ENSC 894

LAB 2 REPORT

Team 5 : Amrutha Suresh, Mohammed Shuhad

Configuration

CPU type : Intel Core™ i9-10900 10-core CPU (20 hyper-threads)

Level 1 cache size	32 KB private cache for each core
Level 2 cache size	1 MB private cache for each core
Level 3 cache size	19.25 MB shared cache shared by all 10 cores

Performance speedup under parallelization (10 threads only) and why

- Parallelization is done at j loop to maximize the spatial and temporal locality of references.
- The following pragma is applied before the two j loops since this offers the best speedup for 4096 * 4096 matrices.

  #pragma omp parallel for shared(i) schedule(guided)

- The schedule is chosen as guided since it offers the least run time. The below table shows the run time for 2048 * 2048 matrix.

Schedule	Run time
static	5.570062236
dynamic	3.885811614
guided	3.800373964

- The average run time and speedup is calculated below.

2048 * 2048			
	Sequential Run time	j - parallelized	i - parallelized
	34.11303807	4.360219538	4.668602231
	34.97096829	4.360219538	4.126140981
	34.06520759	4.360219538	3.294203046
Average	34.38307132	4.360219538	4.029648753
Speedup	NA	7.885628468	8.532523162
4096 * 4096			
	Sequential Run time	j - parallelized	i - parallelized
	872.9083182	112.1796422	86.25351922
	876.7516127	110.8395555	100.2724625
	838.3237407	112.4973551	202.3521137
Average	862.6612239	111.8388509	129.6260318
Speedup	NA	7.713430679	6.654999863

Speedup = Sequential execution time / Parallel execution time

Screenshot of output log

```
asa280@ensc-mmc-16:~/lab2_new$ ./mm
kernel execution: 112.179642193
sum of C array = 27789682688.000000
asa280@ensc-mmc-16:~/lab2_new$ ./mm
kernel execution: 110.839555537
sum of C array = 27789682688.000000
asa280@ensc-mmc-16:~/lab2_new$ ./mm
kernel execution: 112.497355078
sum of C array = 27789682688.000000
```

Why Speedup Occur

The OpenMP pragma '*parallel for*' is used to convert the proceeding for loop to run in a threaded environment based on the number of threads set.

Here, the number of threads is set to 10 since the cpu is 10 core.

The pragma parallel for automatically considers the loop variable as a private variable. The variable i is declared as shared to avoid any run time calculation issues.

The OpenMP clause schedule is used to distribute iterations effectively to different threads and to maximize the efficiency. The guided option is specified since this minimizes run time due to the fact that this is a combination of dynamic (threads are assigned dynamically depending on availability) and the chunk size is chosen based on the number of iterations remaining.

Hence the combination of pragma parallel for and schedule clause causes the speedup.

-

Performance speedup under vectorization

Vectorisation is done automatically by the compiler when the O3 flag is used. The same runtime can be achieved with the O2 flag, by telling explicitly which loop to vectorise . "#pragma omp simd" added before the j loop results in improved performance compared to the serial execution.

Execution time & Speedup

	Sequential Run time	Vectorised Run Time
	872.9083182	177.9590562
	876.7516127	178.2260404
	838.3237407	176.6490929
Average	862.6612239	177.6113965
Speedup	NA	**4.857015039**

Code change

```
for (i = 0; i < NI; i++)
{
  for (j = 0; j < NJ; j++)
  {
    C[i*NJ+j] *= beta;
  }
  #pragma omp simd
  for (j = 0; j < NJ; j++)
  {
    for (k = 0; k < NK; ++k)
    {
      C[i*NJ+j] += alpha * A[i*NK+k] * B[k*NJ+j];
    }
  }
}
```

Why Speedup Occur

`C[i*NJ+j] += alpha * A[i*NK+k] * B[k*NJ+j];`

This statement comprises 3 Arithmetic operations : 2 multiplication and 1 addition. When vectorisation is applied a total of 4 arithmetic operations from 4 iterations of j loops is done together. The steps would be as follow :

- 4 Multiplication : 1 multiplication each from 4 iterations of j
- 4 Multiplication : 1 multiplication each from 4 iterations of j
- 4 Addition : 1 addition each from 4 iterations of j

This is the reason why a speedup of 4 (approx.) happens.

There is no performance increase when the inner k loop is vectorised because the addition can only be done after the multiplication is complete. This data dependency prevents vectorisation of that loop.

Performance speedup under different tiling strategies and sizes, and why

Strategy 1 : Basic Tiling with OpenMP pragma "*parallel for*"

Calculation of tile size

L1 cache = 32 KB

To build a machine to run matrix multiply at 1/2 peak arithmetic speed of the machine, where M_{fast} = Size of fast memory (cache) and b = block size

Mfast >= $3b^2$

32000 >= $3b^2$

b <= 103 bytes

According to Theorem (Hong & Kung, 1981), number of words moved between fast and slow memory = Ω (n^3 / $(Mfast)^{1/2}$) = Ω (n^3 / $(Mfast)^{1/2}$)

Tile size calculation

$m \leq 52.2$

m = 32, would give the best result

```
int tileSize = 32;
omp_set_num_threads(10);
#pragma omp parallel for
  for (i = 0; i < NI; i++)
  {
    for (j = 0; j < NJ; j++)
    {
      C[i*NJ+j] *= beta;
    }
    for (j = 0; j < NJ; j+= tileSize)
    {
      for (k = 0; k < NK; k += tileSize)
      {
        for(int jj = j ; jj < j + tileSize ; jj++)
        {
          for(int kk = k ; kk < k + tileSize ; kk++)
          {
            C[i*NJ+jj] += alpha * A[i*NK+kk] * B[kk*NJ+jj];
          }
        }
      }
    }
  }
```

Tile Size vs Execution Time

Tile size (m x m)	Execution Time 1	Execution Time 2	Execution Time 3	Average
512	31.06105906	28.56844925	32.26722483	30.63224438
256	28.76633162	29.17618683	28.22082404	28.72111416
128	29.24032485	30.07914932	28.72070958	29.34672791
64	25.84478785	25.13690798	26.92615865	25.96928482
32	25.82318169	24.94306878	25.96069716	25.57564921
16	35.38576506	31.32361318	32.06404051	32.92447292

Thus the above table complies with the calculated tile size.

Comparison with Other Strategies

Strategy	Run time	Speedup
Basic Tiling (without parallelization)	147.8851335	5.83331944
Basic Tiling with parallelization	25.57564921	33.72978793
Tiling (tile size=16) + matrix transpose	46.81447078	18.42723435
Tiling (tile size=4) + matrix transpose	39.93039127	21.60412649
Tiling + reduction	154.0408179	5.600211915

Based on the run times, we have chosen basic tiling with OpenMP pragma as the best performance strategy. This must be due to the optimized level of memory accesses based on the cache size and matrix sizes. In case of strategies like transpose and reduction, more operations are present which may have contributed to the higher run time.

Tiling with matrix transpose

For the matrix multiplication of A * B, we take the matrix transpose of B, such that the statement in the innermost loop access consecutive locations instead of locations which are away NJ positions (where NJ is the number of columns in B). The code snippet is attached in Appendix. This logic is added in addition to tiling.

Tiling with reduction

For the matrix multiplication of A * B, the intermediate sum is stored in a scalar variable and written back to output array at end of computation. This logic is added in addition to tiling.

APPENDIX

Code Snippet - Tiling + Matrix Transpose

```
static
void kernel_gemm(float C[NI*NJ], float A[NI*NK], float B[NK*NJ], float
alpha, float beta)
{
  int i, j, k, jj,kk,block=4;
  omp_set_num_threads(10);
 #pragma parallel for schedule(guided)
  for(i=0;i<NK;i++)
      for(j=0;j<NJ-(NJ-i);j++)
            std::swap(B[i*NJ+j],B[j*NJ+i]);

// => Form C := alpha*A*B + beta*C,
//A is NIxNK
//B is NKxNJ
//C is NIxNJ
  #pragma parallel omp
  for (i = 0; i < NI; i++) {
for (j = 0; j < NJ; j+=block) {
      for (jj = j; jj < j+block; jj++)
            C[i*NJ+jj] *= beta;
}
for (j = 0; j < NJ ; j+=block) {
      for (k = 0; k < NK ; k+=block) {
            for (jj = j; jj < j+block; jj++)
                  for (kk=k; kk<k+block; kk++)
                  C[i*NJ+jj] += alpha * A[i*NK+kk] * B[jj*NJ+kk];
      }
}
  }
}
```

Code Snippet - Tiling + Reduction

```
static
void kernel_gemm(float C[NI*NJ], float A[NI*NK], float B[NK*NJ], float
alpha, float beta)
{
  int i, j, k, jj,kk,block=16;
```

```
  omp_set_num_threads(10);
// => Form C := alpha*A*B + beta*C,
//A is NIxNK
//B is NKxNJ
//C is NIxNJ
 #pragma parallel for
  for (i = 0; i < NI; i++) {
for (j = 0; j < NJ; j+=block) {
      for (jj = j; jj < std::min(j+block,NJ); jj++)
            C[i*NJ +jj] *= beta;
}
for (j = 0; j < NJ ; j+=block) {
      for (k = 0; k < NK ; k+=block) {
            for (jj = j; jj<std::min(j+block,NJ); jj++){
                  float sum = C[i*NJ + jj];
                  #pragma parallel for shared(i,j) reduction(+:sum)
                  for (kk=k; kk<std::min(k+block,NK); kk++) {
                  sum = sum + alpha * A[i*NK + kk] * B[kk*NJ+ jj];
                  }
                  C[i*NJ + jj] = sum;
            }
      }
}
  }
}
```