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 $C = A \times B$, where $A$ is an $M \times K$ matrix, $B$ is a $K \times N$ matrix, and $C$ is a $M \times N$ matrix. The main function structure of matrix multiplication is shown in Fig. 9.1.
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

```c
int main(void)
{
    //allocate and initialize the matrix A, B, and C
    // read the matrix A and B
    ...
    //execute C=A*B on MIC card
    //export matrix C
    //release matrix A, B and C
    ...
    return 0;
}
```

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. 9.1** Main function structure of matrix multiplication
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-Threaded switched on). In this situation, the program consumes 170.83s and the speedup is 1.83. This program can be designated with P_OMP.
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]

```c
#pragma offload target(mic)
in(i, j, k, M, K, N)
in(A: length(M*K))
in(B: length(K*N))
out(C: length(M*N))
{
    #pragma omp parallel for private(j, k) num_threads(THREAD_NUM)
    for(i=0;i<M;i++)
    {
        for(j=0;j<N;j++)
        {
            float sum = 0.0f;
            #pragma ivdep
            for(k=0;k<K;k++)
            {
                sum += A[i*K + k] * B[k*widthN + j];
            }
            C[i*N + j] = sum;
        }
    }
}
```

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]
```
#pragma offload target(mic)
in(i, j, k, M, K, N)
in(A: length(M*K))
in(B: length(K*N))
out(C: length(M*N))
{
    #pragma omp parallel for private(j, k) num_threads(THREAD_NUM)
    for(i=0;i<M;i++)
    {
        for(k=0;k<K;k++)
        {
            float temp = 0.0f;
            #pragma ivdep
            for(j=0;j<N;k++)
            {
                C[i*N + j] = temp * B[k*N + j];
            }
        }
    }
}
```

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.
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.
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 $C_{m\times n} = A_{m\times k} \times B_{k\times n}$, 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 $Mc$.
- **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 $Kc$.
- **Step 3**: Based on Steps 1 and 2, partition the matrix in the direction $j$ (for $j=0;i<N;j++$), which is shown in Fig. 9.6(c). The matrices $B$ and $C$ are partitioned by $Nc$.

The sequential algorithm of matrix multiplication is shown in the Algorithm MatrixMul_V4.1.
Fig. 9.5  Diagram of matrix multiplication

\[ C_{ij} = A_{ik} B_{kj} \]

partition by direction \( k \)

partition by direction \( j \)

partition by direction \( i \)

Fig. 9.6  Partition method of matrix multiplication
Algorithm MatrixMul_V4.1

```c
#define Mc 1024
#define Kc 1024
#define Nc 1024

void matrixMul(float *A, float *B, float *C, int M, int K, int N)
{
    int i, j, k;
    int ii, jj, kk;
    int i_end, j_end, k_end;

    i_end = Mc;
    for(ii=0;ii<M;ii+=Mc)
    {
        if(Mc>M-ii)
            i_end = M-ii;
        k_end = Kc;
        for(kk=0;kk<K;kk+=Kc)
        {
            if(Kc>K-kk)
                k_end = K-kk;
            j_end = Nc;
            for(jj=0;jj<N;jj+=Nc)
            {
                if(Nc>N-jj)
                    j_end = N-jj;
                for(i=ii;i<ii+i_end;i++)
                {
                    for(j=jj;j<jj+j_end;j++)
                    {
                        float temp = 0;
                        for(k=kk;k<kk+k_end;k++)
                        {
                            temp += A[i*K+k] * B[k*N + j];
                        } //for(k=kk;k<kk+k_end;k++)
                        C[i*N + j] += temp;
                    } //for(j=jj;j<jj+j_end;j++)
                } //for(i=ii;i<ii+i_end;i++)
            } //for(jj=0;jj<N;jj+=Nc)
        } //for(kk=0;kk<K;kk+=Kc)
    } //for(ii=0;ii<M;ii+=Mc)
}
```
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.
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 asynchronousization. The asynchronous version of block matrix multiplication is shown below in Algorithm MatrixMul_V4.3.
Algorithm MatrixMul_V4.3:

```c
#define Mc 1024
#define Kc 1024
#define Nc 1024

/*It should be declared globally when the pointer nocopy is used, and the keyword attribute should be added before that.*/
#pragma offload_attribute(push, target(mic))
float *Ac;
float *Bc0;
float *Bc1; /*Declare the double buffer space Bc0 and Bc1, which are used for asynchronous data transfer.*/
float *Cc;
#pragma offload_attribute(pop)

/*The functions called in offload must be defined by the keywords __attribute__ (( target (mic))) or __declspec( target (mic))*/
__attribute__ (( target (mic)))
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)
{
    int i, j, k;
    #pragma omp parallel for private(i,j,k) num_threads(THREAD_NUM)
    for(i=0; i<i_end; i++)
    {
        for(k=0; k<k_end; k++)
        {
            float temp = Ac[i*Kc +k];
            #pragma ivdep
            for(j=0; j<j_end; j++)
            {
                Cc[(ii+i)*N +(jj+j)] += temp*Bc[k*Nc +j];
            }
        }
    }
}

void matrixMul(float *A, float *B, float *C, int M, int K, int N, int THREAD_NUM)
{
    int ii,jj,kk,jj0,jj1;
    int i_end,j_end,k_end, j_end0, j_end1;
    /*Allocate the partitioned space*/
    Ac = (float *)malloc(sizeof(float)*Mc*Kc);
```
Bc0 = (float *)malloc(sizeof(float) * Kc * Nc);
Bc1 = (float *)malloc(sizeof(float) * Kc * Nc);
Cc = C;

/* Allocate space on MIC */
#pragma offload target(mic:0) 
	nocopy(Ac:length(Mc*Kc) alloc_if(1) free_if(0)) 
	nocopy(Bc0:length(Kc*Nc) alloc_if(1) free_if(0)) 
	nocopy(Bc1:length(Kc*Nc) alloc_if(1) free_if(0)) 
	nocopy(Cc: length(M*N) alloc_if(1) free_if(0))
{
}

i_end = Mc;
for(ii=0; ii<M; ii+=Mc)
{
  if(Mc>M-ii)
    i_end=M-ii;
  k_end=Kc;
  for(kk=0; kk<K; kk+=Kc)
    {
      if(Kc>K-kk)
        k_end=K-kk;
      for(i=0; i<i_end; i++)
        for(k=0; k<k_end; k++)
          Ac[i*Kc+k] = A[(ii+i)*K+(kk+k)];
    }
  if(Kc>K-kk)
    k_end=K-kk;
  for(i=0; i<i_end; i++)
    for(k=0; k<k_end; k++)
      Ac[i*Kc+k] = A[(ii+i)*K+(kk+k)];
}

#pragma offload target(mic:0) 

in(Ac:length(Mc*Kc) alloc_if(0) free_if(0)) // Transfer A(ii, kk)
{
}

j_end = Nc;
j_end0 = Nc;
j_end1 = Nc;
jj0 = 0;
if(Nc>N-jj0)
  j_end0 = N-jj0;
for(k=0; k<k_end; k++)
  for(j=0; j<j_end0; j++)
    Bc0[k*Nc+j] = B[(kk+k)*N+(jj0+j)];

#pragma offload_transfer target(mic:0) in(Bc0:length(Kc*Nc) alloc_if(0) free_if(0)) signal(Bc0) /*
Asynchronous data transfer.*/

int js=0;
for(jj=0; jj<N; jj+=Nc, js++) // Partition B
{
if(js%2==0)
{
    jj1 = jj+Nc;
    if(jj1<N)
    {
        if(Nc>N-jj1)
            j_end1=N-jj1;
        for(k=0; k<k_end; k++)
            for(j=0; j<j_end1; j++)
                Bc1[k*Nc+j] = B[(kk+k)*N+(jj1+j)];
    #pragma offload_transfer target(mic:0) in(Bc1:length(Kc*Nc) alloc_if(0) free_if(0)) signal(Bc1) /* Asynchronous data transfer.*/
    
        if(Nc>N-jj)
            j_end = N-jj;
    #pragma offload target(mic:0) \
            in(i_end, k_end, j_end, Kc, N, Nc, ii, jj) \ 
            nocopy(Ac,Bc0,Cc) wait(Bc0)
    
        kernel(Ac, Bc0, Cc, i_end, k_end, j_end, Kc, Nc, N, ii, jj, THREAD_NUM); //Call the kernel
    
    } else
    {
        jj0 = jj+Nc;
        if(jj0<N)
        {
            if(Nc>N-jj0)
                j_end0=N-jj0;
            for(k=0; k<k_end; k++)
                for(j=0; j<j_end0; j++)
                    Bc0[k*Nc+j] = B[(kk+k)*N+(jj0+j)];
        #pragma offload_transfer target(mic:0) in(Bc0:length(Kc*Nc) alloc_if(0) free_if(0)) signal(Bc0) /* Asynchronous data transfer.*/
        } else
        {
            if(Nc>N-jj)
                j_end = N-jj;
        #pragma offload target(mic:0) \
                in(i_end, k_end, j_end, Kc, N, Nc, ii, jj) \ 
                nocopy(Ac,Bc1,Cc) wait(Bc1)
        
            kernel(Ac, Bc1, Cc, i_end, k_end, j_end, Kc, Nc, N, ii, jj, THREAD_NUM); //Call the kernel
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
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 $M_c \times N$. 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.

Fig. 9.10 Matrix multiplication based on single-node CPU+MIC hybrid computing
Algorithm MatrixMul_V4.3

```c
#define Mc 1024
#define Kc 1024
#define Nc 1024

/* It should be declared globally when the pointer nocopy is used, and the keyword attribute should be added before that.*/
#pragma offload_attribute(push, target(mic))
float *Ak;
float *Bc0;
float *Bc1; /* Declare the double buffer space Bc0 and Bc1, which are used for asynchronous data transfer.*/
#pragma offload_attribute(pop)

__attribute__((target(mic)))
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)
{
    int i, j, k;

    #pragma omp parallel for private(i, j, k) num_threads(THREAD_NUM)
    for(i=0; i<i_end; i++)
    {
        for(k=0; k<k_end; k++)
        {
            float temp = Ak[i*Kc + k];
            #pragma ivdep
            for(j=0; j<j_end; j++)
            {
                Cc[i*Nc + (jj + j)] += temp * Bc[k*Nc + j];
            }
        }
    }

    int main(int argc, char *argv[])
    {
        int THREAD_NUM_OPM = 1;
        int THREAD_NUM_MIC = 1;
        int M, K, N;
        int myrank, root=0, totalrank;
        MPI_Status status;
        int MIC_NUM = 2;
        int deviceID=-1;
```
int nodeID=-1;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD, &totalrank);

/*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*/
if(myrank==root)
{
  Initialize M, K, N, MIC_NUM, THREAD_NUM_OMP and THREAD_NUM_MIC;
  MPI_Bcast(&MIC_NUM,1,MPI_INT,root,MPI_COMM_WORLD);
  MPI_Bcast(&THREAD_NUM_OMP,1,MPI_INT,root,MPI_COMM_WORLD);
  MPI_Bcast(&THREAD_NUM_MIC,1,MPI_INT,root,MPI_COMM_WORLD);
  MPI_Bcast(&M,1,MPI_INT,root,MPI_COMM_WORLD);
  MPI_Bcast(&K,1,MPI_INT,root,MPI_COMM_WORLD);
  MPI_Bcast(&N,1,MPI_INT,root,MPI_COMM_WORLD);
  float *A, *B, *C;
  A =(float *)malloc(sizeof(float)*M*K);
  B =(float *)malloc(sizeof(float)*K*N);
  C =(float *)malloc(sizeof(float)*M*N);
  int i,j,k;
  int ii;
  int processID;
  int *flag = (int *)malloc(sizeof(int)*totalrank);

  for(i=0;i<totalrank;i++) //Store the line number of the matrix C in each device.
    flag[i] = -1;
  //Initialize A and B;
  MPI_Bcast(B, K*N, MPI_FLOAT, root, MPI_COMM_WORLD);

  for(ii=0; ii<M; ii+=Mc) /*Allocate data dynamically and receive the results from each computation process.*/
    {
      MPI_Recv(&processID, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
      MPI_COMM_WORLD, &status); //Communicate with the computation processes.
      if(flag[processID] != -1)
        {
          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.
          flag[processID] = ii;
          MPI_Send(&ii, 1, MPI_INT, processID, processID, MPI_COMM_WORLD); /*Send line number.*/
          MPI_Send(A+ii*K, MIN(Mc, M-ii)*K, MPI_FLOAT, processID, processID,
          MPI_COMM_WORLD); //Send the partitioned matrix Ai.
        }
    }
}
for(i=1; i<totalrank; i++) //Notify all the computation processes that the tasks have been allocated.
{
    MPI_Recv(&processID, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status); //Communicate with computation processes.
    if(flag[processID] != -1)
        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.*/
        flag[processID] = -1;
        MPI_Send(&ii, 1, MPI_INT, processID, processID, MPI_COMM_WORLD); /*Notify the computation processes that tasks have been completed.*/
    }
    free(…)
}
else //computation processes
{
    float *B;
    float *Ac;
    int ii=-1,jj,kk,jj0,jj1;
    int i, j, k;
    int i_end,j_end,k_end, j_end0, j_end1;
    int M, K, N;
    deviceID = (myrank-1)%(MIC_NUM+1); //Compute the device(CPU or MIC) number on single node.
    B =(float *)malloc(sizeof(float)*K*N);
    Ac = (float *)malloc(sizeof(float)*Mc*K);
    Ak = (float *)malloc(sizeof(float)*Mc*Kc);
    Bc0 = (float *)malloc(sizeof(float)*Kc*Nc);
    Bc1 = (float *)malloc(sizeof(float)*Kc*Nc);
    Cc = (float *)malloc(sizeof(float)*Mc*N);
    MPI_Bcast(B, K*N, MPI_FLOAT, root, MPI_COMM_WORLD);
    if(deviceID<MIC_NUM) //The processes allocate space on MIC.
        {
            #pragma offload target(mic:deviceID) \
            nocopy(Ak:length(Mc*Kc) alloc_if(1) free_if(0)) \
            nocopy(Bc0:length(Kc*Nc) alloc_if(1) free_if(0)) \
            nocopy(Bc1:length(Kc*Nc) alloc_if(1) free_if(0)) \
        }
9.3 Multi-thread Matrix Multiplication Based on MIC

```c
nopy(Bc:length(Kc*Nc) alloc_if(1) free_if(0)) \nopy(Cc:length(Mc*N) alloc_if(1) free_if(0))
{
    while(1)
        MPI_Send(&myrank, 1, MPI_INT, 0, myrank, MPI_COMM_WORLD); /*Communicate with the main process.*/
        if(ii!=-1)
            MPI_Send(Cc, MIN(Mc, M-ii)*N, MPI_FLOAT, 0, myrank, MPI_COMM_WORLD);
        /*Send results to the main process.*/
        MPI_Recv(&ii, 1, MPI_INT, 0, myrank, MPI_COMM_WORLD, &status); /*Receive line numbers.*/
        if(ii<M)
            {
                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.*/
            for(i=0;i<Mc*N;i++)
                Cc[i] = 0.0f;
            i_end=MIN(Mc,M-ii);
            k_end=Kc;
            for(kk=0;kk<K;kk+=Kc)
            {
                if(Kc>K-kk)
                    k_end=K-kk;
                for(i=0; i<i_end; i++)
                    for(k=0;k<k_end; k++)
                        Ak[i*Kc+k] = Ac[i*K+(kk+k)];
            if(deviceID<MIC_NUM)
                #pragma offload target(mic: deviceID) in(Ak: length(Mc*Kc) alloc_if(0) free_if(0))
        }
        j_end = Nc;
```
j_end0 = Nc;
j_end1 = Nc;
jj0=0;
if(Nc>N-jj0)
j_end0 = N-jj0;
for(k=0; k<k_end; k++)
    for(j=0; j<j_end0; j++)
        Bc0[k*Nc+j] = B[(kk+k)*N+(jj0+j)];
if(deviceID<MIC_NUM)
{
    #pragma offload transfer target(mic:deviceID) in(Bc0:length(Kc*Nc) alloc_if(0) free_if(0)) signal(Bc0)
    // Asynchronous data transfer.
}
int js=0;
for(jj=0;jj<N;jj+=Nc,js++)
{
    if(js%2==0)
    {
        jj1 = jj+Nc;
        if(jj1<N)
        {
            if(Nc>N-jj1)
                j_end1=N-jj1;
            for(k=0; k<k_end; k++)
                for(j=0; j<j_end1; j++)
                    Bc1[k*Nc+j] = B[(kk+k)*N+(jj1+j)];
            if(deviceID<MIC_NUM)
            {
                #pragma offload_transfer target(mic:deviceID) in(Bc1:length(Kc*Nc) alloc_if(0) free_if(0)) signal(Bc1)
            }
        }
    }
}
if(Nc>N-jj)
    j_end = N-jj;
if(deviceID<MIC_NUM)
{
    #pragma offload target(mic: deviceID) \
    in(i_end, k_end, j_end, Kc, N, Nc, ii, jj) \
    nocopy(Ak,Bc0,Cc) wait(Bc0)
    { 
        matrixMul(Ak, Bc0, Cc, i_end, k_end, j_end, Kc, Nc, N, jj, THREAD_NUM_MIC); //Call the kernel of MIC.
    }
200 else
201 {
202     matrixMul(Ak, Bc0, Cc, i_end, k_end, j_end, Kc, Nc, N, jj,
203         THREAD_NUM_OMP); //Call the multi-thread kernel of CPU.
204     }
205 else
206 {
207     jj0 = jj+Nc;
208     if(jj0<N)
209     {
210         if(Nc>N-jj0)
211             i_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             #pragma offload target(mic: deviceID) in(Bc0:length(Kc*Nc) alloc_if(0) free_if(0))
217             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) \ in(i_end, k_end, j_end, Kc, Nc, ii, jj) \
225         nocopy(Ak,Bc1,Cc) wait(Bc1)
226     {
227         matrixMul(Ak, Bc1, Cc, i_end, k_end, j_end, Kc, Nc, N, jj,
228             THREAD_NUM_MIC);
229         }
230     }
231     else
232     {
233         if(deviceID<MIC_NUM)
234             matrixMul(Ak, Bc1, Cc, i_end, k_end, j_end, Kc, Nc, N, jj,
235                 THREAD_NUM_OMP);
236         }
237     }
238 }
239 if(deviceID<MIC_NUM)
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