HPC 4MA 2021/2022

Members

PHAM Tuan Kiet

VO Van Nghia

Date

12 Dec, 2021

Contents

Chapter 1

OpenMP

1.1 Optimization techniques

1.1.1 Naive dot

We first mention here the original `naive_dot` function. This function serves as an anchor (or base case) for performance comparision as well as for making sure we have the right result when using other techniques.

```
  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < K; k++) C[i + ldc * j] += A[i + lda * k] * B[k + ldb *
     j];
```

Below is the output of `naive_dot` for `M = 1`, `K = 2` and `N = 2`:

```
## ( 1.00  1.50 )
##
## ( 1.00  1.50 )
## ( 1.50  2.00 )
##
## Frobenius Norm   = 5.550901
## Total time naive = 0.000001
## Gflops           = 0.013333
##
## ( 3.25  4.50 )
```

As

$$\begin{pmatrix} 1 & 1,5 \end{pmatrix} \begin{pmatrix} 1 & 1,5 \\ 1,5 & 2 \end{pmatrix} = \begin{pmatrix} 3,25 & 4,5 \end{pmatrix}$$

The result of this function is correct. We move on to the next technique.

1.1.2 Spatial locality

Spatial locality refers to the following scenario: if a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In order to take advantages of this property, we notice that:

- In memory, `A`, `B`, `C` are stored in contiguous memory block.
- When using the index order `i`, `j`, `k`, we access `B` consecutively (as we access `B` by `B[k + ldb * j]`), but not `A` and `C`.
- Data from `A`, `B`, `C` are loaded in a memory block consisting of severals consecutive elements to cache. Thus, we could make use of spatial locality when reading data continously.

From 3 points above, we decide to switch the index order to `k`, `j`, `i`. Now we see that both reading and writing operations on `C` are in cache, this brings us a critical gain in performance. In addition, reading operations on `A` are in cache too but those on `B` are not.

```
for (k = 0; k < K; k++)
  for (j = 0; j < N; j++)
    for (i = 0; i < M; i++) C[i + ldc * j] += A[i + lda * k] * B[k + ldb *
   j];
```

For comparision, we have a table below with `M = 4`, `K = 8` and `N = 4`.

Table 1.1: naive vs saxpy when M, N, K is small

algo	time	norm	gflops
naive	0.000002	3.461352	0.111304
saxpy	0.000001	3.461352	0.182857

We have the frobenius norm of both techniques are 3,461352 which indicate we have the right computation result. In addition, calculating time is already significantly small (≈ 0 second in both methods) and the difference between these two can therefore be ommited.

However, if we set `M = 2048`, `K = 2048` and `N = 2048`, there will be a huge performance gain as in the table shown below. In addition, from now, for an easier comparision between results, we will consider the default value of `M`, `K` and `N` is `M = 2048`, `K = 2048` and `N = 2048` if not explicitly mentioned.

Table 1.2: naive vs saxpy when M, N, K is big

algo	time	norm	gflops
naive	149.30573	2.323362	0.115065
saxpy	62.04563	2.323362	0.276891

Here, the `naive_dot` function is approximately 2,41 times slower than the `saxpy_dot` function.

1.1.3 OpenMP parallelization

In this section, we will analyse the main technique of this chapter: `OpenMP`. First, we show how we enable it on each function. We add a directive above each loop we want to parallelize whose general form is as below:

- Variables inside `private` are tied to one specific thread (each thread has their own copies of those variables).
- `SCHEDULE_HPC` is replaced by the schedule[1] we want.
- `NUM_THREADS_HPC` is corresponding to the number of threads to use for `parallel` regions.

```
#pragma omp parallel for schedule(SCHEDULE_HPC) default(shared) private(i, j
    ) \
    num_threads(NUM_THREADS_HPC)
```

In addition, inside `norm` function, we add a reduction clause `reduction(+ : norm)` as we want to sum up every `norm` from each thread to one final `norm` and taking square of that final sum. Finally, we have to add `#pragma omp atomic` above each line that updating the result matrix (`C`). It is because that matrix is shared among threads, `atomic` makes sure that there is only one `+=` operation (which is essentially reading and writing) on one specific pair of indices at a given time. Note that `norm` does not need atomicity thank to `reduction`.

Here, we show a comparision between with and without `OpenMP`. Default `OpenMP` options will be `SCHEDULE_HPC = static` and `NUM_THREADS_HPC = 4`.

Table 1.3: naive vs saxpy with OpenMP

algo	time	norm	gflops	omp
naive	45.08355	2.323362	0.381067	x
naive	149.30573	2.323362	0.115065	
saxpy	15.29221	2.323362	1.123440	x
saxpy	62.04563	2.323362	0.276891	

Thank to `OpenMP`, naive approach is faster than 0,3 times while the `saxpy_dot` took less 0,25 times than before. Both approachs performance are significantly improved.

1.1.4 Cache blocking (Tiled)

The main idea of the cache blocking technique (or tiled) is breaking the whole matrices into smaller sub-matrices so the data needed for one multiplication operation could fit into the cache, therefore leads to a much faster calculation. Furthermore, if we enable `OpenMP`, the computation would be even faster as each sub-matrice is processed by a separate thread. However, if we set `BLOCK` size too small, the benefit of dividing

1. https://www.openmp.org/spec-html/5.0/openmpse49.html#x288-20520006.1

matrix is overshadowed by the additional loops and operations. Meanwhile, a too large `BLOCK` size leads to an overfitting (data for one operation can not be fitted into the cache), and therefore a slower operation. The principal source code is shown below:

```
for (k = 0; k < K; k += BLOCK)
  for (j = 0; j < N; j += BLOCK)
    for (i = 0; i < M; i += BLOCK)
      for (kk = 0; kk < BLOCK; kk++)
        for (jj = 0; jj < BLOCK; jj++)
          for (ii = 0; ii < BLOCK; ii++)
            C[(ii + i) + ldc * (jj + j)] +=
                A[(ii + i) + lda * (kk + k)] * B[(kk + k) + ldb * (jj + j)
  ];
```

The above code will work only if `M`, `N` and `K` are divisible by `BLOCK`. A more generic version could be found in full source-code.

We have a table comparision between all techniques we are dicussing so far below. Also, we set the default size of `BLOCK = 4`.

Table 1.4: naive vs saxpy vs tiled

algo	time	norm	gflops	omp
naive	45.08355	2.323362	0.381067	x
naive	149.30573	2.323362	0.115065	
saxpy	15.29221	2.323362	1.123440	x
saxpy	62.04563	2.323362	0.276891	
tiled	17.01662	2.323362	1.009594	x
tiled	17.14018	2.323362	1.002316	

In the table above, cache blocking technique is already fast enough. However, `OpenMP` does not help speeding it up as the default `BLOCK` size is not optimized in this case.