

In this chapter, we introduce the programming model and optimization methods by using an example in matrix multiplication. The optimization covers computational methods, the communication between the CPU and the MIC, and the linkage between the CPU and the MIC.

Chapter Objectives.

- Learn how to implement matrix multiplication.
- Learn optimization methods of matrix multiplication on the MIC.
- Comprehend MIC optimization by specific examples.

9.1 Series Algorithm of Matrix Multiplication

For this example we have a matrix $C=A*B$, where A is an $M*K$ matrix, B is a $K*N$ matrix, and C is a $M*N$ matrix. The main function structure of matrix multiplication is shown in Fig. 9.1.

```
[pseudo code]
int main(void)
{
1. //allocate and initialize the matrix A, B, and C
   // read the matrix A and B
   ...
2. //execute C=A*B on MIC card
3. //export matrix C
   //release matrix A, B and C
   ...
   return 0;
}
```

Fig. 9.1 Main function structure of matrix multiplication

The sequential algorithm of matrix multiplication is shown in Algorithm MatrixMul_V1, which contains three levels of loops. The inner loop iterates by variable k , in which a row of matrix A is multiplied by a column from matrix B. Then the product is an element of matrix C. In the two outer-level loops, for loops over every element in matrix C, i loops over the rows, and j loops over the columns.

Algorithm MatrixMul_V1

```
[code snippet]
1   for(i=0;i<M;i++)
2   {
3       for(j=0;j<N;j++)
4       {
5           float sum = 0.0f;
6           for(k=0;k<K;k++)
7           {
8               sum += A[i*K + k] * B[k*N + j];
9           }
10          C[I * N + j] = sum;
11      }
12  }
```

In this example, the size of matrix A, B, and C is 4096*4096. The sequential program consumes 312.83s on the Intel Xeon 5675 3.07GHz platform. (This program can be marked as P_baseline.) The analysis result of VTune is shown in Fig. 9.2, from which we can see that most of the time is consumed in the instruction

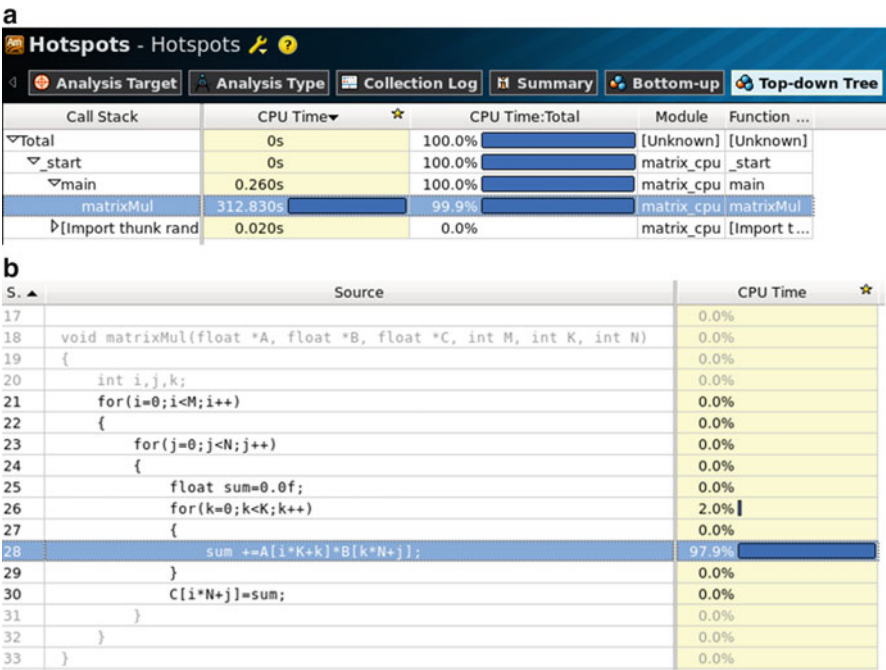


Fig. 9.2 Results of serial matrix multiplication in VTune

“sum += A[i*K+k] * B[k*N+j]”. This instruction is in the third level of the loops, in which there is no dependence in the loops except for the innermost level.

9.2 Multi-thread Matrix Multiplication Based on OpenMP

According to the sequential matrix multiplication algorithm, we could implement the parallel version based on OpenMP. From the sequential code we could see, there is not any dependency in the two outer loops. Therefore the two outer loops can be readily parallelized by OpenMP. (The number of loops in the outer level must be greater than the number of threads.) In the Algorithm MatrixMul_V2, the variable THREAD_NUM is the number of threads. For example, on the two-channel Xeon 5675 3.07GHz platform with 6 cores in each CPU, the THREAD_NUM could be set to 24 (with Hyper-Threading switched on). In this situation, the program consumes 170.83s and the speedup is 1.83. This program can be designated with P_OMP.

Algorithm MatrixMul_V2

[code snippet]

```

1  #pragma omp parallel for private(j, k) num_threads(THREAD_NUM)
2  for(i=0; i<M; i++)
3  {
4      for(j=0; j<N; j++)
5      {
6          float sum = 0.0f;
7          #pragma ivdep
8          for(k=0; k<K; k++)
9          {
10             sum += A[i*K + k] * B[k*widthN + j];
11          }
12             C[i*N + j] = sum;
13         }
14     }

```

9.3 Multi-thread Matrix Multiplication Based on MIC

9.3.1 Basic Version

After creating the OpenMP version, we can begin to run the program on MIC with offload mode. Shown in Algorithm MatrixMul_V3.1, the directive “#pragma offload target(mic)” shows that the data will be offloaded to MIC. “In()” and “out()” show the data transfer from the CPU to the MIC and the MIC to the CPU, respectively. “Length” is the length of data transmitted. The program is vectorized automatically by “#pragma”. This version of program consumes 174s on the KNC platform (60 cores, 1.0GHz, 240threads), and the speedup is 1.80. This program can be marked as P_MIC_base.

Algorithm MatrixMul_V3.1

[code snippet]

```

1  #pragma offload target(mic)\
2  in(i, j, k, M, K, N)\
3  in(A: length(M*K))\
4  in(B: length(K*N))\
5  out(C: length(M*N))\
6  {
7      #pragma omp parallel for private(j, k) num_threads(THREAD_NUM)
8      for(i=0; i<M; i++)
9      {
10         for(j=0; j<N; j++)
11         {
12             float sum = 0.0f;
13             #pragma ivdep
14             for(k=0; k<K; k++)
15             {
16                 sum += A[i*K + k] * B[k*widthN + j];
17             }
18             C[i*N + j] = sum;
19         }
20     }
21 }
```

9.3.2 Vectorization Optimization

In Algorithm MatrixMul_V3.1, although one can employ automatic vectorization, because of the sum operation in the inner loop and the discontinuity of the array B access, the result which comes out is not so good. Instead, a better result can be achieved by interchanging the orders of loops. As shown in Algorithm MatrixMul_V3.2, the array B and C could be accessed continuously in the modified program. In array A, only one element is accessed in the inner loop.

The modified sequential version without vectorization consumes 53.07s on the CPU platform, while after vectorization, it is 25.00s. This version could be marked as P_baseline_vec, which runs 12.24 times faster than P_baseline. After the same optimization, the OpenMP version of this program consumes 4.53s, which is marked as P_OMP_vec. It runs 5.52 times faster than P_baseline_vec. The MIC version consumes 3.43s after optimization, and runs 7.92 times faster than P_baseline_vec. This program can be designated as P_MIC_vec.

Algorithm MatrixMul_V3.2

[code snippet]

```

1  #pragma offload target(mic)\
2  in(i, j, k, M, K, N)\
3  in(A: length(M*K))\
4  in(B: length(K*N))\
5  out(C: length(M*N))\
6  {
7      #pragma omp parallel for private(j, k) num_threads(THREAD_NUM)
8      for(i=0; i<M; i++)
9      {
10         for(k=0; k<K; k++)
11         {
12             float temp = 0.0f;
13             #pragma ivdep
14             for(j=0; j<N; j++)
15             {
16                 C[i*N + j] = temp * B[k*N + j];
17             }
18         }
19     }
20 }
```

9.3.3 SIMD Instruction Optimization

In order to improve on the performance of MIC-version programs even more, SIMD instructions could be applied. The MIC version with SIMD instructions is shown in Algorithm MatrixMul_V3.3. Matrix multiplication consumes 2.00s when SIMD instructions applied. This is marked as P_MIC_simd, which runs 12.5 times faster than P_baseline_vec, and 71.5% faster than P_MIC_vec. Although some programs could be accelerated by SIMD instructions, however, this makes the program more difficult to read and to understand. So the SIMD instructions are optional.

Algorithm MatrixMul_V3.3

[code snippet]

```

1  #pragma offload target(mic)\
2  in(i, j, k, M, K, N)\
3  in(A: length(M*K))\
4  in(B: length(K*N))\
5  out(C: length(M*N))\
6  {
7      #pragma omp parallel for private(j, k) num_threads(THREAD_NUM)
8      for(i=0; i<M; i++)
9      {
10         #ifdef __MIC__
11         __m512 _A, _B, _C;
12         for(k=0; k<K; k++)
13         {
14             _A = _mm512_set_1to16_ps(A[i*K + k]);
15             for(j=0; j<N/16; j+=16)
16             {
17                 _B = _mm512_loadunpacklo_ps(_B, (void*)(&B[k*N + j]));
18                 _B = _mm512_loadunpackhi_ps(_B, (void*)(&B[k*N + j + 16]));
19                 _C = _mm512_loadunpacklo_ps(_B, (void*)(&C[i*N + j]));
20                 _C = _mm512_loadunpackhi_ps(_B, (void*)(&C[i*N + j + 16]));
21                 _C = _mm512_ad_ps(_C, _mm512_mul_ps(_A, _B));
22                 _mm512_packstorelo_ps((void*)(&C[i*N + j]), _C);
23                 _mm512_packstorehi_ps((void*)(&C[i*N + j + 16]), _C);
24             }
25         }
26         #endif
27     }
28 }

```

The performance of the whole optimizations is shown in Fig. 9.3, and the speedup in Fig. 9.4.

9.3.4 Block Matrix Multiplication

For large matrix multiplication, we employ mostly block matrices, which benefits MIC optimization because:

1. Block matrix multiplication can make a better use of cache, increase the hit ratio, and then improve performance.
2. Block matrix multiplication can use the dual buffer and hide the communication between the CPU and the MIC by the MIC nocopy technique.

Program Version	Time (s)
P_baseline	312.83
P_OMP	170.83
P_MIC_base	174
P_baseline_vec	25
P_OMP_vec	4.53
P_MIC_vec	3.43
P_MIC_simd	2

Fig. 9.3 Performance of matrix multiplication

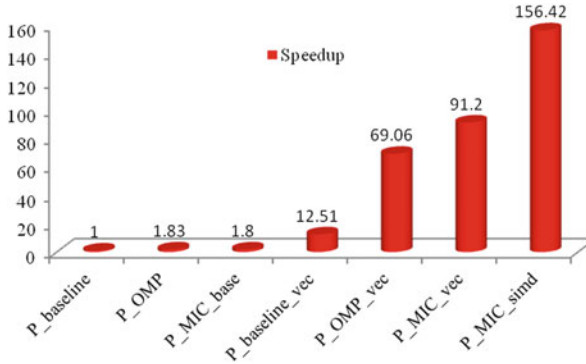


Fig. 9.4 Speedup of matrix multiplication

3. Because of the limited memory, block matrix multiplication can be applied in matrix multiplication of any scale.
4. Block matrix multiplication can create a good load balance between the CPU and the MIC by the means of allocating jobs dynamically for the CPU+MIC hybrid architecture.

We now introduce the optimized algorithm for large block matrix multiplication on the MIC.

9.3.4.1 Block Matrix Multiplication

Matrix multiplication can be denoted by $C_{m*n} = A_{m*k} * B_{k*n}$, which is shown in Fig. 9.5. There are three procedures in matrix multiplication:

Step 1: Partition the matrix in the direction i (for $i=0; i < M; i++$), which is shown in Fig. 9.6(a). The matrix A and C are partitioned by M_c .

Step 2: Based on Step 1, partition the matrix in the direction k (for $k=0; k < K; k++$), which is shown in Fig. 9.6(b), and the dimension of each submatrix is K_c .

Step 3: Based on Steps 1 and 2, partition the matrix in the direction j (for $j=0; j < N; j++$), which is shown in Fig. 9.6(c). The matrices B and C are partitioned by N_c .

The sequential algorithm of matrix multiplication is shown in the Algorithm MatrixMul_V4.1.

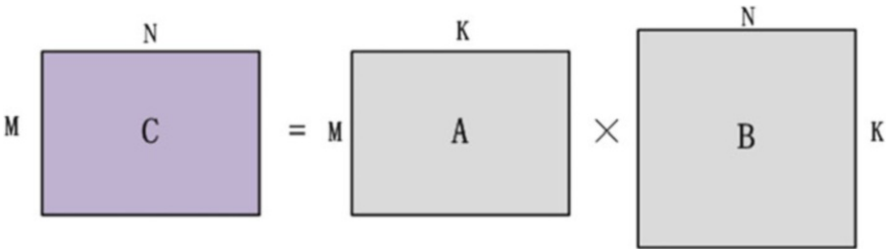


Fig. 9.5 Diagram of matrix multiplication

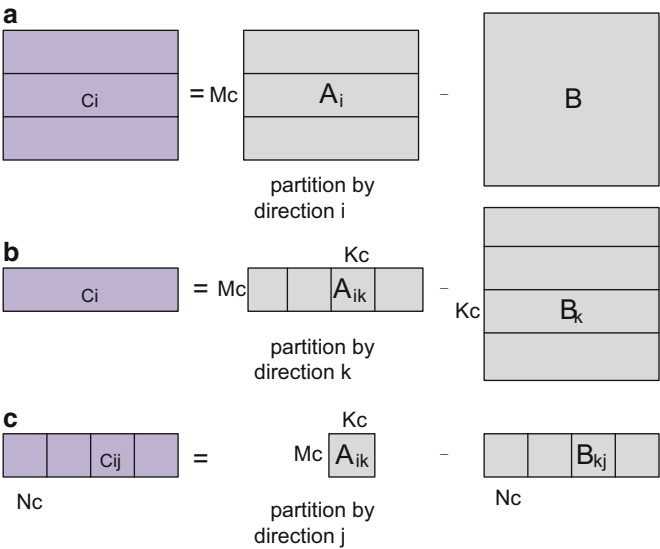


Fig. 9.6 Partition method of matrix multiplication

Algorithm MatrixMul_V4.1

[code snippet]

```

1  #define Mc 1024
2  #define Kc 1024
3  #define Nc 1024
4
5  void matrixMul(float *A, float *B, float *C, int M, int K, int N)
6  {
7      int i, j, k;
8      int ii, jj, kk;
9      int i_end, j_end, k_end;
10
11     i_end = Mc;
12     for(ii=0;ii<M;ii+=Mc)
13     {
14         if(Mc>M-ii)
15             i_end = M-ii;
16         k_end = Kc;
17         for(kk=0;kk<K;kk+=Kc)
18         {
19             If(Kc>K-kk)
20                 k_end = K-kk;
21             j_end = Nc;
22             for(jj=0;jj<N;jj+=Nc)
23             {
24                 if(Nc>N-jj)
25                     j_end = N-jj;
26
27                 for(i=ii;i<ii+i_end;i++)
28                 {
29                     for(j=jj;j<jj+j_end;j++)
30                     {
31                         float temp = 0;
32                         for(k=kk;k<kk+k_end;k++)
33                         {
34                             temp += A[i*K+k] * B[k*N + j];
35                         } //for(k=kk;k<kk+k_end;k++)
36                         C[i*N + j] += temp;
37                     } // for(j=jj;j<jj+j_end;j++)
38                 } // for(i=ii;i<ii+i_end;i++)
39             } // for(jj=0;jj<N;jj+=Nc)
40         } // for(kk=0;kk<K;kk+=Kc)
41     } // for(ii=0;ii<M;ii+=Mc)
42 }
```

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \\ C_{20} & C_{21} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \\ A_{20} & A_{21} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Fig. 9.7 Example of block matrix multiplication

Partition by direction i	Partition by direction k	Partition by direction j	Computation process
ii=0	kk=0	jj=0	$C_{00} += A_{00} * B_{00}$
		jj=1	$C_{01} += A_{00} * B_{01}$
	kk=1	jj=0	$C_{00} += A_{01} * B_{10}$
		jj=1	$C_{01} += A_{01} * B_{11}$
ii=1	kk=0	jj=0	$C_{10} += A_{10} * B_{00}$
		jj=1	$C_{11} += A_{10} * B_{01}$
	kk=1	jj=0	$C_{10} += A_{11} * B_{10}$
		jj=1	$C_{11} += A_{11} * B_{11}$
ii=2	kk=0	jj=0	$C_{20} += A_{20} * B_{00}$
		jj=1	$C_{21} += A_{20} * B_{01}$
	kk=1	jj=0	$C_{20} += A_{21} * B_{10}$
		jj=1	$C_{21} += A_{21} * B_{11}$

Fig. 9.8 Process of block matrix multiplication

The process of block matrix multiplication is shown below by some examples (Fig. 9.7).

The computation process of block matrix multiplication is shown in Fig. 9.8.

9.3.4.2 Block Matrix Multiplication Based on the MIC

For matrix multiplication on the MIC, the cache usage and performance can be greatly enhanced by partitioning. The MIC version of block matrix multiplication is shown in Algorithm MatrixMul_V4.2. To test the impact of block matrix on performance, the matrix is set to 16384*16384. The primary time elapsed is 301.37s without partitioning, while the partitioned version only consumes 131.19s, which is 2.3 times faster. The same algorithm consumes 206.31s on the 2-channel, 8-core Xeon 5675 with 24 threads, while the MIC version is 1.57 times faster than OpenMP.

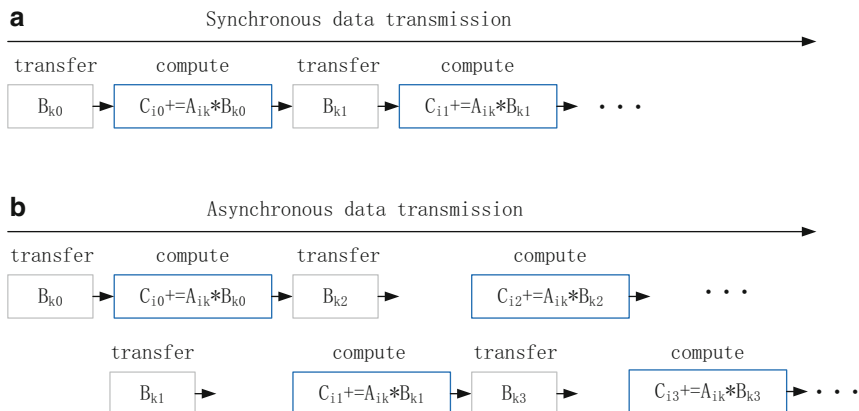


Fig. 9.9 Asynchronous block matrix multiplication

Algorithm MatrixMul_V4.2

9.3.4.3 Optimization of Asynchronous Matrix Multiplication

It is inefficient to transfer the data between PC memory and MIC memory by means of the PCI-E bus. Actually, the communication process between the CPU and the MIC can be hidden by using asynchronous computing, which is shown in the following example of matrix multiplication.

As is shown in the third step in Fig. 9.7), the blocks are transferred to MIC one at a time. This process on MIC could be shown in Fig. 9.9(a), and the asynchronous version is shown in Fig. 9.9(b), which introduces how to decrease the communication between the CPU and the MIC by asynchronization. The asynchronous version of block matrix multiplication is shown below in Algorithm MatrixMul_V4.3.

Algorithm MatrixMul_V4.3:

[code snippet]

```

1  #define Mc 1024
2  #define Kc 1024
3  #define Nc 1024
4
5  /*It should be declared globally when the pointer nocopy is used, and the keyword attribute should be
   added before that.*/
6  #pragma offload_attribute(push, target(mic))
7  float *Ac;
8  float *Bc0;
9  float *Bc1; /*Declare the double buffer space Bc0 and Bc1, which are used for asynchronous data
   transfer.*/
10 float *Cc;
11 #pragma offload_attribute(pop)
12
13 /*The functions called in offload must be defined by the keywords __attribute__ (( target (mic))) or
   __declspec( target (mic))*/
14 __attribute__ (( target (mic)))
15 void kernel(float *Ac, float *Bc, float *Cc, int i_end, int k_end, int j_end, int Kc, int Nc, int N, int ii, int jj, int
   THREAD_NUM)
16 {
17     int i, j, k;
18     #pragma omp parallel for private(i,j,k) num_threads(THREAD_NUM)
19     for(i=0; i<i_end; i++)
20     {
21         for(k=0; k<k_end; k++)
22         {
23             float temp = Ac[i*Kc +k];
24             #pragma ivdep
25             for(j=0; j<j_end; j++)
26             {
27                 Cc[(ii+i)*N +(jj+j)] += temp*Bc[k*Nc +j];
28             }//for(j=0; j<j_end; j++)
29         }//for(k=0; k<k_end; k++)
30     }//for(i=0; i<i_end; i++)
31 }
32
33 void matrixMul(float *A, float *B, float *C, int M, int K, int N, int THREAD_NUM)
34 {
35     int ii,jj,kk,jj0,jj1;
36     int i_end,j_end,k_end,j_end0,j_end1;
37
38     /*Allocate the partitioned space*/
39     Ac = (float *)malloc(sizeof(float)*Mc*Kc);

```

```

40     Bc0 = (float *)malloc(sizeof(float)*Kc*Nc);
41     Bc1 = (float *)malloc(sizeof(float)*Kc*Nc);
42     Cc = C;
43
44     /*Allocate space on MIC*/
45     #pragma offload target(mic:0) \
46         nocopy(Ac:length(Mc*Kc) alloc_if(1) free_if(0)) \
47         nocopy(Bc0:length(Kc*Nc) alloc_if(1) free_if(0)) \
48         nocopy(Bc1:length(Kc*Nc) alloc_if(1) free_if(0)) \
49         nocopy(Cc: length(M*N) alloc_if(1) free_if(0))
50     {
51     }
52
53     i_end=Mc;
54     for(ii=0;ii<M;ii+=Mc)
55     {
56         if(Mc>M-ii)
57             i_end=M-ii;
58         k_end=Kc;
59         for(kk=0;kk<K;kk+=Kc)
60         {
61             if(Kc>K-kk)
62                 k_end=K-kk;
63             for(i=0; i<i_end; i++)
64                 for(k=0;k<k_end; k++)
65                     Ac[i*Kc+k] = A[(ii+i)*K+(kk+k)];
66     #pragma offload target(mic:0) \
67         in(Ac:length(Mc*Kc) alloc_if(0) free_if(0)) //Transfer A(ii, kk)
68         {
69         }
70         j_end = Nc;
71         j_end0 = Nc;
72         j_end1 = Nc;
73         jj0=0;
74         if(Nc>N-jj0)
75             j_end0 = N-jj0;
76         for(k=0; k<k_end; k++)
77             for(j=0; j<j_end0; j++)
78                 Bc0[k*Nc+j] = B[(kk+k)*N+(jj0+j)];
79     #pragma offload_transfer target(mic:0) in(Bc0:length(Kc*Nc) alloc_if(0) free_if(0)) signal(Bc0) /*
        Asynchronous data transfer.*/
80         int js=0;
81         for(jj=0; jj<N; jj+=Nc, js++) //Partition B
82         {

```

```

83         if(js%2==0)
84         {
85             jj1 = jj+Nc;
86             if(jj1<N)
87             {
88                 if(Nc>N-jj1)
89                     j_end1=N-jj1;
90                 for(k=0; k<k_end; k++)
91                     for(j=0; j<j_end1; j++)
92                         Bc1[k*Nc+j] = B[(kk+k)*N+(jj1+j)];
93 #pragma offload_transfer target(mic:0) in(Bc1:length(Kc*Nc) alloc_if(0) free_if(0)) signal(Bc1) /*
Asynchronous data transfer.*/
94             }
95             if(Nc>N-jj)
96                 j_end = N-jj;
97 #pragma offload target(mic:0) \
98 in(i_end, k_end, j_end, Kc, N, Nc, ii, jj) \
99 nocopy(Ac,Bc0,Cc) wait(Bc0)
100         {
101             kernel(Ac, Bc0, Cc, i_end, k_end, j_end, Kc, N, ii, jj, THREAD_NUM); //Call the
kernel
102         }
103     }
104     else
105     {
106         jj0 = jj+Nc;
107         if(jj0<N)
108         {
109             if(Nc>N-jj0)
110                 j_end0=N-jj0;
111             for(k=0; k<k_end; k++)
112                 for(j=0; j<j_end0; j++)
113                     Bc0[k*Nc+j] = B[(kk+k)*N+(jj0+j)];
114 #pragma offload_transfer target(mic:0) in(Bc0:length(Kc*Nc) alloc_if(0) free_if(0)) signal(Bc0) /*
Asynchronous data transfer.*/
115         }
116         if(Nc>N-jj)
117             j_end = N-jj;
118 #pragma offload target(mic:0) \
119 in(i_end, k_end, j_end, Kc, N, Nc, ii, jj) \
120 nocopy(Ac,Bc1,Cc) wait(Bc1)
121     {
122         kernel(Ac, Bc1, Cc, i_end, k_end, j_end, Kc, N, ii, jj, THREAD_NUM); //Call the
kernel

```

```

123         }
124     }
125 }
126 }
127 }
128 /*Return results and release space on MIC */
129 #pragma offload target(mic:0) \
130     nocopy(Ac:length(Mc*Kc) alloc_if(0) free_if(1)) \
131     nocopy(Bc0:length(Kc*Nc) alloc_if(0) free_if(1)) \
132     nocopy(Bc1:length(Kc*Nc) alloc_if(0) free_if(1)) \
133     out(Cc:length(M*N) alloc_if(0) free_if(1))
134 {
135 }
136 }
```

Please note that 1) The core algorithm of block matrix multiplication is shown in lines 13–31. 2) The memory allocation on the MIC is shown in lines 44–51. 3) The asynchronous transfer is shown in lines 79–125.

- 3.1.) First the value of Bc0 (line 79) is transferred to prepare the data for asynchronization. The word `offload_transfer` is applied in asynchronization.
- 3.2.) The loop of partitioning matrix B is shown in line 81, in which the asynchronous transfer is applied.
- 3.3.) The procedure of asynchronous transfer is shown in lines 83–124:
 - 3.1.1) The “if” branch in lines 83–103 shows: The value Bc1 is transferred (line 93), which is used in core(line 101).
 - 3.2.2) The “else” branch in lines 104–124 shows: The value Bc0 is transferred line 114), which is used in core (line 122 4) The value of Cc is transferred back in lines 128–135, and the memory occupied on the MIC is now released.

9.3.4.4 Matrix Multiplication Based on CPU+MIC hybrid Computing

CPU and MIC are all based on x86 architecture, and the same optimization. So we can employ the paradigm of hybrid computing of CPU+MIC to greatly improve the computational performance on a CPU+MIC platform. Moreover, we can execute the same source code on both the CPU and the MIC. We then introduce below matrix multiplication based on two-way CPU + multi-MIC on a single node hybrid computing.

CPU+MIC hybrid computing can be achieved by MPI/OpenMP+offload mode. Here, matrix multiplication based on the single node CPU+MIC hybrid computing is achieved by MPI+offload mode, which is shown in Fig. 9.10. For this purpose, programmers can use the OpenMP+offload mode and multi-node version by themselves.

Matrix multiplication based on single-node CPU+MIC hybrid computing is shown in Algorithm MatrixMul_V4.3. All the CPU cores in the node can be

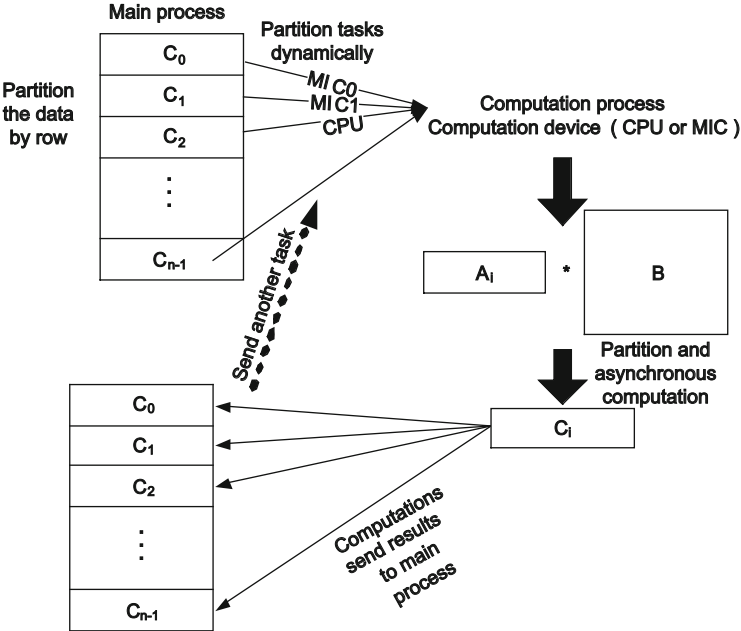


Fig. 9.10 Matrix multiplication based on single-node CPU+MIC hybrid computing

regarded as one device, and each MIC card could also be considered as a device in this node. If there are MIC_NUM MIC cards, the whole number of devices is MIC_NUM+1, and every device is controlled by an MPI process according to the process ID. The data allocation is achieved by the main process. In matrix multiplication, the data is allocated dynamically by dividing matrix C by rows into computing devices. Each amount of allocation is $M_c * N$. In other words, every device applies the data from main process and gets the results of M_c lines in matrix C. Then another set of data is applied until all the data have been multiplied.

Algorithm MatrixMul_V4.3

[code snippet]

```

1  #define Mc 1024
2  #define Kc 1024
3  #define Nc 1024
4
5  /* It should be declared globally when the pointer nocopy is used, and the keyword attribute should be
   added before that.*/
6  #pragma offload_attribute(push, target(mic))
7  float *Ak;
8  float *Bc0;
9  float *Bc1; /* Declare the double buffer space Bc0 and Bc1, which are used for asynchronous data
   transfer.*/
10 float *Cc;
11 #pragma offload_attribute(pop)
12
13 __attribute__((target(mic)))
14 void matrixMul (float *Ak, float *Bc, float *Cc, int i_end, int k_end, int j_end, int Kc, int Nc, int N, int jj, int
   THREAD_NUM)
15 {
16     int i, j, k;
17     #pragma omp parallel for private(i,j,k) num_threads(THREAD_NUM)
18     for(i=0; i<i_end; i++)
19     {
20         for(k=0; k<k_end; k++)
21         {
22             float temp = Ak[i*Kc + k];
23             #pragma ivdep
24             for(j=0; j<j_end; j++)
25             {
26                 Cc[i*N + (jj+j)] += temp*Bc[k*Nc + j];
27             } //for(j=0; j<j_end; j++)
28         } //for(k=0; k<k_end; k++)
29     } //for(i=0; i<i_end; i++)
30 }
31
32 int main( int argc, char *argv[] )
33 {
34     int THREAD_NUM_OMP = 1;
35     int THREAD_NUM_MIC = 1;
36     int M,K,N;
37     int myrank, root=0, totalrank;
38     MPI_Status status;
39     int MIC_NUM=2;
40     int deviceID=-1;

```

```

41     int nodeID=-1;
42     MPI_Init(&argc,&argv);
43     MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
44     MPI_Comm_size(MPI_COMM_WORLD, &totalrank);
45
46     /*The main process controls the data partitioning, and allocates tasks dynamically according to the row of
matrix C. The size of the matrix, which is allocated to the device each time, is Mc*N*/
47     if(myrank==root)
48     {
49         Initialize M, K, N, MIC_NUM, THREAD_NUM_OMP and THREAD_NUM_MIC;
50         MPI_Bcast(&MIC_NUM,1,MPI_INT,root,MPI_COMM_WORLD);
51         MPI_Bcast(&THREAD_NUM_OMP,1,MPI_INT,root,MPI_COMM_WORLD);
52         MPI_Bcast(&THREAD_NUM_MIC,1,MPI_INT,root,MPI_COMM_WORLD);
53         MPI_Bcast(&M,1,MPI_INT,root,MPI_COMM_WORLD);
54         MPI_Bcast(&K,1,MPI_INT,root,MPI_COMM_WORLD);
55         MPI_Bcast(&N,1,MPI_INT,root,MPI_COMM_WORLD);
56         float *A, *B, *C;
57         A=(float *)malloc(sizeof(float)*M*K);
58         B=(float *)malloc(sizeof(float)*K*N);
59         C=(float *)malloc(sizeof(float)*M*N);
60         int i,j,k;
61         int ii;
62         int processID;
63         int *flag = (int *)malloc(sizeof(int)*totalrank);
64
65         for(i=0;i<totalrank;i++) //Store the line number of the matrix C in each device.
66             flag[i] = -1;
67         //Initialize A and B;
68         MPI_Bcast(B, K*N, MPI_FLOAT, root, MPI_COMM_WORLD);
69
70         for(ii=0; ii<M; ii+=Mc) /*Allocate data dynamically and receive the results from each computation
process.*/
71         {
72             MPI_Recv(&processID, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status); //Communicate with the computation processes.
73             if(flag[processID] != -1)
74                 MPI_Recv(CM+flag[processID]*N, MIN(Mc, M-flag[processID])*N, MPI_FLOAT,
processID, processID, MPI_COMM_WORLD, &status); //Receive the results from computation processes.
75             flag[processID] = ii;
76             MPI_Send(&ii, 1, MPI_INT, processID, processID, MPI_COMM_WORLD); /*Send line
number.*/
77             MPI_Send(A+ii*K, MIN(Mc, M-ii)*K, MPI_FLOAT, processID, processID,
MPI_COMM_WORLD); //Send the partitioned matrix Aii.
78         }

```

```

79     for(i=1; i<totalrank; i++) //Notify all the computation processes that the tasks have been allocated.
80     {
81         MPI_Recv(&processID, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status); //Communicate with computation processes.
82         if(flag[processID] != -1)
83             MPI_Recv(CM+flag[processID]*N, MIN(Mc, M-flag[processID])*N, MPI_FLOAT,
processID, processID, MPI_COMM_WORLD, &status); /*Receive the last result from computation
processes.*/
84         flag[processID] = -1;
85         MPI_Send(&ii, 1, MPI_INT, processID, processID, MPI_COMM_WORLD); /*Notify the
computation processes that tasks have been completed.*/
86     }
87     free(...)
88 }
89 else //computation processes
90 {
91     float *B;
92     float *Ac;
93     int ii=-1,jj,kk,jj0,jj1;
94     int i, j, k;
95     int i_end,j_end,k_end, j_end0, j_end1;
96     int M, K, N;
97     MPI_Bcast(&MIC_NUM,1,MPI_INT,root,MPI_COMM_WORLD);
98     MPI_Bcast(&THREAD_NUM_OMP,1,MPI_INT,root,MPI_COMM_WORLD);
99     MPI_Bcast(&THREAD_NUM_MIC,1,MPI_INT,root,MPI_COMM_WORLD);
100    MPI_Bcast(&M,1,MPI_INT,root,MPI_COMM_WORLD);
101    MPI_Bcast(&K,1,MPI_INT,root,MPI_COMM_WORLD);
102    MPI_Bcast(&N,1,MPI_INT,root,MPI_COMM_WORLD);
103    deviceId = (myrank-1)%(MIC_NUM+1); //Compute the device(CPU or MIC) number on single
node.
104
105    B=(float *)malloc(sizeof(float)*K*N);
106    Ac = (float *)malloc(sizeof(float)*Mc*K);
107    Ak = (float *)malloc(sizeof(float)*Mc*Kc);
108    Bc0 = (float *)malloc(sizeof(float)*Kc*Nc);
109    Bc1 = (float *)malloc(sizeof(float)*Kc*Nc);
110    Cc = (float *)malloc(sizeof(float)*Mc*N);
111
112    MPI_Bcast(B, K*N, MPI_FLOAT, root, MPI_COMM_WORLD);
113    if(deviceID<MIC_NUM) //The processes allocate space on MIC.
114    {
115        #pragma offload target(mic:deviceId) \
116        nocopy(Ak:length(Mc*Kc) alloc_if(1) free_if(0)) \
117        nocopy(Bc0:length(Kc*Nc) alloc_if(1) free_if(0)) \

```

```

118         nocopy(Bc1:length(Kc*Nc) alloc_if(1) free_if(0)) \
119         nocopy(Cc:length(Mc*N) alloc_if(1) free_if(0))
120     {
121     }
122 }
123 while(1)
124 {
125     MPI_Send(&myrank, 1, MPI_INT, 0, myrank, MPI_COMM_WORLD); /*Communicate with the
main process.*/
126     if(ii!=1)
127         MPI_Send(Cc, MIN(Mc, M-ii)*N, MPI_FLOAT, 0, myrank, MPI_COMM_WORLD);
/*Send results to the main process.*/
128     MPI_Recv(&ii, 1, MPI_INT, 0, myrank, MPI_COMM_WORLD, &status); /*Receive line
numbers.*/
129     if(ii<M)
130     {
131         MPI_Recv(Ac, MIN(Mc, M-ii)*K, MPI_FLOAT, 0, myrank, MPI_COMM_WORLD,
&status); /*Receive the partitioned matrix Ac from the main process.*/
132         for(i=0; i<Mc*N; i++)
133             Cc[i] = 0.0f;
134         if(deviceID<MIC_NUM)
135         {
136 #pragma offload target(mic: deviceID) in(Cc: length(Mc*N) alloc_if(0) free_if(0))
137             {
138             }
139         }
140
141         i_end=MIN(Mc,M-ii);
142         k_end=Kc;
143         for(kk=0; kk<K; kk+=Kc)
144         {
145             if(Kc>K-kk)
146                 k_end=K-kk;
147             for(i=0; i<i_end; i++)
148                 for(k=0; k<k_end; k++)
149                     Ak[i*Kc+k] = Ac[i*K+(kk+k)];
150             if(deviceID<MIC_NUM)
151             {
152 #pragma offload target(mic: deviceID) in(Ak: length(Mc*Kc) alloc_if(0) free_if(0))
153                 {
154                 }
155             }
156
157             j_end = Nc;

```

```

158         j_end0 = Nc;
159         j_end1 = Nc;
160         jj0=0;
161         if(Nc>N-jj0)
162             j_end0 = N-jj0;
163         for(k=0; k<k_end; k++)
164             for(j=0; j<j_end0; j++)
165                 Bc0[k*Nc+j] = B[(kk+k)*N+(jj0+j)];
166         if(deviceID<MIC_NUM)
167             {
168 #pragma offload_transfer target(mic:deviceID) in(Bc0:length(Kc*Nc) alloc_if(0) free_if(0)) signal(Bc0)
// Asynchronous data transfer.
169             }
170
171         int js=0;
172         for(jj=0;jj<N;jj+=Nc,js++)
173             {
174                 if(js%2==0)
175                 {
176                     jj1 = jj+Nc;
177                     if(jj1<N)
178                     {
179                         if(Nc>N-jj1)
180                             j_end1=N-jj1;
181                         for(k=0; k<k_end; k++)
182                             for(j=0; j<j_end1; j++)
183                                 Bc1[k*Nc+j] = B[(kk+k)*N+(jj1+j)];
184                         if(deviceID<MIC_NUM)
185                             {
186 #pragma offload_transfer target(mic:deviceID) in(Bc1:length(Kc*Nc) alloc_if(0) free_if(0)) signal(Bc1)
187                             }
188                     }
189                     if(Nc>N-jj)
190                         j_end = N-jj;
191                     if(deviceID<MIC_NUM)
192                         {
193 #pragma offload target(mic: deviceID) \
194 in(i_end, k_end, j_end, Kc, N, Nc, ii, jj) \
195 nocopy(Ak,Bc0,Cc) wait(Bc0)
196                         {
197                             matrixMul(Ak, Bc0, Cc, i_end, k_end, j_end, Kc, Nc, N, jj,
198                             THREAD_NUM_MIC); //Call the kernel of MIC.
199                         }

```

```

200         else
201         {
202             matrixMul(Ak, Bc0, Cc, i_end, k_end, j_end, Kc, Nc, N, jj,
THREAD_NUM_OMP); //Call the multi-thread kernel of CPU.
203         }
204     }
205     else
206     {
207         jj0 = jj+Nc;
208         if(jj0<N)
209         {
210             if(Nc>N-jj0)
211                 j_end0=N-jj0;
212             for(k=0; k<k_end; k++)
213                 for(j=0; j<j_end0; j++)
214                     Bc0[k*Nc+j] = B[(kk+k)*N+(jj0+j)];
215             if(deviceID<MIC_NUM)
216             {
217 #pragma offload_transfer target(mic: deviceID) in(Bc0:length(Kc*Nc) alloc_if(0) free_if(0))
signal(Bc0)
218                 }
219             }
220             if(Nc>N-jj)
221                 j_end = N-jj;
222             if(deviceID<MIC_NUM)
223             {
224 #pragma offload target(mic: deviceID) \
225                 in(i_end, k_end, j_end, Kc, N, Nc, ii, jj) \
226                 nocopy(Ak,Bc1,Cc) wait(Bc1)
227                 {
228                     matrixMul(Ak, Bc1, Cc, i_end, k_end, j_end, Kc, Nc, N, jj,
THREAD_NUM_MIC);
229                 }
230             }
231         else
232         {
233             matrixMul(Ak, Bc1, Cc, i_end, k_end, j_end, Kc, Nc, N, jj,
THREAD_NUM_OMP);
234         }
235     }
236 }
237 }
238 }
239 if(deviceID<MIC_NUM)

```

```

240         {
241     #pragma offload target(mic: deviceID) \
242         out(Cc: length(Mc*N) alloc_if(0) free_if(0)) //Return the results from MIC.
243         {
244         }
245     }
246 }
247 else
248     break; //Exit the loop.
249 }
250
251 if(deviceID<MIC_NUM)
252 {
253     #pragma offload target(mic: deviceID) \
254         nocopy(Ak: length(Mc*Kc) alloc_if(0) free_if(1)) \
255         nocopy(Bc0: length(Kc*Nc) alloc_if(0) free_if(1)) \
256         nocopy(Bc1: length(Kc*Nc) alloc_if(0) free_if(1)) \
257         nocopy(Cc: length(Mc*N) alloc_if(0) free_if(1))
258     {
259     } //Release the space on MIC.
260 }
261 free(...);
262 }
263 MPI_Finalize();
264 }

```

Note:

1. The main process is shown in lines 47–88.

First, the main process is required for broadcasting the initialized data to computing processes (lines 50–68). The main process allocates the data dynamically to the computation processes and receives the results from each computation process, which is shown in lines 70–78. The main process first receives the applications of computation processes (line 72), then checks if the computation processes have the results (line 73). If success (line 74), the main process will send the line number and block data to computation processes (line 76 and 77). Line 79–86 showss that the main process receives the results from computing processes for the last time and alert all the computation processes. Finally, the computation terminates.

2. The computation process is shown in lines 89–262.

- a. First the computation process receives initialized data from the main process (lines 97–102). Line 112 shows the value of matrix B is obtained. All the computation processes need the value of matrix B.
- b. Lines 113–122 show that the computation processes allocate memory on MIC, and Bc0, Bc1 are double buffer memory used for asynchronization operations.

- c. The whole while loop continuously applies data from main process and computes C_i for computation processes.
- d. At first the computation processes send the application from the main process, and then check for the results (line 126). They will send the results to the main process if there have been results, and then receive the line number ii (line 128). If $ii < M$, the computation hasn't completed. They will receive the A_c data from main process (line 131).
- e. The computing procedures of computational processes are the same as the block matrix multiplication and asynchronous communication (lines 132–245).

Lines 251–260 show the applied memory on MIC is released.

High-Performance Computing on the Intel® Xeon Phi™

How to Fully Exploit MIC Architectures

Wang, E.; Zhang, Q.; Shen, B.; Zhang, G.; Lu, X.; Wu, Q.;

Wang, Y.

2014, XXIII, 338 p. 153 illus., Hardcover

ISBN: 978-3-319-06485-7