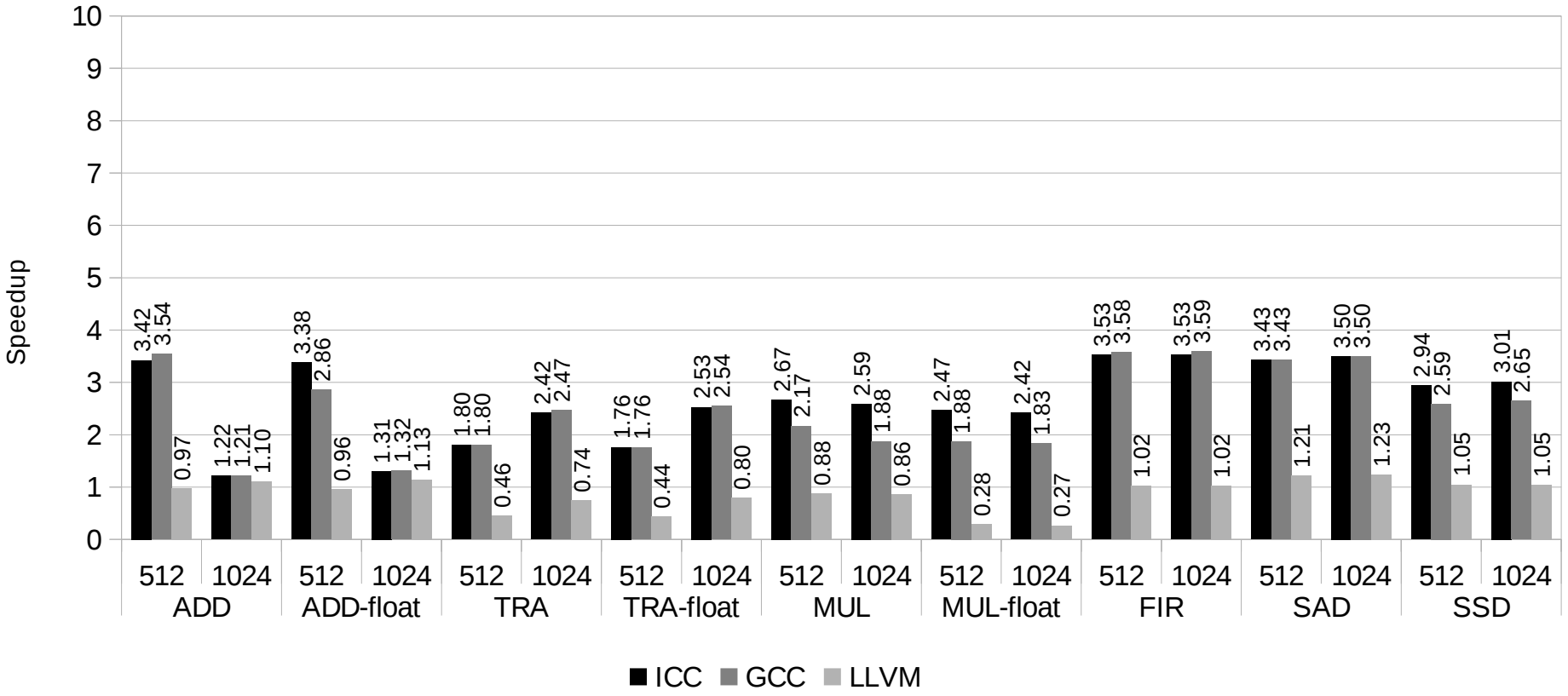
Performance Improvement of CAV-AVX2 over Scalar Implementation (Multi-core)

Experimental Results (12/18)



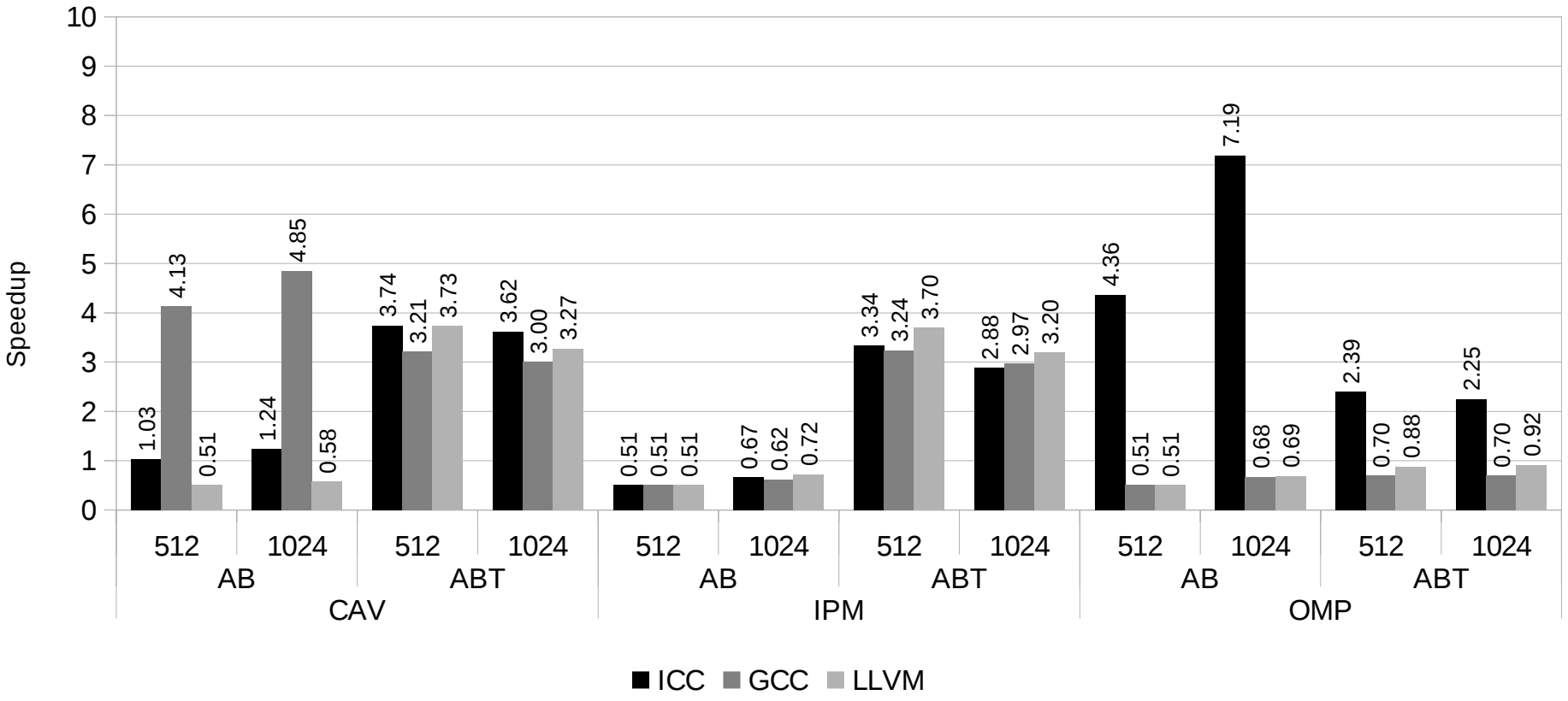
Performance Improvement of OMP-AVX2 over Scalar Implementation (Single-core)

Experimental Results (13/18)



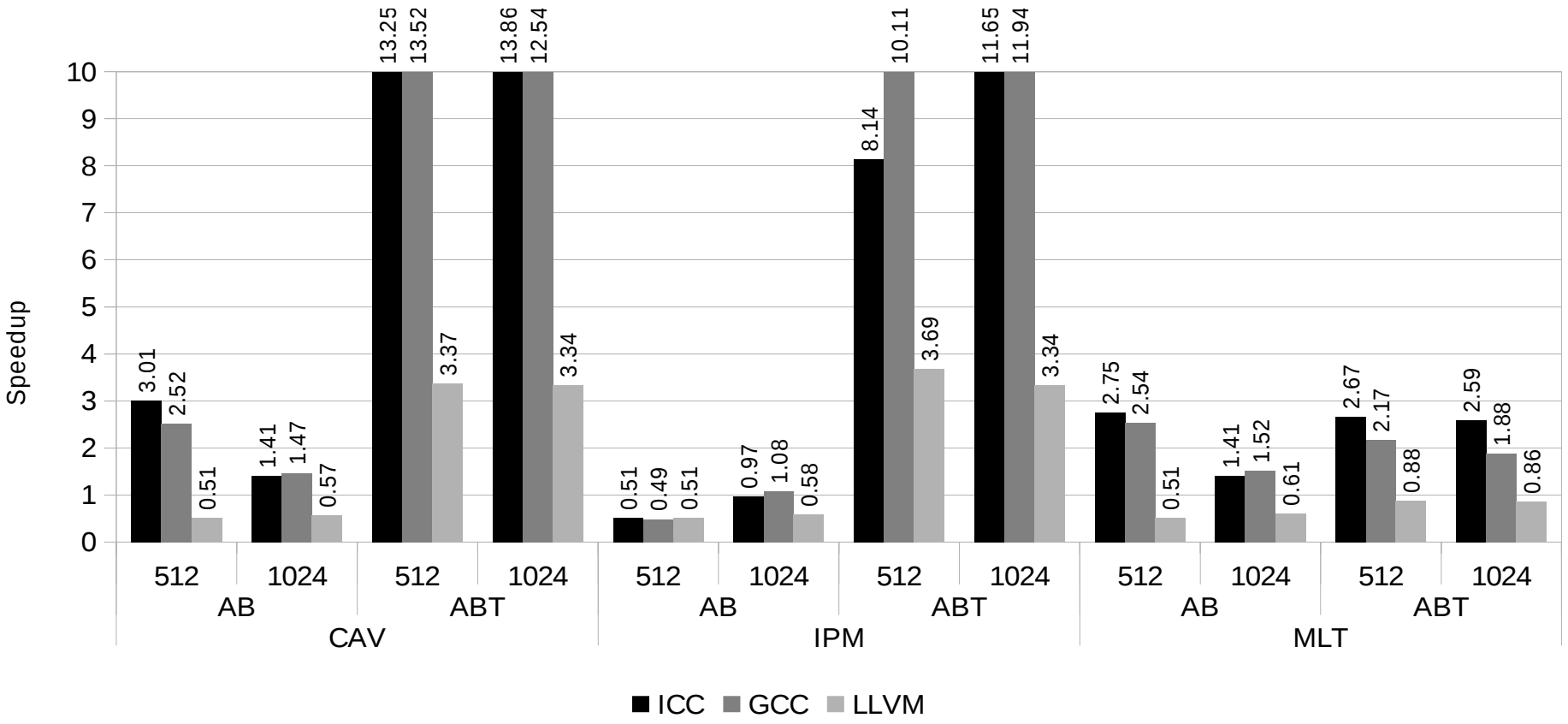
Performance Improvement of OMP-MLT over Scalar Implementation (Multi-core)

Experimental Results (14/18)



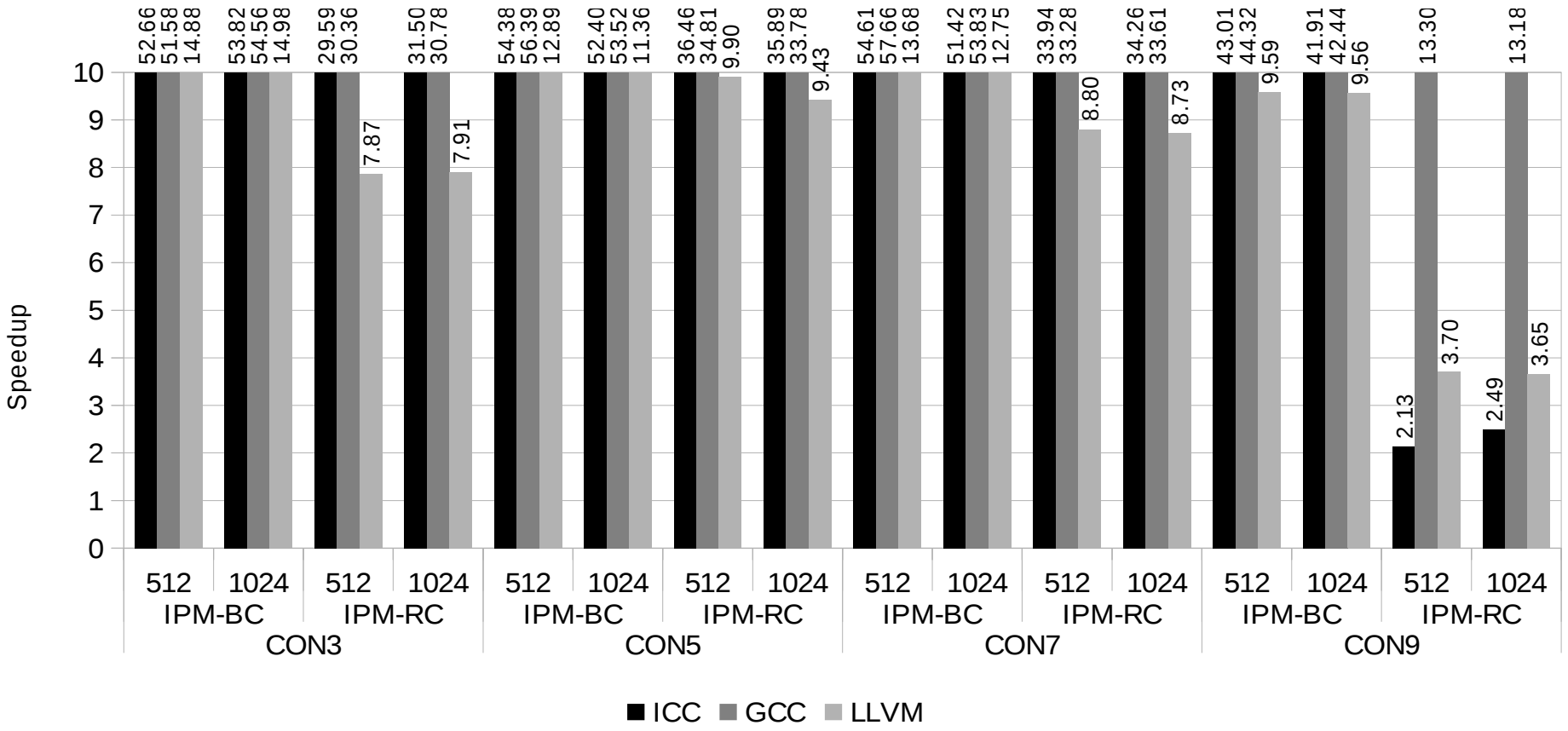
Performance Comparison of Different Matrix-Matrix Multiplication over Scalar Implementation (Single-core)

Experimental Results (15/18)



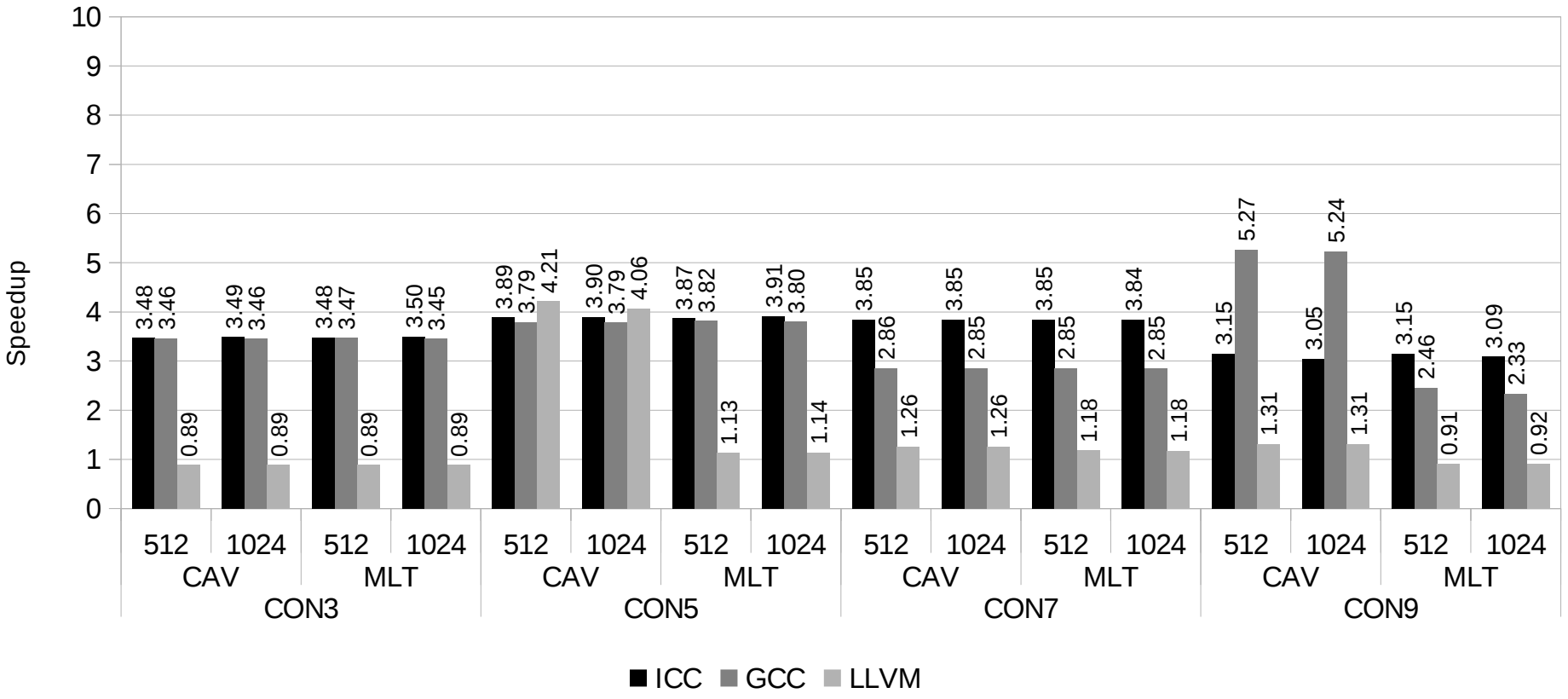
Performance Comparison of Different Matrix-Matrix Multiplication over Scalar Implementation (Multi-core)

Experimental Results (16/18)



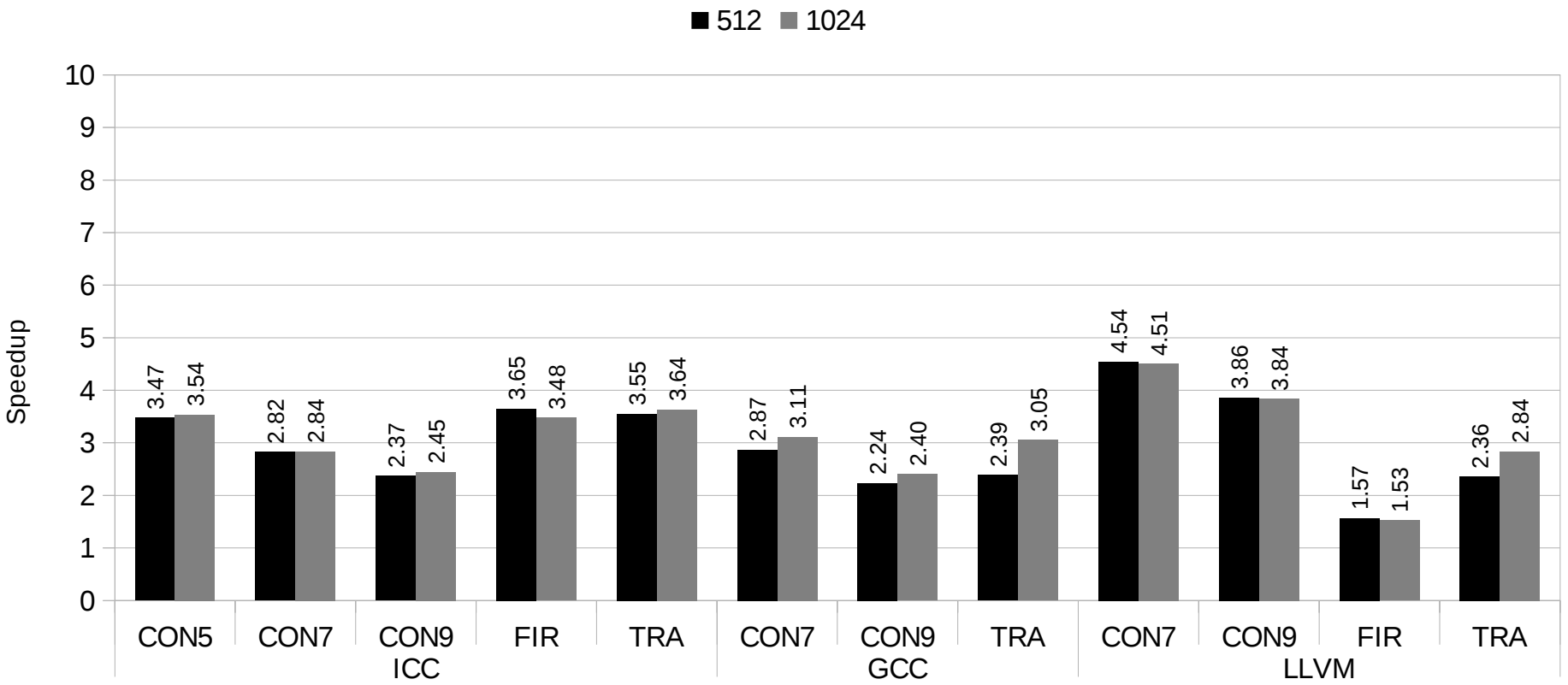
Performance Comparison of Explicit Implementation of Convolution Matrix over Scalar Implementation (Multi-core)

Experimental Results (17/18)



Performance Comparison of Implicit Implementation of Convolution Matrix over Scalar Implementation (Multi-core)

Experimental Results (18/18)



Performance Comparison of Proposed Framework over Scalar Implementation

Conclusions

- Multimedia applications (MMAs) are very important
- Multimedia Kernels (MMKs) are the bottleneck of MMAs
- SIMD technologies are available to improve the performance of MMKs (As we've done)
- Mapping strategy of a certain kernel to SIMD instructions yields different speedups
- ICC gains more performance than other compilers
- Source code is important for auto-vectorizer.
- A new framework has been proposed to improve the performance of compiler's automatic vectorization

Future Work Recommendations

- A perfect vectorizing compiler has not been produced yet
- An x86 compatible architecture for GPPs (poor Itanium)
- New and more efficient standards for multimedia

Academic Outputs

- IEEE's Publications
  - ***H. Amiri*** *and A. Shahbahrami, "High Performance Implementation of 2D Convolution using Intel's Advanced Vector Extensions," in 2017 Artificial Intelligence and Signal Processing (AISP), 2017.*
  - ***H. Amiri*** *and A. Shahbahrami, "High Performance Implementation of 2-D Convolution Using AVX2," in 19th International Symposium on Computer Architecture and Digital Systems (CADS'17), 2017.*
- Elsevier's Publications
  - ***H. Amiri****, A. Shahbahrami, A. Pohl, and B. Juurlink, "Performance Evaluation of Implicit and Explicit SIMDization," J. Microprocess. Microsystems, 2018. (****Submitted 08 Nov 2017 – under review****)*
  - ***H. Amiri*** *and A. Shahbahrami, "A Comparison of Intel's Vector Extensions," J. Syst. Archit., 2018. (****Submitted 14 Jan 2018 – under review****)*

Thank you for your attention