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.