TEAM_09

Members : Abhishek Vanamala
Jahnavi Valisetty

Introduction

One of the important problems in linear algebra is matrix multiplication which engineers have been solving with computers since the early days. In this project, we are trying to optimize the code of matrix multiplication for the lonestar5 supercomputer.

About Lonestar 5 machine single core

One core has a clock rate of 2.6 GHz and can turbo to 3.5GHz, so it can issue 3.5 billion instructions per second. Lonestar 5's processors also have a 256-bit *vector width (AVX2 vector extension)*, meaning each instruction can operate on 4 double data elements(64 bits) at a time. Furthermore, the Lonestar 5 microarchitecture includes two *fused multiply-add* (FMA) instruction, which means 2 floating point operations can be performed in a single instruction. So, the theoretical peak of Lonestar 5's nodes is:

- 2.6 GHz * 4-element vector * 2 ops in an FMA * 2 (FMA's) = 41.6 GFlops/s

Optimizations tried and their results

1) Naive code with ijk loop implementation

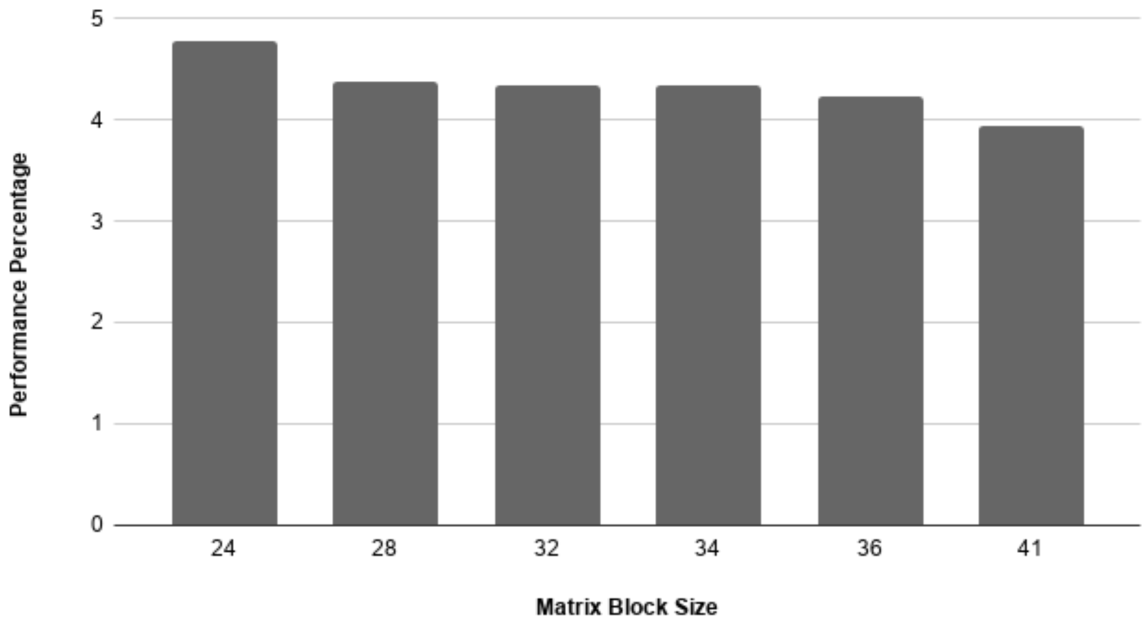
We ran the code provided in the assignment without any improvement.
Maximum percentage: 3.70

2) Naive code with ijk loop implementation where loop variables were defined outside the loop

Loop variable i, j and k and temp variable cij were initialized outside the loop.
Maximum percentage: 3.76
Reason: We can see a slight improvement here because when the loop variables are initialized outside the loop then there is no need to make new variables everytime inside the loop and the program can re-use the previous variables which will save some time.

3) Blocked implementation with different block size

We tried different block sizes to figure out what could be the optimal size.
Educated Guess: Level 1 cache has 32 KB dedicated data cache and 32 KB dedicated instruction cache.
Data cache size = 32 KB = 32768 B
Number of float values which can fit = 32768/8 = 4096
// Double size = 8B
Number of float values in one matrix = 4096/3 = 1365
// 3 Matrices A, B and C
Matrix Size= sqrt(1365) ~ 36

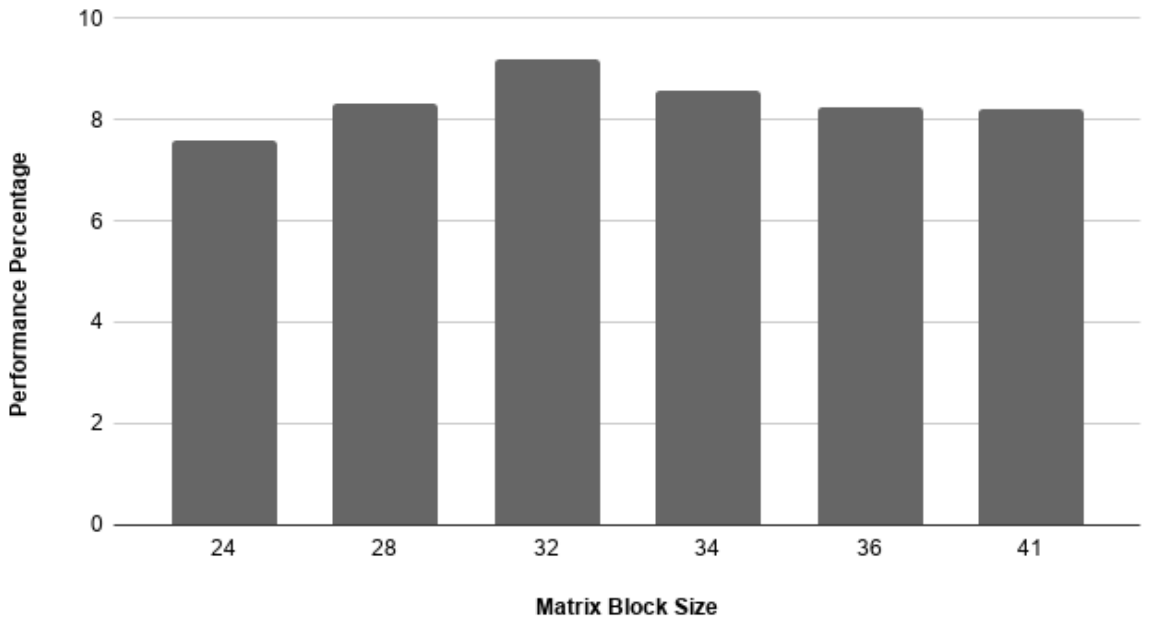


Maximum percentage: 4.78
Reason: As we can see from the educated guess that optimum number should be 36 but we observe from the graph data that for the matrix size 32, we are getting high performance values. The reason could be, except for matrices, cache has to hold temporary variables like i, j, k etc. Hence, if we will try to fit matrices values then whenever processor will ask for temporary variable values, cache miss will be there which will decrease the performance.

4) Loop unrolling with 6 instructions:

We tried different number of multiplication operations at a time inside innermost loop and here is the result: -

Maximum Percentage: 9.43
Reason: As processor supports fused multiply add operations, it can perform multiple operations in one instructions. Here from observation, we found out that doing 6 operations at a time is good.



5) Loop interchange

We tried loop interchange on block size 32 which has max percentage of 9.43 to see if it improves the performance

Maximum percentage: 10.91 (from 9.43)

Reason: As we can see, the performance increased from 8.09 to 9.91. In loop interchange we are using spatial locality to improve the percentage. Here matrices A, B and C are accessed in stride 1 fashion.

6) Loop interchange with loop unrolling

On the above code, we tried the loop unrolling as well to see if it can surpass our previous value which was 13.20 (Block size 32 and loop unrolling only).

Maximum percentage: 8.69

Code snippet :

```
C1[i1+j1*lda] += A1[i1+k1*lda]*B1[k1+j1*lda];
C1[i1+(j1+1)*lda] += A1[i1+k1*lda] * B1[k1+(j1+1)*lda];
C1[i1+(j1+2)*lda] += A1[i1+k1*lda] * B1[k1+(j1+2)*lda];
C1[i1+(j1+3)*lda] += A1[i1+k1*lda] * B1[k1+(j1+3)*lda];
```

Reason : Here we observe that, instead of performance improvement, performance decreased. As we can see, everytime C=C+A*B happens, processor has to fetch value of C and then store it in C also. Storing the value becomes overhead and the performance dipped.

7) Simple unrolled version 0f AVX2 intrinsic:

With Blocking, Loop unrolling and Interchanging we achieved the above performance.
In order to further optimise the performance of the computations we tried to use AVX2

Vector Intrinsic because it performs Linear Transformation of a matrix into column vector so that it can utilize SIMD instructions in the processor.

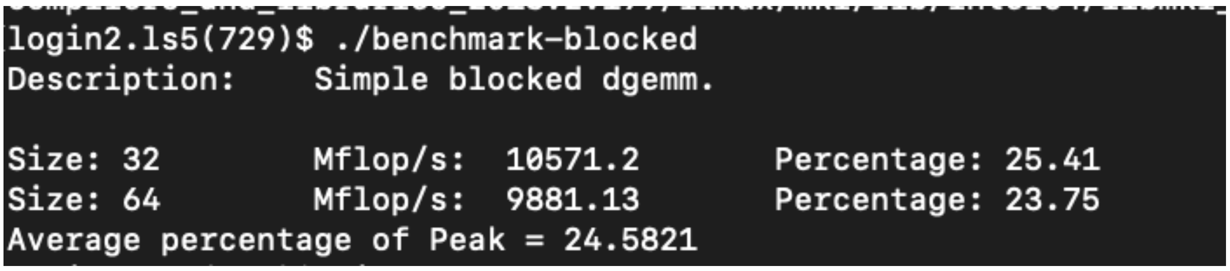
Reason :

- More optimised code.
- Easier to read and is more concise.
- Fewer lines of code which generally means fewer bugs.

Code snippet :

```
static void do_block_avx_ijk(int lda, int M, int N, int K, double* A, double* B, double* C)
{
 for (int i = 0; i<M;i+=4){
 for(j=0;j<N;++j){
 __m256d VC= _mm256_load_pd(C+i+j*lda);
 for(int k=0;k<K;++k){
  __m256d VA = _mm256_load_pd(C+i+k*lda);
  __m256d VB = _mm256_broadcast_sd(B+k+j*lda);
  VC = _mm256_add_pd(VC, _mm256_mul_pd(VA, VB));}}}
}
```

Result:



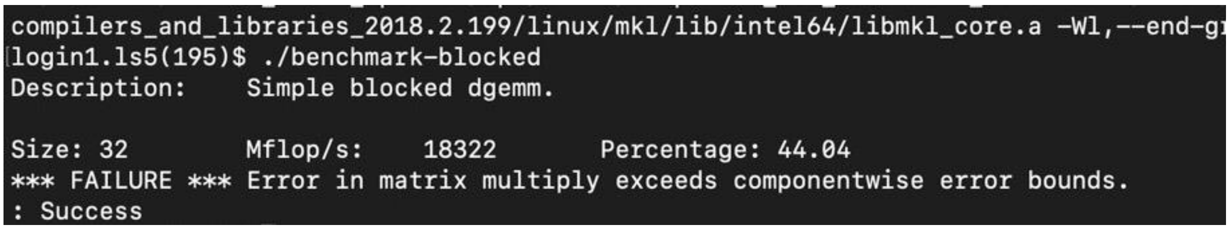
```
login2.ls5(729)$ ./benchmark-blocked
Description:    Simple blocked dgemm.

Size: 32        Mflop/s:  10571.2       Percentage: 25.41
Size: 64        Mflop/s:  9881.13       Percentage: 23.75
Average percentage of Peak = 24.5821
```

8) Loop Interchange version 0f AVX_2 vectorisation:

Code snippet :

```
static void do_block_avx_ikj(int lda, int M, int N, int K, double* A, double* B, double* C)
{
for(int i = 0; i < M; i+=4){
for(int k = 0; k< K;++k){
 __m256d VA = _mm256_load_pd(A+i+k*lda);
 for(int j = 0; j<N;++j){
 __m256d VC = _mm256_load_pd(C+i+j*lda);
 __m256d VB = _mm256_broadcast_sd(B+k+j*lda);
VC = _mm256_add_pd(VC, _mm256_mul_pd(VA, VB));
_mm256_storeu_pd(C+i+j*lda, VC);}
}
}
```

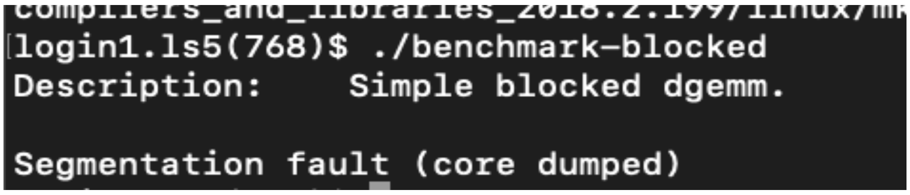
Result:



```
compilers_and_libraries_2018.2.199/linux/mkl/lib/intel64/libmkl_core.a -Wl,--end-g
[login1.ls5(195)$ ./benchmark-blocked
Description:    Simple blocked dgemm.

Size: 32        Mflop/s:    18322       Percentage: 44.04
*** FAILURE *** Error in matrix multiply exceeds componentwise error bounds.
: Success
```

We were able to achieve a peak percentage of 44.04 with loop interchange and vector intrinsic. But the problem we faced is this percentage is achieved for matrices whose sizes are multiples of 32.

Error:



```
compilers_and_libraries_2018.2.199/linux/mk
[login1.ls5(768)$ ./benchmark-blocked
Description:    Simple blocked dgemm.

Segmentation fault (core dumped)
```

Reason : Core Dump/Segmentation fault is a specific kind of error caused by accessing memory that "does not belong to you."

- When a piece of code tries to do read and write operation in a read only location in memory or freed block of memory, it is known as core dump.
- It is an error indicating memory corruption.

To overcome this specific error we tried to use padding to the matrices to directly copy matrix which is not aligned and copy value into old matrix. And proceed with loop unrolling and vectorisation. But zero padding a matrix didn't make much sense as it requires n*n operations.