**Matrix Block Multiplication**

**Introduction**

The matrix block multiplication is known to be faster the naïve matrix multiplication, **C = C + A \* B**. In this approach, we can divide the n×n matrix to b×b blocks, so there will be NXN total blocks where b=n/N. This approach is faster because it can take advantage of the hierarchical memory management system. Our goal is to achieve peak performance according to ATLAS’s performance based on dPerfSumm.txt file. We will also use other techniques like register blocking, loop unrolling to get peak performance.

**The Algorithm**

The matrix block multiplication algorithm is described below:

Consider A, B, C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

for i = 1 to N

for j = 1 to N

{read block C(i,j) into fast memory}

for k = 1 to N

{read block A(i,k) into fast memory}

{read block B(k,j) into fast memory}

C(i,j) = C(i,j) + A(i,k) \* B(k,j) {do a matrix multiply on bxb blocks}

{write block C(i,j) back to slow memory}

**Design and implementation**

In the implementation, the main challenge is to load the **A, B, C** matrices to faster memory. We used memaligned and dynamically allocated pointer to double to load b×b sub matirces into faster memory. If the block size is within reasonable dimension then complier will try to load that into relevant cache memory. For b×b sub matrix multiplication we use our previous fastest combination IKJ. The matrix size, n is in range { 64, 128, 256, 512, 1024, 2048 }. We will try to find the block size, b ranging from 2 to n/2 or n for which we can achieve the peak performance.

**Results**

After installing Atlas on aprun node, we found its performance at the following location:

$ head -5 ATLAS\_DIR/BUILD\_DIR/bin/INSTALL\_LOG/dPerfSumm.txt

Clock\_rate=2600 Mhz

% clock MFLOP ROUTINE/PROBLEM

======= ========= ===============

98.7 2566.6 Chosen kgemm

97.9 2544.2 Generated kgemm

Without any optimization, the peak performance is achieved for block size = 128 for N = 256, 512, 1024 and 2048. The maximum peak is 33%, but previous naïve peak was around 25%

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Block\n** | **64** | **128** | **256** | **512** | **1024** | **2048** |
| **2** | 0.300454 | 0.29596 | 0.298676 | 0.267782 | 0.230758 | 0.209993 |
| **4** | 0.689781 | 0.675091 | 0.675181 | 0.649737 | 0.607908 | 0.542324 |
| **8** | 1.372674 | 1.340051 | 1.360913 | 1.391333 | 1.334803 | 1.229328 |
| **16** | 1.56961 | 1.894485 | 1.922354 | 1.914949 | 1.904525 | 1.886925 |
| **32** | **2.309898** | 2.519288 | 2.549315 | 2.537056 | 2.514906 | 2.500049 |
| **64** |  | **2.931054** | 3.032482 | 2.999185 | 2.993597 | 2.995418 |
| **128** |  |  | **3.199961** | **3.25736** | **3.298042** | **3.309583** |
| **256** |  |  |  | 2.771207 | 2.792233 | 2.821632 |
| **512** |  |  |  |  | 1.575882 | 1.594635 |
| **1024** |  |  |  |  |  | 1.115389 |
| **2048** |  |  |  |  |  |  |

Next looping unrolling is tested to unroll 2, 4, and 8 for each input size using the block size for which the highest Gflop can be achieved. The results are given below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Loop\n** | **64** | **128** | **256** | **512** | **1024** | **2048** |
| **2** | **2.**484772 | **3.**16748 | 3.32815 | **3.**50164 | 3.707563 | 3.724146 |
| **4** | 2.51947 | 3.498148 | **3.**538074 | 3.833477 | **3.**941247 | **3.**867375 |
| **8** | 2.664133 | 3.688863 | 3.912091 | 4.173092 | 4.268849 | 4.241475 |

After doing loop unrolling, register blocking and compiler optimization, the peak performance is achieved for block size = 256 and for N = 512, 1024 and 2048. When we found that peak performance is achieved for mainly for block size = 64, 128 and 256; specific code is written for those to utilize array with attribute aligned. Instead of using double for loop to copy matrices to faster memory, memcpy C function is used to get better performance which involves in a single for loop.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Block\n** | **64** | **128** | **256** | **512** | **1024** | **2048** |  |
| **2** | 0.234813 | 0.25992 | 0.239796 | 0.223313 | 0.157726 | 0.11318 |  |
| **4** | 0.563708 | 0.56612 | 0.620919 | 0.566658 | 0.448263 | 0.362771 |  |
| **8** | 1.202967 | 1.15102 | 1.314566 | 1.221717 | 1.023949 | 0.973981 |  |
| **16** | 1.934057 | 2.221516 | 2.466353 | 2.449403 | 2.218825 | 2.153776 |  |
| **32** | 2.03991 | 3.063773 | 3.396175 | 3.494391 | 3.482738 | 3.495053 |  |
| **64** | **2.701503** | **3.927704** | 3.484377 | 3.580526 | 3.627009 | 3.657738 |  |
| **128** |  | 3.521959 | **4.218623** | 4.280858 | 4.271507 | 4.253142 |  |
| **256** |  |  | 3.916228 | **4.335754** | **4.356922** | **4.376038** |  |
| **512** |  |  |  | 3.634878 | 3.706425 | 3.767107 |  |
| **1024** |  |  |  |  | 1.697695 | 1.723933 |  |
| **2048** |  |  |  |  |  | 1.342293 |  |

From the above table, the below chart is generated:

In this chart, X axis represents block size, Y axis represents Gflop, and each bar represents various input size, n. For example, when n=128, block size from 2 to 128 is tried, and peak Gflop is achieved when block size, b=64

**Conclusion**

According to our result, we found that peak performance is achieved for block size, b=256. Using other techniques like loop unrolling, register blocking we achieve close to 43.7% of BigRed2’s peak performance (10Gflop).