

**İstanbul Aydın University**

**Department of Computer Engineering**

# COMPUTER ORGANIZATION (COM282)

**FINAL PROJECT**

**Group Members:**

**Görkem Genar Akkaya – B1705.010095**

**Mustafa Erdoğan – B1505.010012**

**ABSTRACT**

Traditional computer hardware architectures like X86 were clearly not designed for SIMD operation. Over the years, Intel has regularly added extensions to the x86 ISA to reinforce it's practicality , and SIMD extensions were a number of the earliest to appear002E

It started with MMX on the Pentium, however that had pretty obvious limitations. Later on, Intel added the SSE instruction set extension, that removed a number of these limitations. They more iterated on SSE within the coming back decade, till they eventually introduced AVX.

AVX introduced some pretty vital enhacements over its predecessor, SSE4.x

The vector register size was exaggerated from 128bit to 256 bit, doubling potential outturn. Instructions went from the standart a pair of quantity format (a = a+ b) to a 3 operand format ( a = b + c), improving flexibility. Alignement rules were conjointly relaxed.

**INTRODUCTION**

A subword may be lower preciseness unit of data contained among a word. In subword parallelism, multiple subwords are packed into a word then process all words. With the correct subword boundaries this system leads in parallel processing of subwords. Since a similar instruction is applied to all subwords among the word, This is often a sort of SIMD(Single Instruction Multiple Data) processing.

**REVIEW**

It is possible to use subword parallelism to noncontiguous subwords of various sizes inside a word. In sensible implementation is easy if subwords are same size and they are contiguous inside a word. The data parallel programs that benefit from subword parallelism tend to process data that are of the identical size.

For example if word size is 32bits and subwords sizes are 8 and 16 and bits. Therefore an instruction operates on four 8bit subwords, two 16bit subwords, one 32bit subwords in parallel.

· Subword parallelism is an efficient and versatile answer for media processing as result of algorithm exhibit a good deal of data parallelism on lower precision data.

· It is conjointly helpful for computations unrelated to transmission that exhibit data parallelism on lower precision data.

· Graphics and audio applications can benefit from performing simultaneous operations on short vectors

The degree of SIX/ID parallelism inside an instruction, then, depends upon the scale of the subwords. Data parallelism refers to an algorithm’s execution of the identical program module on completely different sets of dacdSubword parallelism is additionally helpful for computations unrelated to transmission that exhibit data parallelism on lower preciseness data. One key advantage of subword parallelism is that it permits all-purpose processors to take advantage of wider word sizes even once not processing high-precision data.

**HARDWARE ACCELERATION METHODS**

**1-UNOPTIMIZED DGEMM CODE**

**Unoptimized Code:**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <cuda.h>

#include <device\_functions.h>

#include <cuda\_runtime\_api.h>

#include <immintrin.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define ARRAY\_SIZE 256

void dgemm(int n, const double\* A, const double\* B, double\* C)

{

int i, j, k;

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

C[i\*n + j] = 0;

for (k =0; k < n; k++)

{

C[i\*n + j] += A[k + i \* n] \* B[k\*n + j];

}

}

}

}

int main()

{

double \*A = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double \*B = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double \*C1 = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double \*C2 = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

for (int i = 0; i < ARRAY\_SIZE \* ARRAY\_SIZE; i++)

{

A[i] = rand() % 100;

B[i] = rand() % 100;

}

clock\_t t;

t = clock();

dgemm(ARRAY\_SIZE, A, B, C1);

t = clock() - t;

double elapsed\_time = ((double)t) / CLOCKS\_PER\_SEC;

printf("Unoptimized DGEMM code took %.6f seconds to execute.\n", elapsed\_time);

return 0;

}

**2-AVX OPTIMIZATION CODE**

**Unoptimized Code:**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <cuda.h>

#include <device\_functions.h>

#include <cuda\_runtime\_api.h>

#include <immintrin.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define ARRAY\_SIZE 256

void dgemm (int n, double\* A, double\* B, double\* C)

3. {

4. for ( int i = 0; i < n; i+=4 )

5. for ( int j = 0; j < n; j++ ) {

6. \_\_m256d c0 = \_mm256\_load\_pd(C+i+j\*n); /\* c0 = C[i][j] \*/

7. for( int k = 0; k < n; k++ )

8. c0 = \_mm256\_add\_pd(c0, /\* c0 += A[i][k]\*B[k][j] \*/

9. \_mm256\_mul\_pd(\_mm256\_load\_pd(A+i+k\*n),

10. \_mm256\_broadcast\_sd(B+k+j\*n)));

11. mm256\_store\_pd(C+i+j\*n, c0); /\* C[i][j] = c0 \*/

12. }

13. }

int main()

{

double \*A = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double \*B = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double \*C1 = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double \*C2 = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

for (int i = 0; i < ARRAY\_SIZE \* ARRAY\_SIZE; i++)

{

A[i] = rand() % 100;

B[i] = rand() % 100;

}

clock\_t t;

t = clock();

dgemm(ARRAY\_SIZE, A, B, C1);

t = clock() - t;

double elapsed\_time = ((double)t) / CLOCKS\_PER\_SEC;

printf("Unoptimized DGEMM code took %.6f seconds to execute.\n", elapsed\_time);

return 0;

}

**3-AVX+UNROLL OPTIMIZATION CODE**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <cuda.h>

#include <device\_functions.h>

#include <cuda\_runtime\_api.h>

#include <immintrin.h>

#include <intrin.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define ARRAY\_SIZE 256

#define UNROLL (4)

void dgemm(

int n, double\* A, double\* B, double\* C)

{

for (int i = 0; i < n; i += UNROLL \* 4)

for (int j = 0; j < n; j++) {

\_\_m256d c[4];

for (int x = 0; x < UNROLL; x++)

c[x] = \_mm256\_load\_pd(C + i + x \* 4 + j \* n);

for (int k = 0; k < n; k++)

{

\_\_m256d b = \_mm256\_broadcast\_sd(B + k + j \* n);

for (int x = 0; x < UNROLL; x++)

c[x] = \_mm256\_add\_pd(c[x],

\_mm256\_mul\_pd(\_mm256\_load\_pd(A + n \* k + x \* 4 + i), b));

}

for (int x = 0; x < UNROLL; x++)

\_mm256\_store\_pd(C + i + x \* 4 + j \* n, c[x]);

}

}

int main()

{

double\* A = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double\* B = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double\* C1 = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double\* C2 = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

for (int i = 0; i < ARRAY\_SIZE \* ARRAY\_SIZE; i++)

{

A[i] = rand() % 100;

B[i] = rand() % 100;

}

clock\_t t;

t = clock();

dgemm(ARRAY\_SIZE, A, B, C1);

t = clock() - t;

double elapsed\_time = ((double)t) / CLOCKS\_PER\_SEC;

printf("Unoptimized DGEMM code took %.6f seconds to execute.\n", elapsed\_time);

return 0;

}

**4-AVX+UNROLL+BLOCKING OPTIMIZATION CODE**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <cuda.h>

#include <device\_functions.h>

#include <cuda\_runtime\_api.h>

#include <immintrin.h>

#include <intrin.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define ARRAY\_SIZE 256

#define UNROLL (4)

#define BLOCKSIZE 32

void do\_block(int n, int si, int sj, int sk,

double\* A, double\* B, double\* C)

{

for (int i = si; i < si + BLOCKSIZE; i += UNROLL \* 4)

for (int j = sj; j < sj + BLOCKSIZE; j++) {

\_\_m256d c[4];

for (int x = 0; x < UNROLL; x++)

c[x] = \_mm256\_load\_pd(C + i + x \* 4 + j \* n);

/\* c[x] = C[i][j] \*/

for (int k = sk; k < sk + BLOCKSIZE; k++)

{

\_\_m256d b = \_mm256\_broadcast\_sd(B + k + j \* n);

/\* b = B[k][j] \*/

for (int x = 0; x < UNROLL; x++)

c[x] = \_mm256\_add\_pd(c[x], /\* c[x]+=A[i][k]\*b \*/

\_mm256\_mul\_pd(\_mm256\_load\_pd(A + n \* k + x \* 4 + i), b));

}

for (int x = 0; x < UNROLL; x++)

\_mm256\_store\_pd(C + i + x \* 4 + j \* n, c[x]);

/\* C[i][j] = c[x] \*/

}

}

void dgemm(int n, double\* A, double\* B, double\* C)

{

for (int sj = 0; sj < n; sj += BLOCKSIZE)

for (int si = 0; si < n; si += BLOCKSIZE)

for (int sk = 0; sk < n; sk += BLOCKSIZE)

do\_block(n, si, sj, sk, A, B, C);

}

int main()

{

double\* A = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double\* B = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double\* C1 = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

double\* C2 = (double\*)calloc(ARRAY\_SIZE \* ARRAY\_SIZE, sizeof(double));

for (int i = 0; i < ARRAY\_SIZE \* ARRAY\_SIZE; i++)

{

A[i] = rand() % 100;

B[i] = rand() % 100;

}

clock\_t t;

t = clock();

dgemm(ARRAY\_SIZE, A, B, C1);

t = clock() - t;

double elapsed\_time = ((double)t) / CLOCKS\_PER\_SEC;

printf("Unoptimized DGEMM code took %.6f seconds to execute.\n", elapsed\_time);

return 0;

}

**5-GPU OPTIMIZATION SOURCE CODE**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <cuda.h>

#include <device\_functions.h>

#include <cuda\_runtime\_api.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define BLOCK\_SIZE 16

#define ARRAY\_SIZE 256

int main(int argc, char const \*argv[])

{

// allocate in host

int \*h\_a, \*h\_b, \*h\_c1, \*h\_c2;

cudaMallocHost((void \*\*)&h\_a, sizeof(int)\*ARRAY\_SIZE\*ARRAY\_SIZE);

cudaMallocHost((void \*\*)&h\_b, sizeof(int)\*ARRAY\_SIZE\*ARRAY\_SIZE);

cudaMallocHost((void \*\*)&h\_c1, sizeof(int)\*ARRAY\_SIZE\*ARRAY\_SIZE);

cudaMallocHost((void \*\*)&h\_c2, sizeof(int)\*ARRAY\_SIZE\*ARRAY\_SIZE);

for (int i = 0; i < ARRAY\_SIZE; ++i) {

for (int j = 0; j < ARRAY\_SIZE; ++j) {

h\_a[i \* ARRAY\_SIZE + j] = rand() % 100;

h\_b[i \* ARRAY\_SIZE + j] = rand() % 100;

}

}

// Allocate in device

int \*d\_a, \*d\_b, \*d\_c;

cudaMalloc((void \*\*)&d\_a, sizeof(int)\*ARRAY\_SIZE\*ARRAY\_SIZE);

cudaMalloc((void \*\*)&d\_b, sizeof(int)\*ARRAY\_SIZE\*ARRAY\_SIZE);

cudaMalloc((void \*\*)&d\_c, sizeof(int)\*ARRAY\_SIZE\*ARRAY\_SIZE);

// copy from host to device memory

cudaMemcpy(d\_a, h\_a, sizeof(int)\*ARRAY\_SIZE\*ARRAY\_SIZE, cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, sizeof(int)\*ARRAY\_SIZE\*ARRAY\_SIZE, cudaMemcpyHostToDevice);

int grid\_rows = BLOCK\_SIZE\*BLOCK\_SIZE;

int grid\_cols = ARRAY\_SIZE / grid\_rows;

dim3 dimGrid(grid\_cols, grid\_cols,1);

dim3 dimBlock(grid\_rows, grid\_rows,1);

float elapsed\_time\_gpu;

//description to calculate time

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

// start time of GPU

cudaEventRecord(start, 0);

matrix\_mult\_gpu <<<dimGrid, dimBlock >> > (d\_a, d\_b, d\_c, ARRAY\_SIZE);

// copy from device to host

cudaMemcpy(h\_c1, d\_c, sizeof(int)\*ARRAY\_SIZE\*ARRAY\_SIZE, cudaMemcpyDeviceToHost);

cudaThreadSynchronize();

// stop time

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

// compute time elapse on GPU computing

cudaEventElapsedTime(&elapsed\_time\_gpu, start, stop);

printf("Time elapsed on matrix multiplician %d on GPU: %.1f s.\n\n", ARRAY\_SIZE, elapsed\_time\_gpu);

// start the CPU version

clock\_t t;

t = clock();

matrix\_mult\_cpu(h\_a, h\_b, h\_c2, ARRAY\_SIZE);

t = clock() - t;

double elapsed\_time = ((double)t) / CLOCKS\_PER\_SEC;

printf("Time elapsed on matrix multiplication %d on CPU: %.1f s.\n\n", ARRAY\_SIZE, elapsed\_time);

// free memory

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

cudaFreeHost(h\_a);

cudaFreeHost(h\_b);

cudaFreeHost(h\_c1);

cudaFreeHost(h\_c2);

return 0;

}

\_\_global\_\_ void matrix\_mult\_gpu(int \*a, int \*b, int \*c, int n)

{

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

int sum = 0;

if (col < n && row < n)

{

for (int i = 0; i < n; i++)

{

sum += a[row \* n + i] \* b[i \* n + col];

}

c[row \* n + col] = sum;

}

}

void matrix\_mult\_cpu(int \*h\_a, int \*h\_b, int \*h\_result, int n)

{

int i, j, k;

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

h\_result[i\*n + j] = 0;

for (k = 0; k < n; k++)

{

h\_result[i\*n + j] += h\_a[k + i \* n] \* h\_b[k\*n + j];

}

}

}

}

**RESULTS AND DISCUSSION**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 256 | 512 | 768 | 1024 | 2048 |
| UNOPTIMIZED | 0.195 S | 1,849 S | 5.798 S | 33,932 S | 290.717 S |
| AVX | 0.062 S | 0.468 S | 1.753 S | 6.854 S | 77.548 S |
| AVX+UNROLL | 0.061 S | 0.441 S | 1.585 S | 5.219 S | 50.7230 S |
| AVX+UNRL+BLCKD | 0.056 S | 0.544 S | 1.661 S | 3.167 S | 22.687 S |
| GPU | 0.2 S | 0.8 S | 1.6 S | 2.8 S | 21.576 S |

In this study , we’ve tried to show performances of dgemm’s codes between each other.The results show that there is huge differences exceptionally unoptimized dgemm code.However , we demonstrate the performance affect of subword parallelism by running same code with avx,avx+unroll.By starting unoptimized version matrix multiplication.

We take a look to how will the performance improve ?. So after unoptimized code which take longer time than other we run five floating point instructions AVX . The highest performance is measured with avx+unroll+blocked version by a long way.

**CONCLUSION**

In this project, we examined in detail the parallel implementation of the double-precision general matrix-matrix multiplication (DGEMM) ,five floating instructions and GPU .We run the proper cache block sizes for AVX.We present the proper loops to be parallelized.

Moreover ,the project shows the performance increase DGEMM for matrices from unoptimized to AVX and then to AVX with unrolling. Unrolling+blocked nearly doubles performance, going from 4.858 to 6.817 seconds. Optimizations for subword parallelism and instruction level parallelism result in an overall speed 10.7 the unoptimized DGEMM on table.

**REFERENCES**

<https://dl.acm.org/doi/pdf/10.1145/3176364.3176374>

<http://www.eecs.harvard.edu/cs146-246/lee96subword.pdf>

<https://pdfs.semanticscholar.org/85aa/c72505e5fe2fc935495b539e47254cc3c957.pdf>

Computer Organization and Design The Hardware/Software Interface Fifth Edition , David Patterson,John Hennessy