1. Q = 01000010010110100011110101110111 (54.560024261474609375)

After storing Q in register where 3 and 9 bits are faulty, its value becomes

Q` = 01100010110110100011110101110111 (2012909608413612212224)

Result error : Q – Q` = -2012909608413612212169.439975738525390625

As 3 and 9 bits are faulty, we’ll try to 3 right shifts and 9 right shifts on faulty Q and then store.

Q\_W = 11001000110010110100011110101110 (-416317.4375)

When we read we shift left by 3 bits and then read

Q” = 01000110010110100011110101110110 (13967.365234375)

Result error: Q – Q” = -13912.805210113525390625

Now, we shift 9 bits right and left and see the error

Q\_W = 10011011001000010010110100011110 (-1.3332186E-22)

Q” = 01000010010110100011110100110110 (54.55977630615234375)

Result error Q-Q” = 0.000247955322265625

So, if we shift 9 bits to right while writing and do 9 bits left shifting while reading we can minimize error to 0.000247955322265625.

With static scheduling it takes 200 units of time for each thread and 110 units of wait time because of thread one to finish the parallel region i.e total 200 + 110 = 310.

Computation time table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Time stamp | Thread 0 | Thread 7 | Others | Total | Remaining |
| At 0 | 0 | 0 | 0 | 0 | 2000 |
| At 50 | 0 | 0 | 8\*50 (25chunks) =400 | 400 | 1600 |
| At 110 | 0 | 60 (30 chunks) | 8\*60 (30 chunks) = 540 | 400 + 540= 940 | 1060 |
| At 216 | 106 (58 chunks) | 106 (58 chunks) | 8\*106 (58 chunks) | 940 +1060 = 2000 | 0 |

After 216 units of time, all threads complete the parallel region.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Time stamp | Thread 0 | Thread 7 | Others | Total | Remaining |
| At 0 | 0 | 0 | 0 | 0 | 2000 |
| At 50 | 0 | 0 | 8\*50 (5 chunks) | 400 | 1600 |
| At 110 | 0 | 60 (6 chunks) | 8\*60 (6 chunks) | 400+540=940 | 1060 |
| At 220 | 110 (11 chunks) | 110 (11 chunks) | 8 \* 110(11 chunks) | 940 + 1100 = 2040 | 0 |

After 110 units of time, 1060 iterations are pending, to divide these among 10 threads with chunks size 10, we need 11\*10 = 110 units of time. All threads come out of barrier after 220 units of time

1. We can parallelize the multiple for loops under one parallel section by keeping

“ #prgama omp master”

for sequential section.

1. Case 1:

var1,var2 variables are shared

By default in global variables are shared, so we just need to add directive

*#pragma omp parallel*

Result: All threads can access var1, var2 but output may not be guaranteed without synchronization

Example output for 2 threads:

Region 1: var1=1, var2=2

Region 1: var1=1, var2=2

After region 1: var1=3, var2=4

Case 2:

var1, var2 are private

To make them private we change directive to

*#pragma omp parallel private(var1,var2)*

Result: As each thread gets its own private copy for those variables without initialization we get garbage values in the region.

Example output for 2 threads:

Region 1: var1=0, var2=0

Region 1: var1=1968174676, var2=32606

After region 1: var1=1, var2=2

Case 3:

Var1, var2 are firstprivate

To make them firstprivate variables we change directive to

*#pragma omp parallel firstprivate(var1,var2)*

Result: Making them firstprivate initialized thread copies with values of master thread, but still after parallel region is over original variables becomes undefined like in case2.

Example output for 2 threads:

Region 1: var1=1, var2=2

Region 1: var1=1, var2=2

After region 1: var1=1, var2=2

1. We can divide the work in for loop with “for” construct and compute the global sum with ‘reduce’ operation

Line in TODO can be replaced with

#pragma omp parallel for reduction(+:sum)

App1: If branch divergence is high then CPU is suitable to run that application because threads synchronization can be bottleneck in GPU -> **CPU**

App2: For I/O intensive apps CPU is suitable than GPU -> **CPU**

App3: As kernel code is 95% of total, we can move all of this into GPU to parallelize. -> **GPU**

App4: As 100th iteration is ran on GPU and data transfer is so much, we can leverage temporal locality and make use of already loaded cache data for next iteration, so we can execute next iteration in GPU. -> **GPU**

App5: For multi-media intensive app we can use GPU to parallelize on different blocks if memory footprint can fit in GPU. -> **GPU**

App6: For BLAS-2 (O(n\*2)) programs we can use CPU -> **CPU**

App7: If data transfer overhead is more than computation then we can run the app in CPU itself -> **CPU**

App8: We can run 0,1,2,3,4 threads on CPU and 5,6,7 threads on GPU to maintain nearly proportional load balancing

CPU load : 0+1+2+3+4 = 10

GPU load : 2.5 + 3 + 3.5 = 9

1. Weights after encoding : [9, -6, -6 , 9, 9, 9, -6, 9]

Input : [4, -7, 9, 10, 13, 15, 3, 1]

D4

For factorized multiplication, we first add corresponding input values for 9, -6

9 -> 4 + 10 + 13 + 15 + 1 = 43 (4 additions)

-6 -> -7 + 9 + 3 = 5 (2 additions)

Then we multiply and add -> 9\*43 + -6\*5 (requires 2 multiplications, 1 addition)

So, total we need 2 multiplications, 7 additions

1. By tiling only first loop we can write like below :

for ( row=0; row<R; row+=row\_tile)

for ( col =0; col<C; col++)

for ( to=0; to<M; to++)

for ( ti =0; ti <N; ti++)

**for(trow=row; trow < min(R, trow+row\_tile); trow++)**

for ( i =0; i<K; i++)

for ( j =0; j<K; j++)

Output\_fmaps [to] [trow] [col] += Weights [to] [ti] [i] [j] \* Input\_fmaps[ti] [S\*trow+i] [S\*col+j]

After this we can re-use the row\_tile number of rows we fetch initially for all [C][M][N] iterations.

1. Here memory organization is such that A[z][y][x] is closer to A[z+1][y][x] than A[z][y+1][x] and A[z][y][x+1]

So we can reorder the given for loops like this to get benefit of spatial locality

for(j =0 to 99)

for(k=0 to 99)

for(i = 0 to 99)

sum = sum+ Array[i][j][k];

In above loops we are accessing A[i][j][k] and then A[i+1][j][k] which is aligned to memory organization.

1. Binarized matrices multiplication can be calculated using XNOR gates

XNOR table: Mulitplication:

Inp1 Inp2 Out Inp1 Inp2 Out

0 0 1 -1 -1 1

1 0 0 1 -1 -1

0 1 0 -1 1 -1

1 1 1 1 1 1

[1 1 0] \* 1 1 0 = [BCNT(XNOR(110, 100)), BCNT(XNOR(110, 111),

0 1 0 BCNT(XNOR(110,001))]

0 1 1

= [BCNT(101), BCNT(110), BCNT(000)]

= [1 1 -3]

Bit Count [BCNT] can be made fast by using look up table rather than computing for every product. Bit count is number of 1’s – number of 0’s in bit string.



|  |  |
| --- | --- |
| Variable | Location ? |
| x\_dim | Register (As thread local) |
| y\_dim | Register (As thread local) |
| iteration | Register (As thread local) |
| pqr | Register (As size fit into register) |
| ABC | Global (As these are created with cudaMalloc) |
| maxValue | Global (As its created outside kernel function with \_\_device\_\_directive) |

1. Loop unrolling is used to increase hardware parallelism, if code inside is independent of previous iteration then we can run multiple blocks of code in one iteration.

int x;

for (x = 0; x < 1000; x++) {

process(x);

}

To unroll above for loop to execute only 50 times, we have to repeat the code inside of it 20 times.

int x;

for (x = 0; x < 50; x+=20) {

process(x); process(x+1); process(x+2); process(x+3); process(x+4);

process(x+5); process(x+6); process(x+7); process(x+8); process(x+9);

process(x+10); process(x+11); process(x+12); process(x+13); process(x+14);

process(x+15); process(x+16); process(x+17); process(x+18); process(x+19);

}

1. As main function is always executed in CPU, qualifier for it is

\_\_host\_\_ int main(void)

random\_ints function is called by main and its not kernel function so it’ll be executed by CPU, so the correct qualifier for it is

\_\_host\_\_ void random\_ints(int\* x, int size)

addFunc2 function is kernel code which is executed by GPU, and its called from main (CPU), so the correct qualifier is

\_\_global\_\_ void addFunc2(int \*a, int \*b, int \*c)

addFunc1 function is called from kernel, so its executed and called only in GPU, so the correct qualifier is

\_\_device\_\_ void addFunc1(int \*a, int \*b, int \*c)

1. Original binary representation : 01000000101101100110011001100110

Decimal value: 5.7

1. Fetching left most 9 bits :

Binary : 01000000100000000000000000000000

Decimal : 4

1. Fetching left-most 12 bits:

Binary: 01000000101100000000000000000000

Decimal : 5.5

1. Fetching left-most 15 bits:

Binary : 01000000101101100000000000000000

Decimal : 5.6875

1. Fetching left-most 18 bits:

Binary : 01000000101101100100000000000000

Decimal : 5.6953125

1. Average pooling with 2\*2 grid and stride 2 will give below result:

|  |  |
| --- | --- |
| 50+52+59+1/4 | 7+11+19+10/4 |
| 57+2+3+4/4 | 42+28+21+35/4 |

=

|  |  |
| --- | --- |
| 40.5 | 11.75 |
| 16.5 | 31.5 |