Multicore-GPU Programming 2024 Spring
Assignment1: Simple Filter on 1D Array

2019143073 Soeun Uhm

1. Background

1.1 Objective

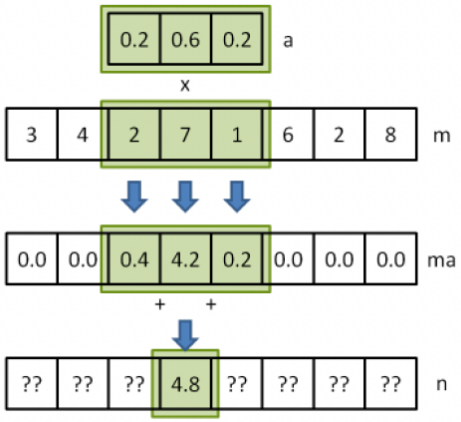

Figure 1. Filtering on 1D array

The goal of this assignment is to implement filtering on 1D array. Actual calculation is done like Figure 1. Filtering on 1D array can be done in serial. However, since this operation can be highly parallelizable, our objective is to implement multi-thread filtering on 1D array.

1.2 Backgrounds

1.2.1 Multi-Thread

Before diving into actual implementation, we must understand the concept of what multi-thread is.

In Von-Neumann Model, program is written serially, and processor executes it serially. However, some part of the program can be executed in parallel since there is no or less dependency exists between them. Executing program in parallel can be done in many ways and multi-thread programming is one of the ways to parallelize the program.

Thread can be summarized in “light-weight process”. Each thread has their own separated context while sharing the same memory space. The key concept of multi-threading is to execute different part of program in parallel. With the aid of hardware and software support, programmers can explicitly separate their workload to each thread, and they are executed concurrently in different cores.

1.2.2 Multi-Core Processor Architecture

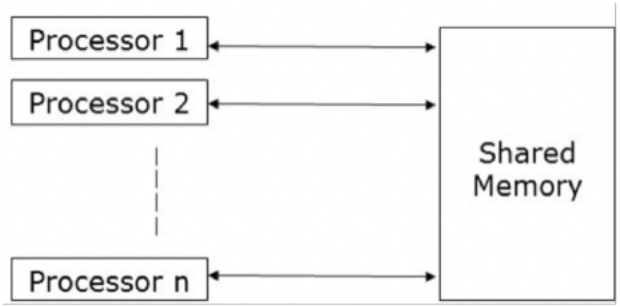

Figure2. Shared memory model

Knowing about underlying hardware architecture helps programmer to optimize their program executes better in multi-thread. One of the most important concepts in multi-core processor is “memory system”.

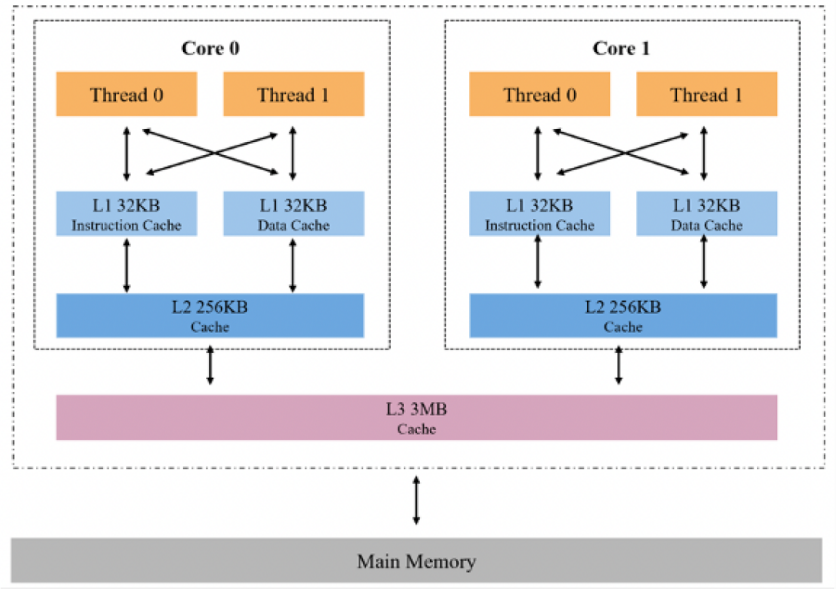

Figure3. Memory system of multicore processors

In shared memory model, multiple cores share the memory. However, it doesn’t mean that the whole memory is shared by all the cores. Rather than that, to reduce the memory access latency, each processor owns their local cache. Therefore, the actual system looks like figure3.

With the aid of this kind of memory system, each core of processor can effectively reduce the memory access latency when accessing the data located in local cache. Also to support effective communication between threads, there is higher level of cache shared by multiple cores. With the presence of local cache, programmers can optimize the code using thread local data that doesn’t need to be shared. It leads to minimize the overhead of synchronization between the threads and reduce the data transfer between CPU and memory.

1.3 Skills

1.3.1 Loop Unrolling

Loop is useful for repeated iteration. However, since loop is consisted of not only loop body, but also test and increment, it leads to overhead. To mitigate this problem, loop unrolling technique is used. Loop unrolling technique is literally unrolling the independent loop for certain extends. This technique can be applied only when the loop is vectorizable.

The degree of unrolling is determined by the architectural characteristics. It is closely related to number of registers. If unrolling exceeds the degree that register can hold, temporal data will be stored in memory, and it leads to increase of execution time.

1.3.2 Removing Memory Aliasing

Assume that there is code update and store intermediate value to memory every loop iteration. Intermediate value doesn't necessarily be stored to memory, and if so, write memory access for every loop leads to massive performance degradation.

To address this problem, we can introduce local variable for storing the intermediate value. Introducing local variable for intermediate value eliminates unnecessary write memory access, and we can expect performance improvement.

2. Implementation

2.1 Workload Characteristics

```
for ( int i =0;i<N -4; i++) {
    for ( int j =0;j< FILTER_SIZE ;j ++) {
        array_out_serial [i] += array_in [i+j] * k[j];
    }
}
```

Above code represents the cpp style code for serial filtering on 1D array. At a glance, we can observe that it consists of nested loop, and updating output at every loop.

```
165d: 48 8d 04 bd 00 00 00 lea 0x0 (,%rdi ,4) ,%rax
1664: 00
1665: c4 c1 7a 10 04 bc vmovss (% r12 ,% rdi ,4) ,% xmm0
166b: 8d 4a 03 lea 0x3 (% rdx ) ,% ecx
166e: 49 8d 34 00 lea (%r8 ,% rax ,1) ,% rsi
1672: 48 63 f9 movslq %ecx ,% rdi
1675: c4 e2 51 a9 06 vfmadd213ss (% rsi ) ,%xmm5 ,%xmm0
167a: c4 c2 59 b9 44 04 04 vfmadd231ss 0x4 (% r12 ,% rax,1) ,%xmm4 ,% xmm0
1681: c4 c2 61 b9 44 04 08 vfmadd231ss 0x8 (% r12 ,% rax,1) ,%xmm3 ,% xmm0
1688: c4 c2 69 b9 44 04 0c vfmadd231ss 0xc (% r12 ,% rax,1) ,%xmm2 ,% xmm0
168f: c4 c2 71 b9 44 04 10 vfmadd231ss 0 x10 (% r12 ,% rax,1) ,%xmm1 ,% xmm0
1696: c5 fa 11 06 vmovss %xmm0 ,(% rsi)
```

For understanding the workload characteristic in more depth, the above code shows the assembly code of iteration part of Serial Filtering. As we can see, the actual iteration is done by Vector Operation which is called AVX in intel. Using SIMD processor, it can perform Filtering operation more efficiently.

Back to our scope, instruction VMOVSS instruction move the value from source to destination which matches to LOAD instruction. “VFMADD231SS xmm1, xmm2, xmm3” means multiply xmm2, xmm3 and add xmm1, finally store it to xmm1. This means that we have 2 memory read and 1 memory write for each calculation.

With the direct approach, Serial Filtering perform 2(N-1)(FILTERSIZE) memory read and (N-1)(FILTERSIZE) memory write, which is quite memory extensive workloads. Therefore, we need to focus on the memory access. Paying attention to this, let’s dive into implementing Parallel Filtering on 1D array.

2.2 Direct Implementation

Let’s start with direct implementation and optimize it. The code below shows how parallel filtering is implemented. It represents thread creation, start and finalization. Lambda filter array() is function executed by each thread.

```
std :: thread threads [NT ];
int size = (N -4) / NT;
int remain = (N -4) % NT;
auto filter_array = []( float * array_in , const float * k, int base ,
int top , int FILTER_SIZE , float * array_out_parallel ){
    for ( int i= base ; i<top ; i ++) {
        for ( int j = 0; j < FILTER_SIZE ; j ++)
            array_out_parallel[i] = local ;
    }
};

for ( int i =0; i<NT; i ++) {
    threads [i] = std :: thread ( filter_array , array_in , k, i*size , (
i +1) *size , FILTER_SIZE , array_out_parallel );
}

for ( int i = size *NT; i< N -4; i ++) {
    for ( int j = 0; j< FILTER_SIZE ; j ++)
        array_out_parallel[i] += array_in[i+j]*k[j];
    }
for ( int i =0; i<NT;i ++) {
    threads[i]. join ()
}
```

One of the most important things to consider in parallel programming is to assign the workload to each thread “well”. In our example, since the workload is very simple, it’s easy to divide and assign them to each thread.

First for loop is starting the threads. I divide workloads uniformly in terms of number of elements that it compute and assigned to each thread. Second for loop is for computing the remaining elements. To produce correct result in the case N-4 is not divided by NT, the main thread performs filtering for remaining elements right after the thread starts. Third for loop is waiting threads until all the threads to be completed using thread.join().

Number of concurrent threads also matters. One can think that the more threads, the more performance increase. However, it isn’t in real case. Basically utilizing multi-threads have certain degree of overhead for creating, starting and finalizing. Also, if workload is not assigned properly fit with hardware characteristics, it can leads to overhead such as false sharing. Therefore, adjusting the number of threads according to its workload is important. In our case, default number of threads is 32, but I will adjust them later on.

2.3With Optimization

2.3.1 Removing Memory Aliasing

```
auto filter_array = []( float * array_in , const float * k, int
base , int top , int FILTER_SIZE , float * array_out_parallel ){
    float local = 0;
    for ( int i= base ; i<top ; i ++) {
        for ( int j = 0; j < FILTER_SIZE ; j ++)
            local += array_in[i+j] * k[j];
        array_out_parallel[i] = local ;
        local = 0;
    }
};
```

The above code removes memory aliasing compared to the direct approach one. By introducing local variable ”local”, processor doesn’t need to access memory for updating intermediate value in nested loop every time. Therefore, we can look forward for reducing memory access latency and eventually performance improvements.

2.3.2 Loop Unrolling + Removing Memory Aliasing

```
auto filter_array = []( float* array_in , const float* k, int
base, int top, int FILTER_SIZE, float* array_out_parallel ){
  float local = 0;
  for ( int i= base ; i<top ; i ++) {
    for ( int j = 0; j < FILTER_SIZE -4; j +=5) {
      local += array_in[i+j] *k[j];
      local += array_in[i+j +1] * k[j +1];
      local += array_in[i+j +2] * k[j +2];
      local += array_in[i+j +3] * k[j +3];
      local += array_in[i+j +4] * k[j +4];
    }
    for ( int j = 5 * (FILTER_SIZE/5) ; j < FILTER_SIZE ; j++){
        local += array_in[i+j] * k[j];
    }
    array_out_parallel[i] = local;
    local = 0;
    }
};
```

The above code applies loop unrolling technique. The degree of loop unrolling is 5. We need additional for loop to calculate the remaining elements. By applying loop unrolling explicitly, we can expect that the loop overhead will decrease and eventually get performance increase. However, in our case, the performance enhancement will not that dramatic. Analysis about this will be conducted on [3. Evaluation] Part.

3. Evaluation

3.1 Direct Implementation

	1	2	3	4	5	Avg	Med
Serial	2.312	2.361	2.550	2.172	2.511	2.381	2.371
Parallel	3.417	3.463	3.740	3.387	3.396	3.481	3.440
Speed Up	0.677	0.682	0.682	0.641	0.740	0.684	0.682

Table1. Evaluation on Direct Implementation

Table1 represents evaluation on Direct Implementation. We can observe that execution time increased compared serial filtering on 1D array. The cause of this can vary.

Firstly, create, start and finalizing the multiple threads derive overhead. Since private memory space per threads should be allocated and each core executing the thread need to bring the data from main memory to local, overhead exists.

Second, memory aliasing leads to disastrous ”False Sharing”, and eventually it cause extensive memory access latency. Since I didn’t consider any hardware specific characteristics, there should be shared cache block between the cores. For every loop, each core updates the element, and to maintain the coherence between caches, cache block that have been modified transfer between cores. It cause false sharing, and leads to extreme performance degradation. To mitigate this, lets move on to optimized code.

3.2 Removing Memory Aliasing

	1	2	3	4	5	Avg	Med
Serial	2.097	2.232	2.242	2.272	2.242	2.217	2.237
Parallel	0.689	0.735	0.731	0.680	0.682	0.703	0.696
Speed Up	3.045	3.037	3.068	3.343	3.286	3.156	3.112

Table2. Evaluation on Removing Memory Aliasing

After removing memory aliasing, we can observe that execution times got 3 times faster compared to serial filtering. Although the performance enhancement satisfies the given spec, can we do better than this? Let’s move on to further optimization technique.

3.3 Loop Unrolling + Removing Memory Aliasing

	1	2	3	4	5	Avg	Med
Serial	2.420	2.255	2.082	2.080	2.093	2.186	2.140
Parallel	0.743	0.695	0.731	0.725	0.753	0.730	0.730
Speed Up	3.255	3.244	2.848	2.869	2.781	2.999	2.934

Table3. Evaluation on Loop Unrolling + Removing Memory Aliasing

The above table shows the execution time and Speed Up compared to serial filtering. We can observe that the performance increase is less than single ”Removing Memory Aliasing” optimization.

In our case, since the nested loop is repeated only for 5 times, loop unrolling doesn’t amortize the loop unrolling overhead. If FILTER SIZE increases, then we can get performance improvement. Also, I think g++ compiler may optimize the original loop automatically.

3.4 Number of Threads

	1	2	3	4	5	Avg	Med
Serial	2.216	2.713	2.237	2.386	2.336	2.378	2.357
Parallel	0.670	0.683	0.529	0.631	0.645	0.632	0.638
Speed Up	3.308	3.971	4.226	3.783	3.620	3.782	3.782

Table4. Evaluation on NT=16 case

The table shown above represents execution time and Speed Up compared to serial filtering, when number of threads is 16 and only removing memory aliasing technique is applied. We can observe that performance has been slightly improved compared to the case NT=32.

This is because the workload distributed to each thread is too small in the case of NT=32. To fully amortize the multi-threads overhead, workload assigned to each thread should be sufficiently large. It means that, if the size of input array gets bigger, NT=32 case will be get better compared to the case of NT=16.

4. Reference

1) *Bryant, R. E., & O’Hallaron, D. R. (2015). Computer Systems: A Programmer’s Perspective (3rd ed.). Pearson. Chapter 5: Optimizing Program Performance.*