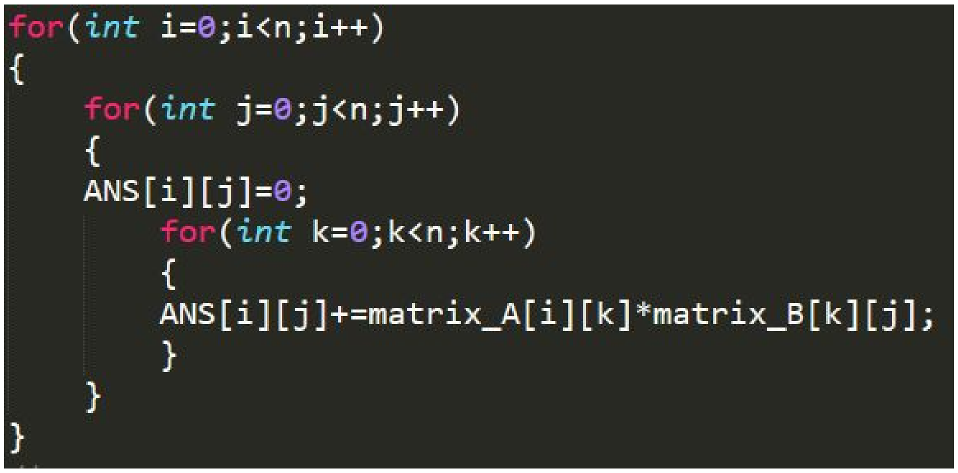
ESE565 speed up for parallel program Weimin Ruan

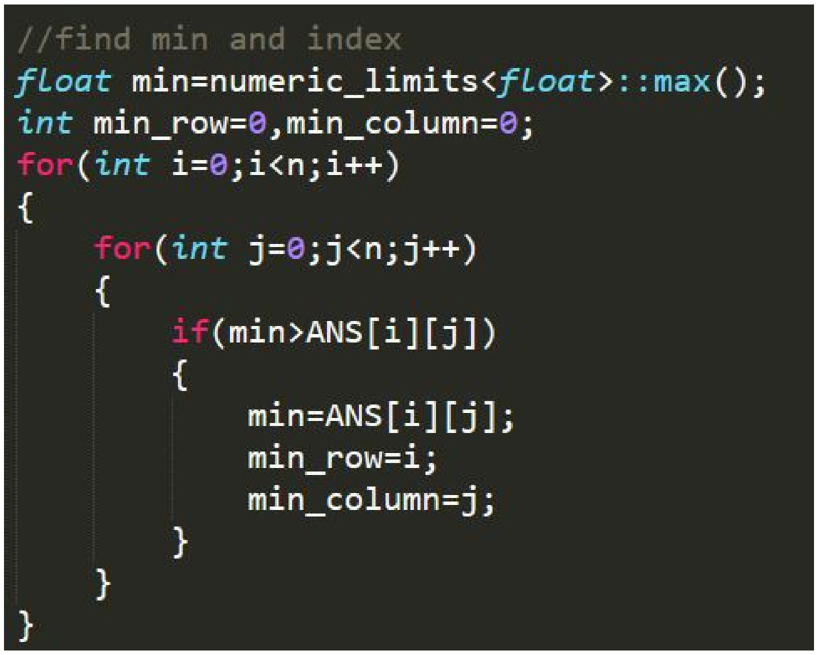
In this assignment, we are going to write some programs that runs matrix multiplication, and find minimum value and its location, we are going to use different acceleration techniques such as cache optimization,SIMD, multithreading/multitasking to speed up our little program, we will come up with a speed up table that computes the speedup of each program compares to the sequential program. And answer a few questions about which technique is the best in terms of speedup.

Sequential program:



```
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
    ANS[i][j]=0;
        for(int k=0;k<n;k++)
        {
        ANS[i][j]+=matrix_A[i][k]*matrix_B[k][j];
        }
    }
}
```

Matrix multiplication part of the program

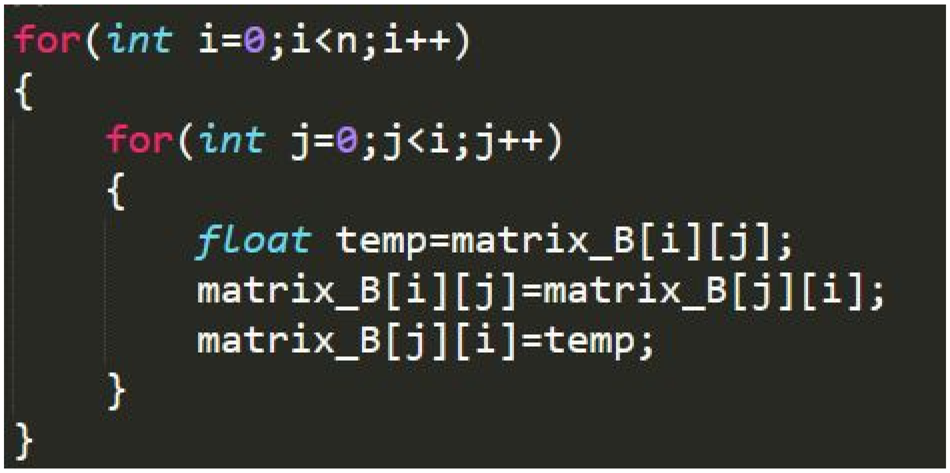


```
//find min and index
float min=numeric_limits<float>::max();
int min_row=0,min_column=0;
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        if(min>ANS[i][j])
        {
            min=ANS[i][j];
            min_row=i;
            min_column=j;
        }
    }
}
```

Minimum find part of the program

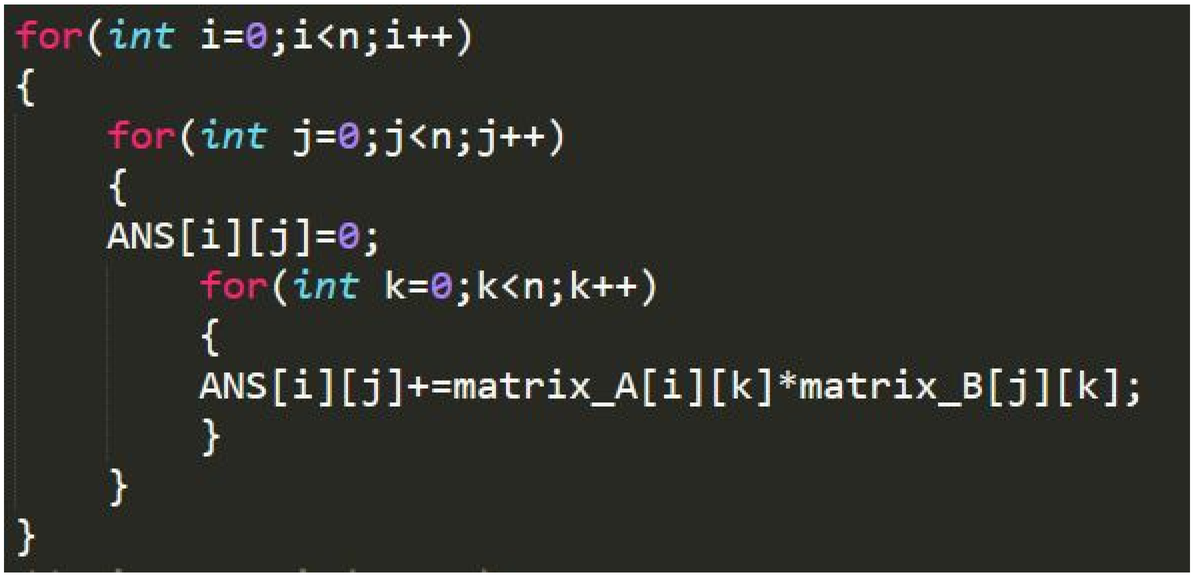
In a sequential program, the program simply follows the definition of the matrix multiplication, which is a dot product between each row of the matrix A and each column of the matrix B, and for the minimum finding, the program comparing element by element to find the index and minimum sequentially.

Sequential program with cache optimization:



```
for(int i=0;i<n;i++)
{
    for(int j=0;j<i;j++)
    {
        float temp=matrix_B[i][j];
        matrix_B[i][j]=matrix_B[j][i];
        matrix_B[j][i]=temp;
    }
}
```

Matrix transpose

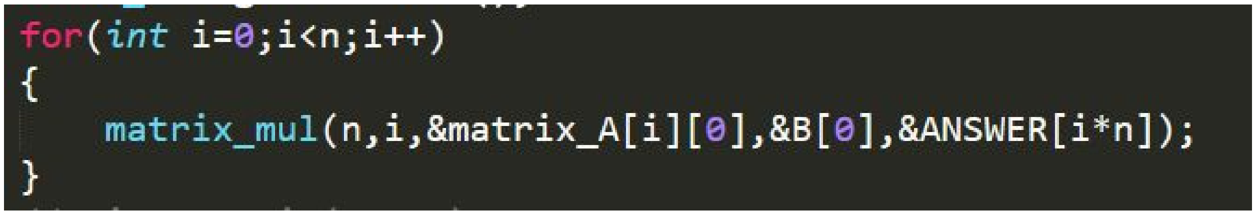


```
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
    ANS[i][j]=0;
        for(int k=0;k<n;k++)
        {
        ANS[i][j]+=matrix_A[i][k]*matrix_B[j][k];
        }
    }
}
```

Matrix multiplication

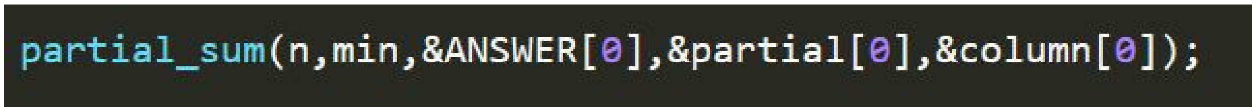
In a sequential program with cache optimization, matrix B is transposed before matrix multiplication, then calculate the dot product between each row matrix A with each row of matrix B,so the computer can have less cache miss during matrix multiplication, and minimum finding stays the same as the sequential program.

ISPC SIMD program with cache optimization:

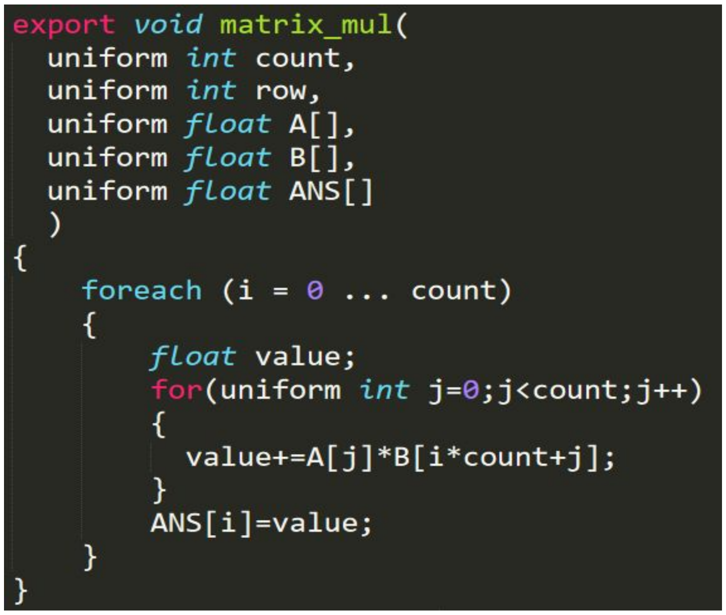


```
for(int i=0;i<n;i++)
{
    matrix_mul(n,i,&matrix_A[i][0],&B[0],&ANSWER[i*n]);
}
```

C++ code for matrix multiplication



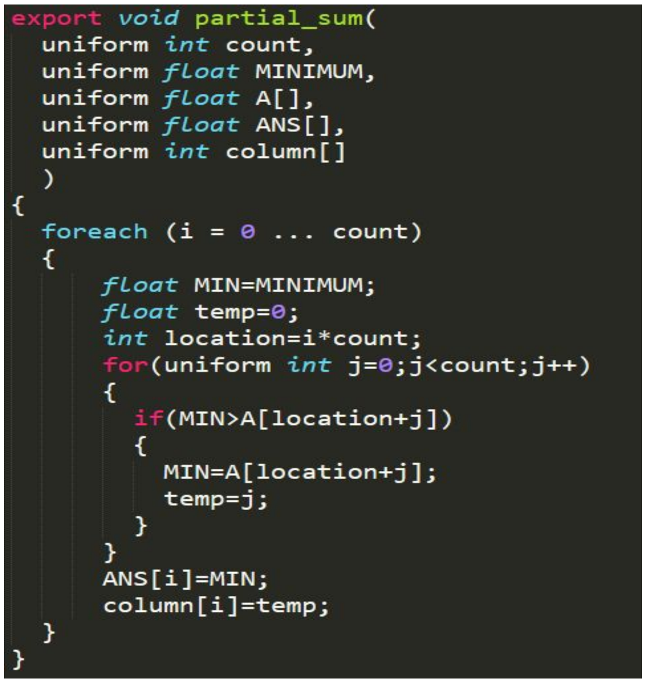
```
partial_sum(n,min,&ANSWER[0],&partial[0],&column[0]);
```



```
export void matrix_mul(
  uniform int count,
  uniform int row,
  uniform float A[],
  uniform float B[],
  uniform float ANS[]
  )
{
    foreach (i = 0 ... count)
    {
        float value;
        for(uniform int j=0;j<count;j++)
        {
          value+=A[j]*B[i*count+j];
        }
        ANS[i]=value;
    }
}
```

ISPC code for matrix multiplication

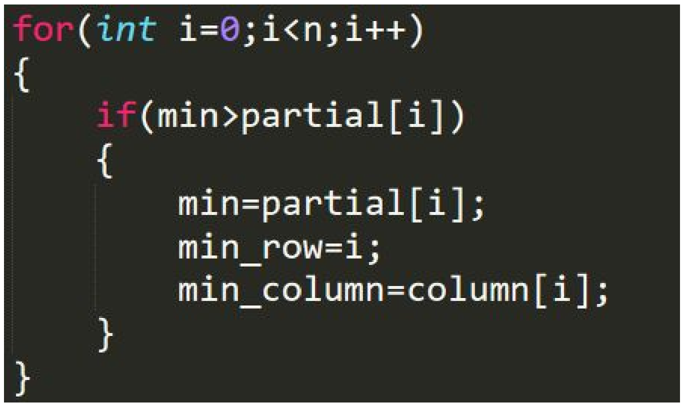
In the ISPC program, two ISPC functions are written, one is for matrix multiplication, one is for minimum finding. For matrix multiplication part, ISPC code will calculate one row of the final matrix with SIMD, and the main program written in c++ will call this function n times, where n being the matrix size of the square matrix.



```
export void partial_sum(
  uniform int count,
  uniform float MINIMUM,
  uniform float A[],
  uniform float ANS[],
  uniform int column[]
  )
{
  foreach (i = 0 ... count)
  {
      float MIN=MINIMUM;
      float temp=0;
      int location=i*count;
      for(uniform int j=0;j<count;j++)
      {
        if(MIN>A[location+j])
        {
          MIN=A[location+j];
          temp=j;
        }
      }
      ANS[i]=MIN;
      column[i]=temp;
  }
}
```

ISPC code for minimum finding

For the minimum finding part of the ISPC, after the final matrix is calculated, ISPC will look for the minimum value for each row of the final matrix, and returns it’s value and column index.

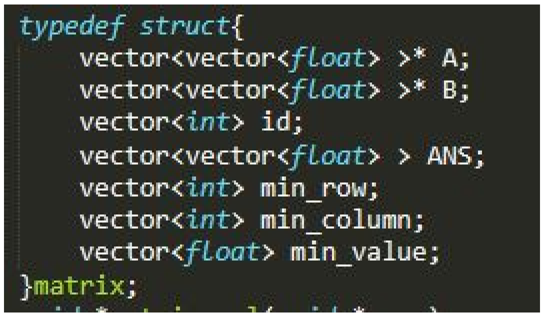


```
for(int i=0;i<n;i++)
{
    if(min>partial[i])
    {
        min=partial[i];
        min_row=i;
        min_column=column[i];
    }
}
```

C++ code for minimum finding

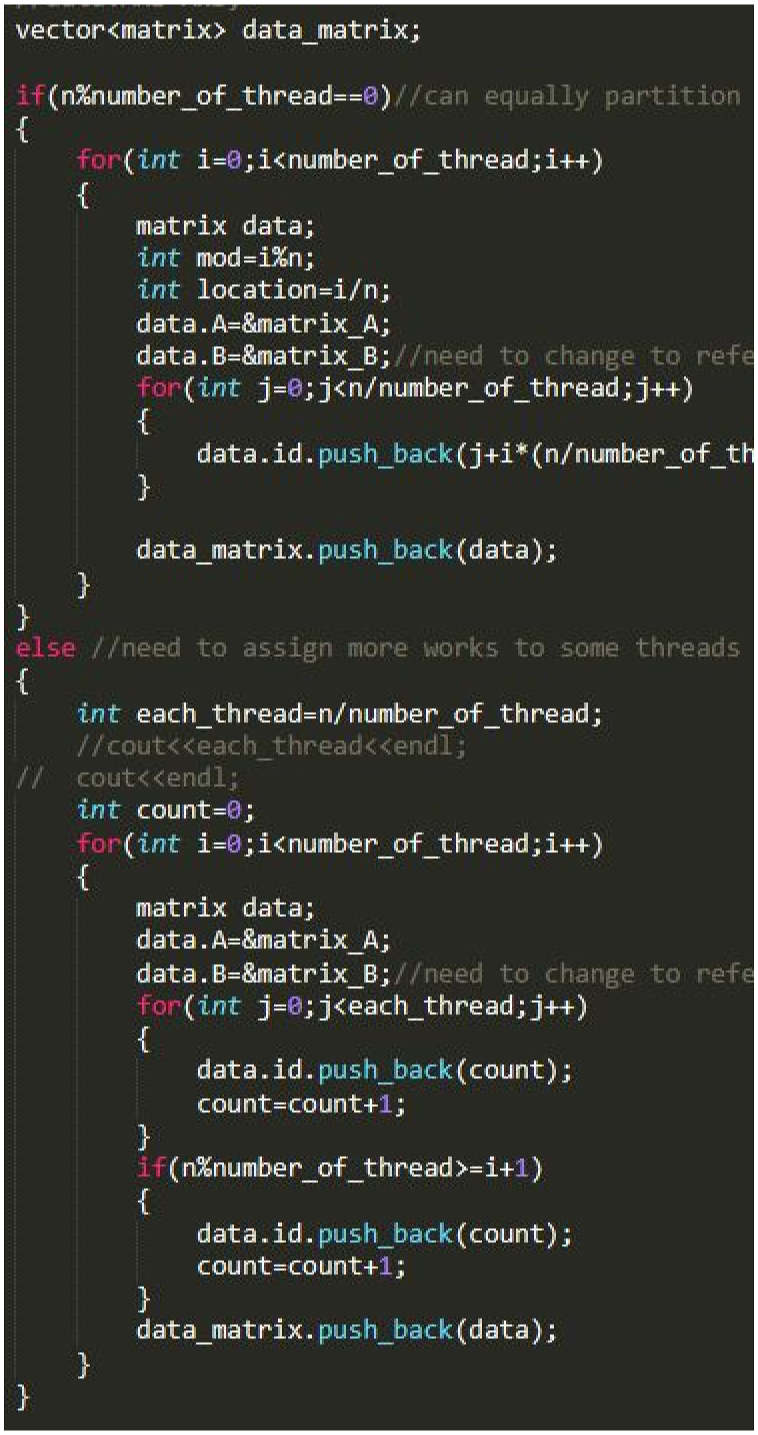
and then, In the main c++ code, we look for the minimum value and index in the array returned by ISPC code.

Pthread program with cache optimization:



```
typedef struct{
    vector<vector<float> >* A;
    vector<vector<float> >* B;
    vector<int> id;
    vector<vector<float> > ANS;
    vector<int> min_row;
    vector<int> min_column;
    vector<float> min_value;
}matrix;
```

For pthread, a special struct is define so data can be pass into the pthread function,

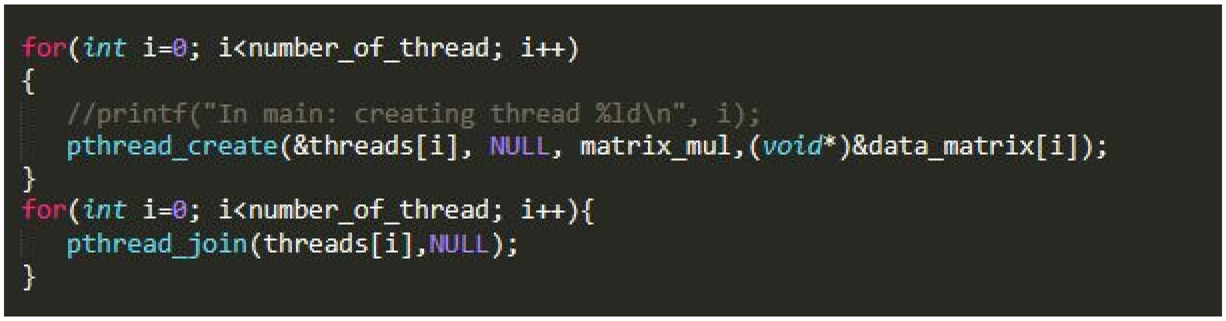


```
vector<matrix> data_matrix;

if(n%number_of_thread==0)//can equally partition
{
    for(int i=0;i<number_of_thread;i++)
    {
        matrix data;
        int mod=i%n;
        int location=i/n;
        data.A=&matrix_A;
        data.B=&matrix_B;//need to change to refe
        for(int j=0;j<n/number_of_thread;j++)
        {
            data.id.push_back(j+i*(n/number_of_th
        }

        data_matrix.push_back(data);
    }
}
else //need to assign more works to some threads
{
    int each_thread=n/number_of_thread;
    //cout<<each_thread<<endl;
//  cout<<endl;
    int count=0;
    for(int i=0;i<number_of_thread;i++)
    {
        matrix data;
        data.A=&matrix_A;
        data.B=&matrix_B;//need to change to refe
        for(int j=0;j<each_thread;j++)
        {
            data.id.push_back(count);
            count=count+1;
        }
        if(n%number_of_thread>=i+1)
        {
            data.id.push_back(count);
            count=count+1;
        }
        data_matrix.push_back(data);
    }
}
```

Task division algorithm

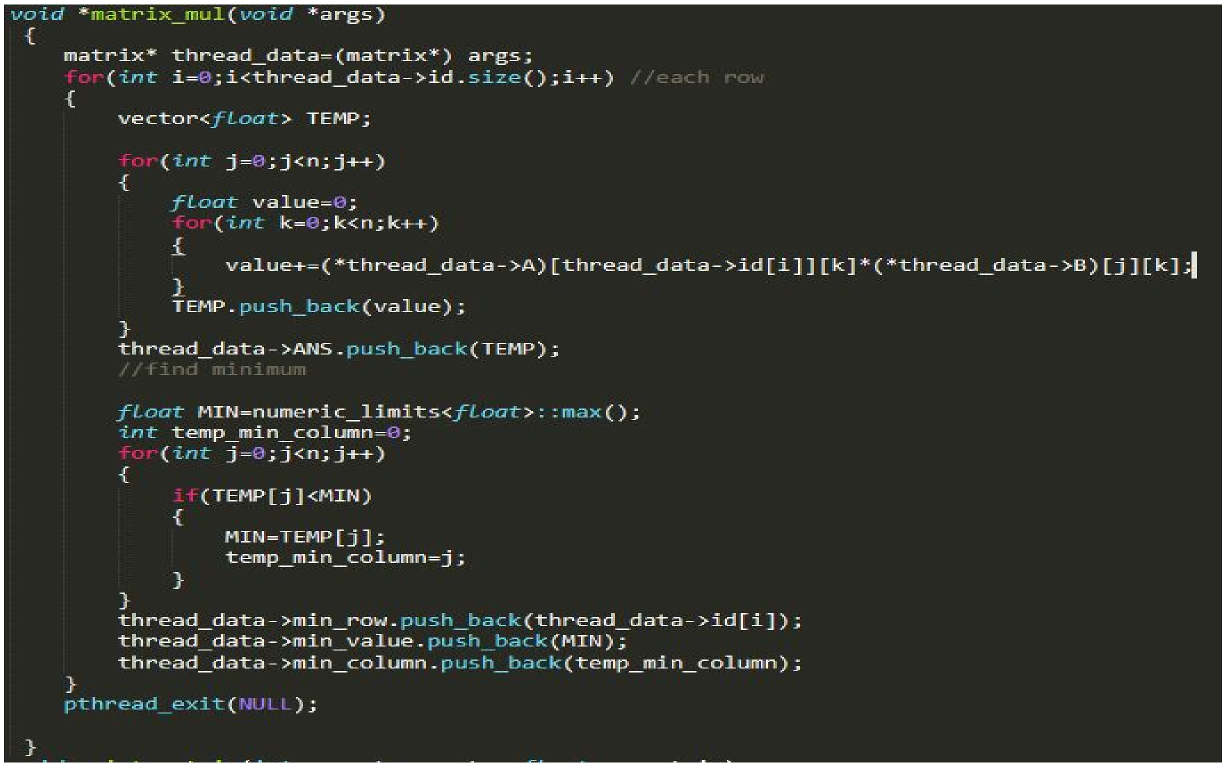
In order to make each thread working on subpart of the whole program, a partition algorithm is required, if the number of task is divisible by thread number, then each thread is assign same number of jobs, if it is not, then every thread will receive floor(task/thread) numbers of jobs, then reminder tasks will be assign to the

first few threads. For example, if 4 threads exist, 10 tasks are given by the program, then each thread will at least have floor(10/4) = 2 tasks, and thread 0 and thread 1 will take one more task to make the reminder task number equal to zero.
Thread 0: 0,1,2
Thread 1: 3,4,5
Thread 2: 6,7
Thread 3: 8,9
In matrix multiplication, since we have the final matrix will have n rows, n tasks are out there.



```
for(int i=0; i<number_of_thread; i++)
{
    //printf("In main: creating thread %ld\n", i);
    pthread_create(&threads[i], NULL, matrix_mul,(void*)&data_matrix[i]);
}
for(int i=0; i<number_of_thread; i++){
    pthread_join(threads[i],NULL);
}
```

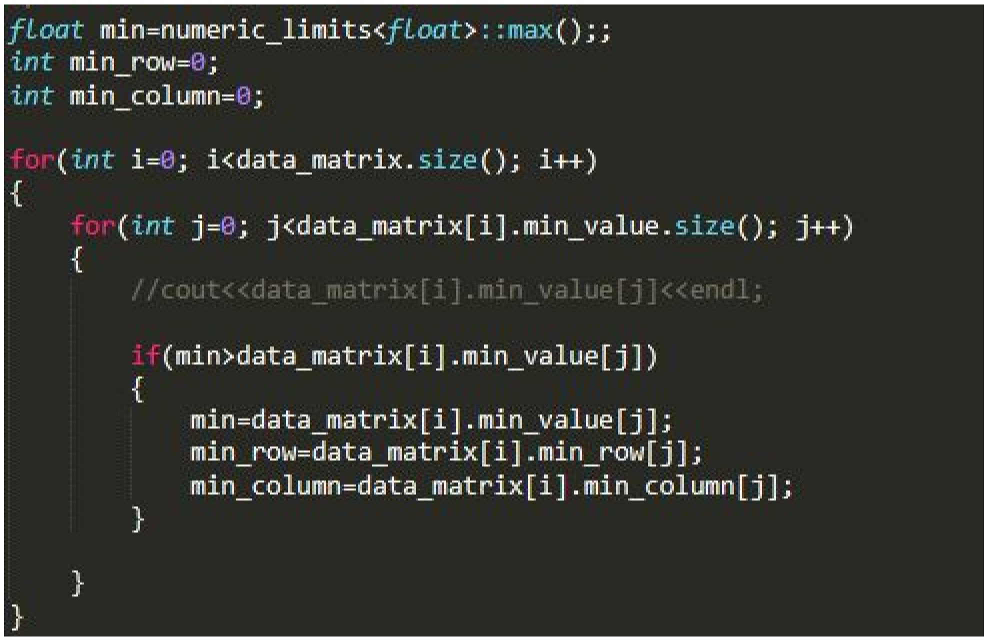
Function call in main
    In this for loop, each thread will start executing matrix_mul with different dataset



```
void *matrix_mul(void *args)
 {
    matrix* thread_data=(matrix*) args;
    for(int i=0;i<thread_data->id.size();i++) //each row
    {
        vector<float> TEMP;

        for(int j=0;j<n;j++)
        {
            float value=0;
            for(int k=0;k<n;k++)
            {
                value+=(*thread_data->A)[thread_data->id[i]][k]*(*thread_data->B)[j][k];
            }
            TEMP.push_back(value);
        }
        thread_data->ANS.push_back(TEMP);
        //find minimum

        float MIN=numeric_limits<float>::max();
        int temp_min_column=0;
        for(int j=0;j<n;j++)
        {
            if(TEMP[j]<MIN)
            {
                MIN=TEMP[j];
                temp_min_column=j;
            }
        }
        thread_data->min_row.push_back(thread_data->id[i]);
        thread_data->min_value.push_back(MIN);
        thread_data->min_column.push_back(temp_min_column);
    }
    pthread_exit(NULL);

 }
```

Pthread function

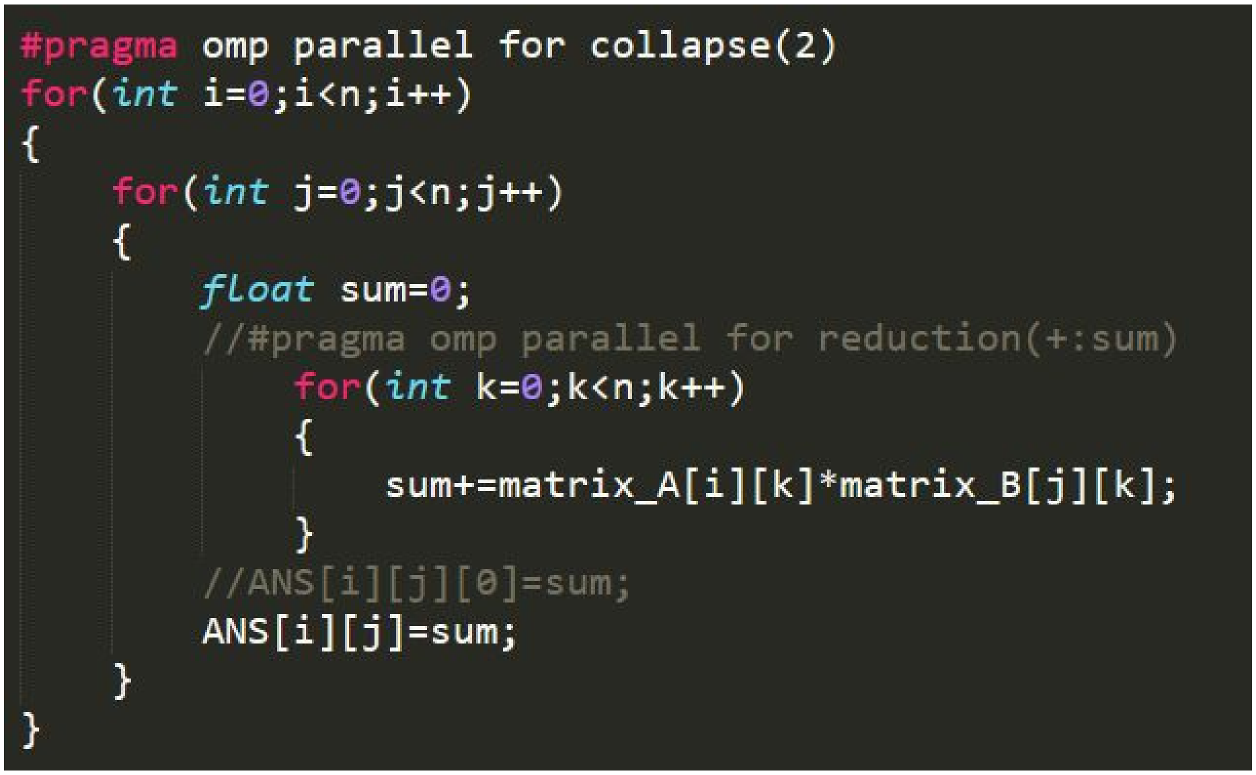
This function will calculate the a subpart of the final matrix, and also find the minimum value and index after subpart matrix multiplication is completed.



```
float min=numeric_limits<float>::max();;
int min_row=0;
int min_column=0;

for(int i=0; i<data_matrix.size(); i++)
{
    for(int j=0; j<data_matrix[i].min_value.size(); j++)
    {
        //cout<<data_matrix[i].min_value[j]<<endl;

        if(min>data_matrix[i].min_value[j])
        {
            min=data_matrix[i].min_value[j];
            min_row=data_matrix[i].min_row[j];
            min_column=data_matrix[i].min_column[j];
        }

    }
}
```

After the pthread function, the program will have a collection of minimum values and indexes, and search minimum value and index sequentially among the returned array will find the minimum value and index for the whole matrix.

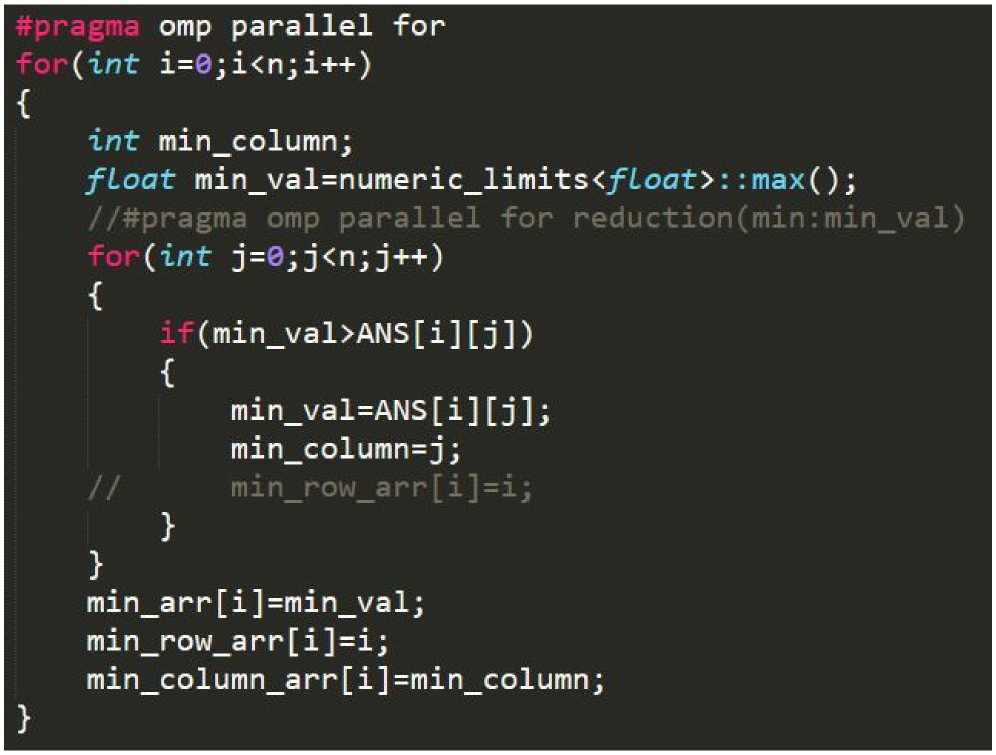
OMP program with cache optimization:



```
#pragma omp parallel for collapse(2)
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        float sum=0;
        //#pragma omp parallel for reduction(+:sum)
            for(int k=0;k<n;k++)
            {
                sum+=matrix_A[i][k]*matrix_B[j][k];
            }
        //ANS[i][j][0]=sum;
        ANS[i][j]=sum;
    }
}
```

OPENMP program for matrix multiplication

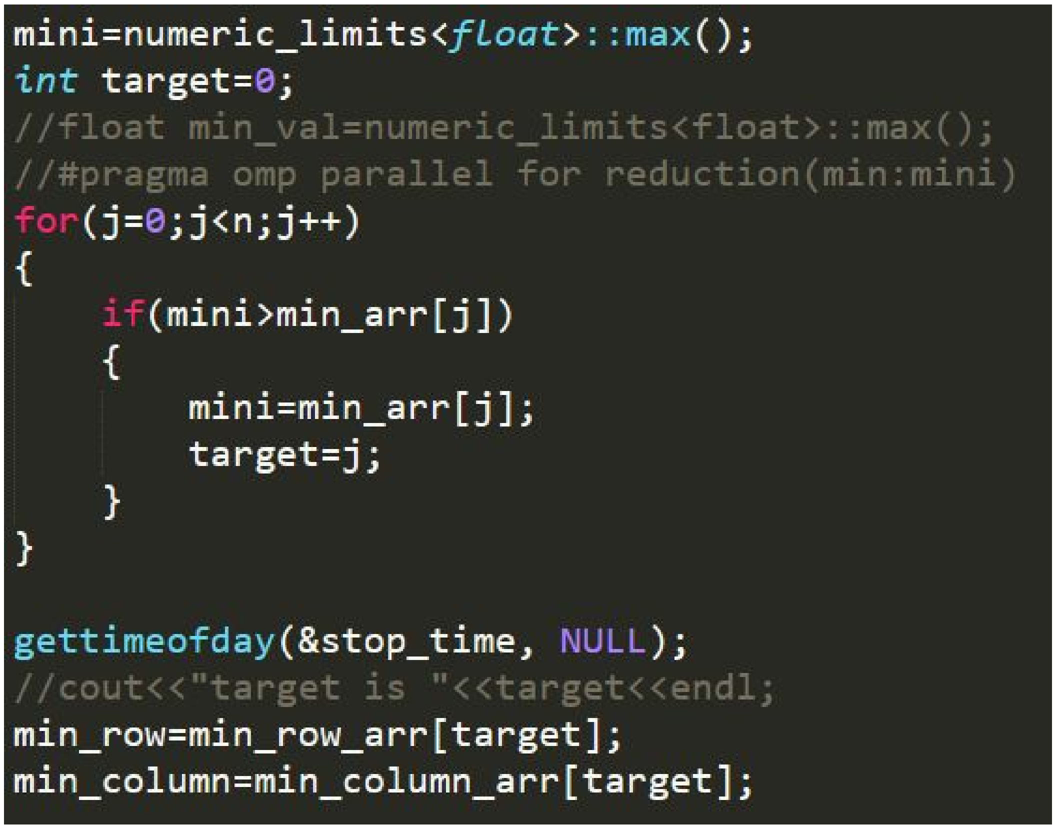
For openmp, there is a special syntax collapse() which allows nested loops to be executed parallelly, which is perfect for matrix multiplication, which means every element in the final matrix can be calculated parallelly.



```
#pragma omp parallel for
for(int i=0;i<n;i++)
{
    int min_column;
    float min_val=numeric_limits<float>::max();
    //#pragma omp parallel for reduction(min:min_val)
    for(int j=0;j<n;j++)
    {
        if(min_val>ANS[i][j])
        {
            min_val=ANS[i][j];
            min_column=j;
    //      min_row_arr[i]=i;
        }
    }
    min_arr[i]=min_val;
    min_row_arr[i]=i;
    min_column_arr[i]=min_column;
}
```

OPENMP program for minimum value and index

This openmp program finds the minimum value and index for each row of the final matrix parallelly.

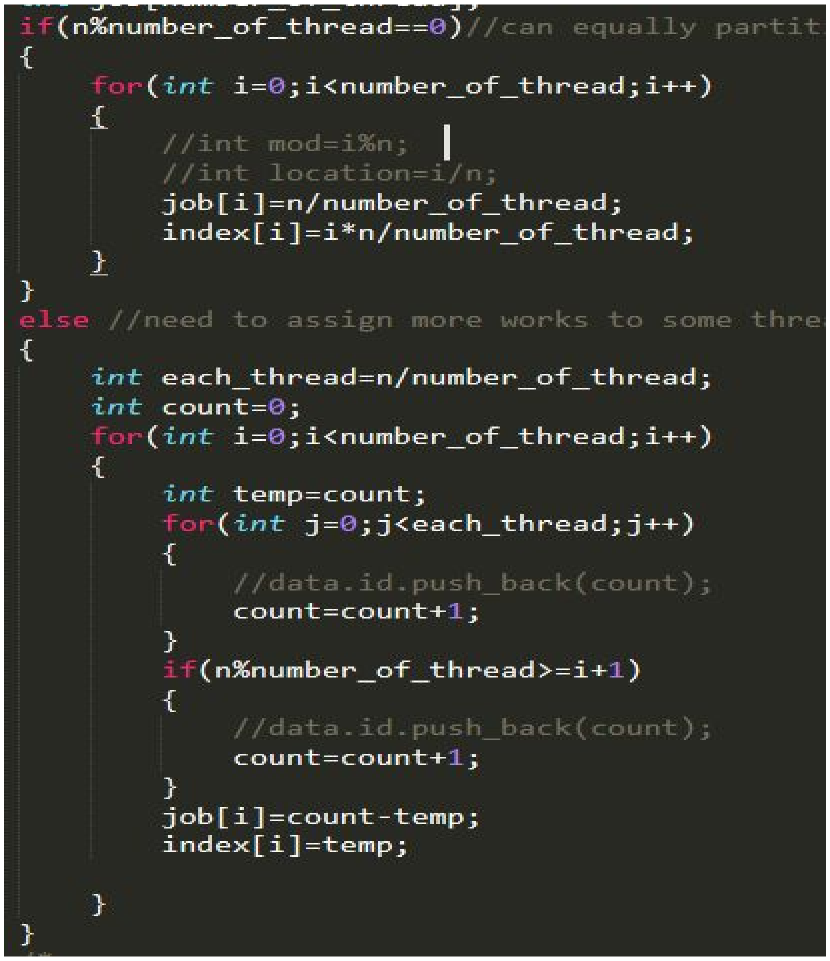


```
mini=numeric_limits<float>::max();
int target=0;
//float min_val=numeric_limits<float>::max();
//#pragma omp parallel for reduction(min:mini)
for(j=0;j<n;j++)
{
    if(mini>min_arr[j])
    {
        mini=min_arr[j];
        target=j;
    }
}

gettimeofday(&stop_time, NULL);
//cout<<"target is "<<target<<endl;
min_row=min_row_arr[target];
min_column=min_column_arr[target];
```

Sequential minimum value finding

Then, a sequential program finds the minimum value and index among values returned by openmp minimum value finding.

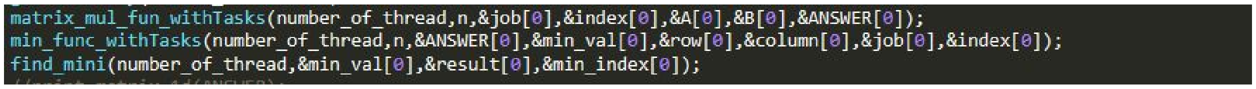
ISPC program with task ISPC tasks:



```
if(n%number_of_thread==0)//can equally partit
{
    for(int i=0;i<number_of_thread;i++)
    {
        //int mod=i%n;
        //int location=i/n;
        job[i]=n/number_of_thread;
        index[i]=i*n/number_of_thread;
    }
}
else //need to assign more works to some thre
{
    int each_thread=n/number_of_thread;
    int count=0;
    for(int i=0;i<number_of_thread;i++)
    {
        int temp=count;
        for(int j=0;j<each_thread;j++)
        {
            //data.id.push_back(count);
            count=count+1;
        }
        if(n%number_of_thread>=i+1)
        {
            //data.id.push_back(count);
            count=count+1;
        }
        job[i]=count-temp;
        index[i]=temp;

    }
}
```

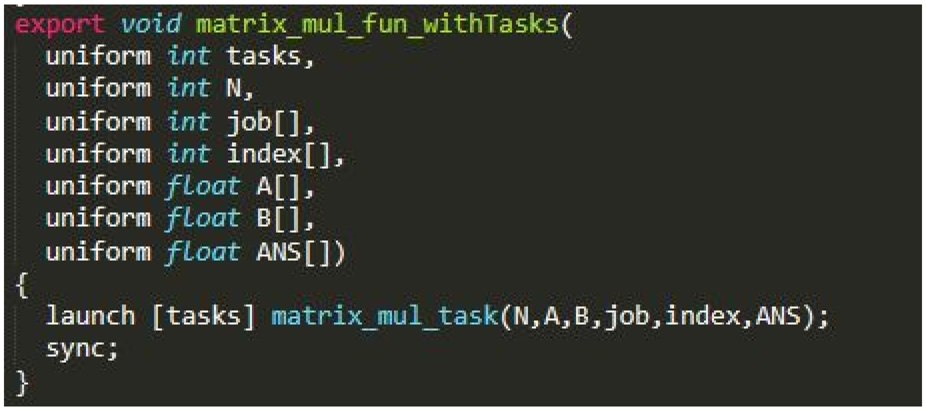
Partition algorithm

This part works the same way as the one in pthread, with different variable names. Basically, assignment different row numbers of the final matrix that are needed to be calculated to different threads.



```
matrix_mul_fun_withTasks(number_of_thread,n,&job[0],&index[0],&A[0],&B[0],&ANSWER[0]);
min_func_withTasks(number_of_thread,n,&ANSWER[0],&min_val[0],&row[0],&column[0],&job[0],&index[0]);
find_mini(number_of_thread,&min_val[0],&result[0],&min_index[0]);
```

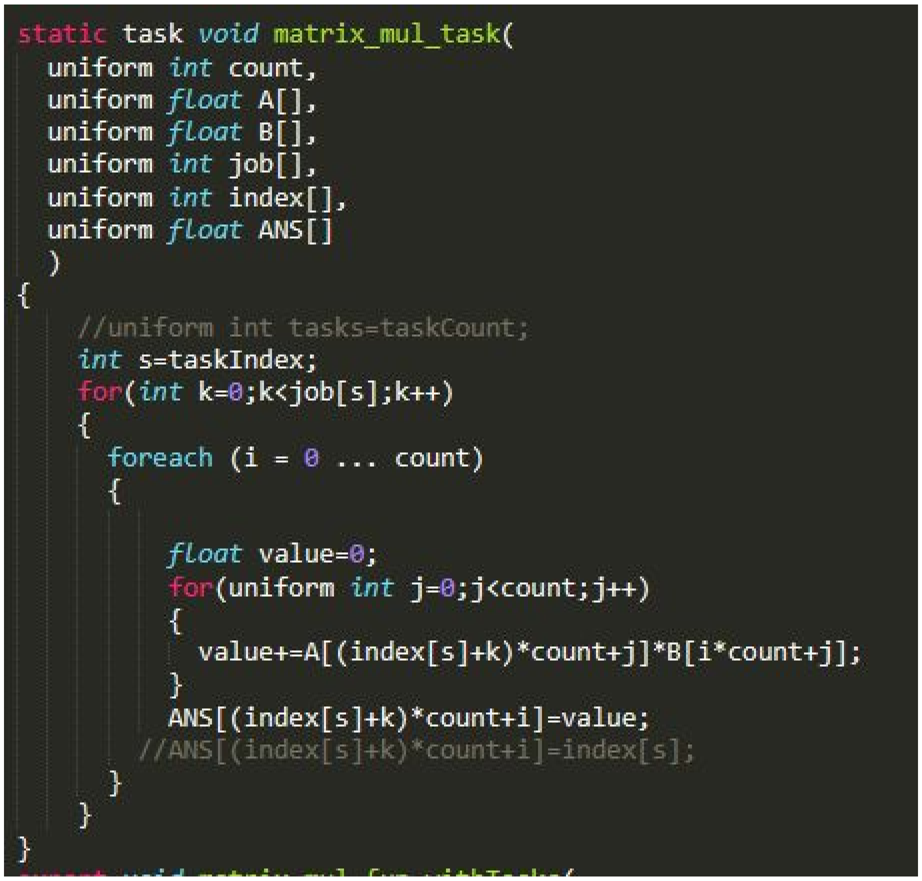
Function call to ISPC function.



```
export void matrix_mul_fun_withTasks(
  uniform int tasks,
  uniform int N,
  uniform int job[],
  uniform int index[],
  uniform float A[],
  uniform float B[],
  uniform float ANS[])
{
  launch [tasks] matrix_mul_task(N,A,B,job,index,ANS);
  sync;
}
```

Task function for matrix multiplication

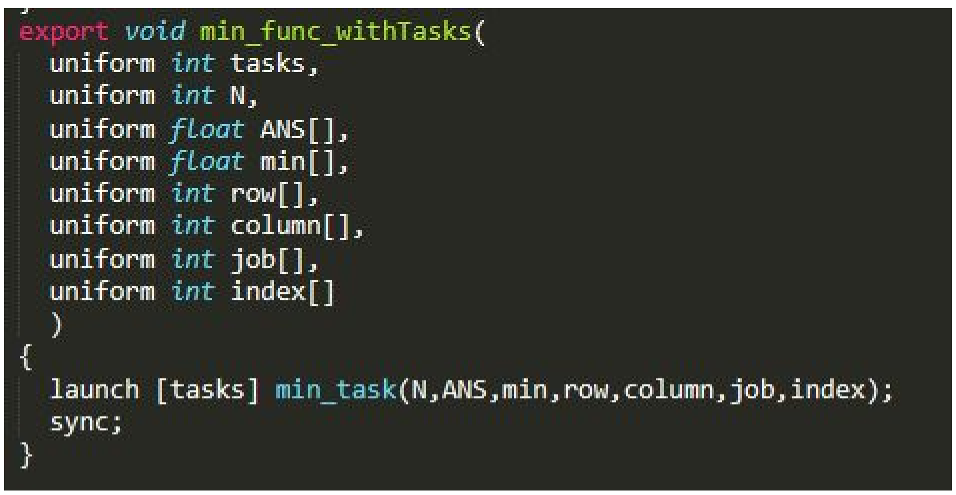
This function initializes “tasks” times to matrix_mul_task to different threads, the “tasks” is equal to thread_num defined by the user.



```
static task void matrix_mul_task(
  uniform int count,
  uniform float A[],
  uniform float B[],
  uniform int job[],
  uniform int index[],
  uniform float ANS[]
  )
{
    //uniform int tasks=taskCount;
    int s=taskIndex;
    for(int k=0;k<job[s];k++)
    {
      foreach (i = 0 ... count)
      {

          float value=0;
          for(uniform int j=0;j<count;j++)
          {
            value+=A[(index[s]+k)*count+j]*B[i*count+j];
          }
          ANS[(index[s]+k)*count+i]=value;
        //ANS[(index[s]+k)*count+i]=index[s];
      }
    }
}
```

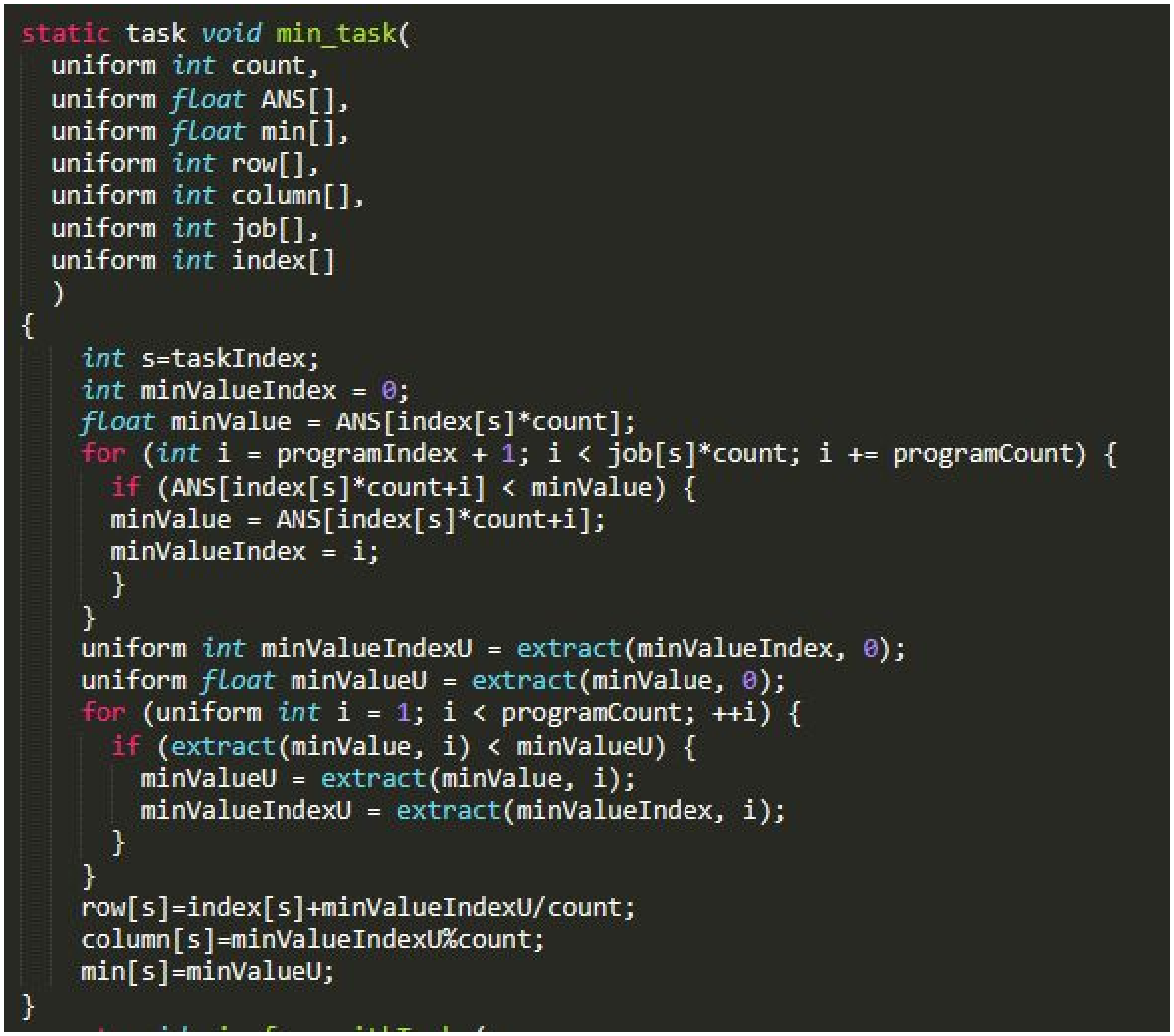
ISPC SIMD function for matrix multiplication

This part is very similar to the ISPC SIMD function from ISPC program with cache optimization, the differences are the external for loop is determined by job[s] and the calculated result is dependent on index[s], these two arrays are assigned by the partition algorithm to make the function running on different thread parallely.



```
export void min_func_withTasks(
  uniform int tasks,
  uniform int N,
  uniform float ANS[],
  uniform float min[],
  uniform int row[],
  uniform int column[],
  uniform int job[],
  uniform int index[]
  )
{
  launch [tasks] min_task(N,ANS,min,row,column,job,index);
  sync;
}
```

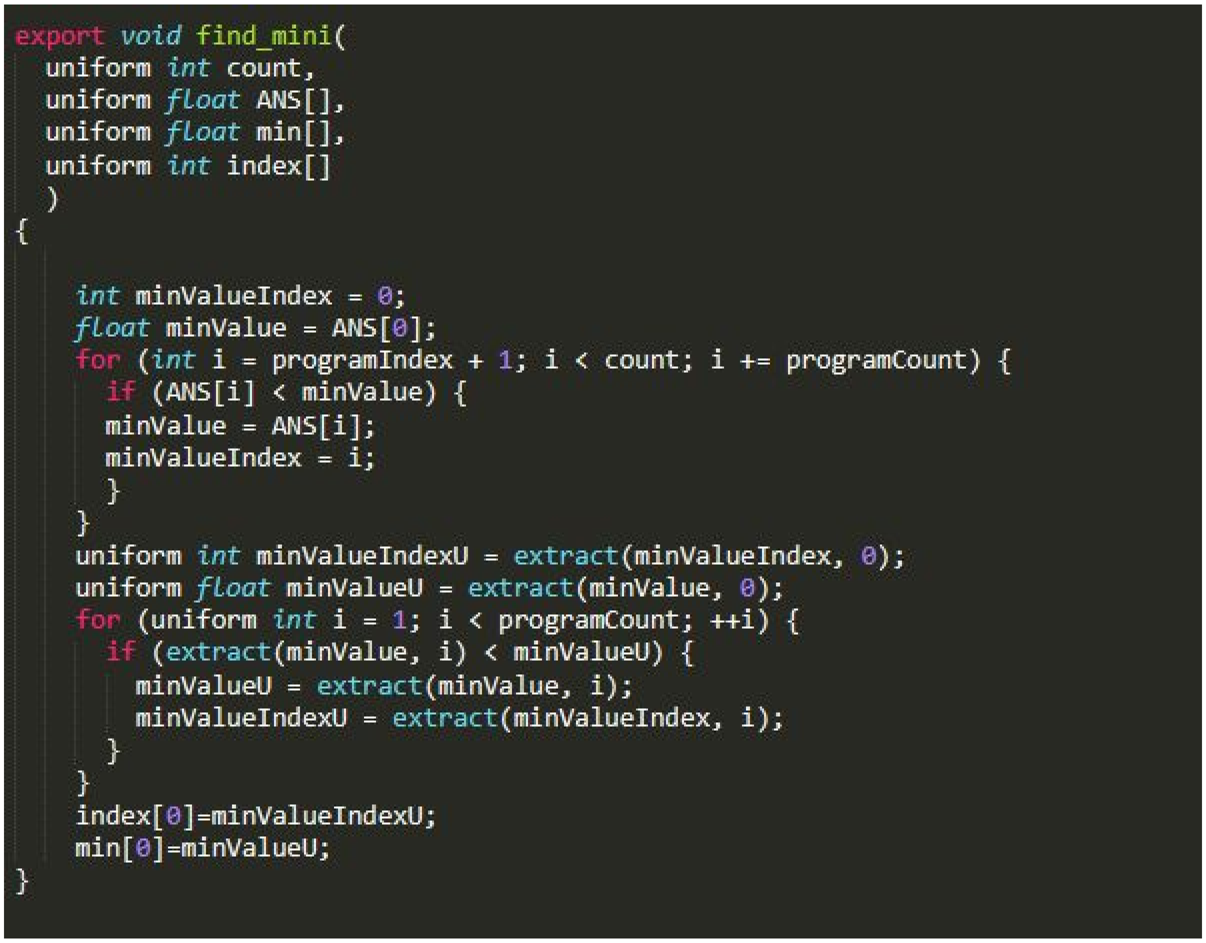
Task function for minimum finding



```
static task void min_task(
  uniform int count,
  uniform float ANS[],
  uniform float min[],
  uniform int row[],
  uniform int column[],
  uniform int job[],
  uniform int index[]
  )
{
    int s=taskIndex;
    int minValueIndex = 0;
    float minValue = ANS[index[s]*count];
    for (int i = programIndex + 1; i < job[s]*count; i += programCount) {
      if (ANS[index[s]*count+i] < minValue) {
      minValue = ANS[index[s]*count+i];
      minValueIndex = i;
      }
    }
    uniform int minValueIndexU = extract(minValueIndex, 0);
    uniform float minValueU = extract(minValue, 0);
    for (uniform int i = 1; i < programCount; ++i) {
      if (extract(minValue, i) < minValueU) {
        minValueU = extract(minValue, i);
        minValueIndexU = extract(minValueIndex, i);
      }
    }
    row[s]=index[s]+minValueIndexU/count;
    column[s]=minValueIndexU%count;
    min[s]=minValueU;
}
```

ISPC SIMD function for minimum finding

This function is also very similar to the “partial sum” function in ISPC programs with cache optimization; by in there, the function looks for the minimum value and index in every row and returns those arrays as a result. Here the program looks for the minimum value and index for the given dataset which is defined by the partition algorithm.

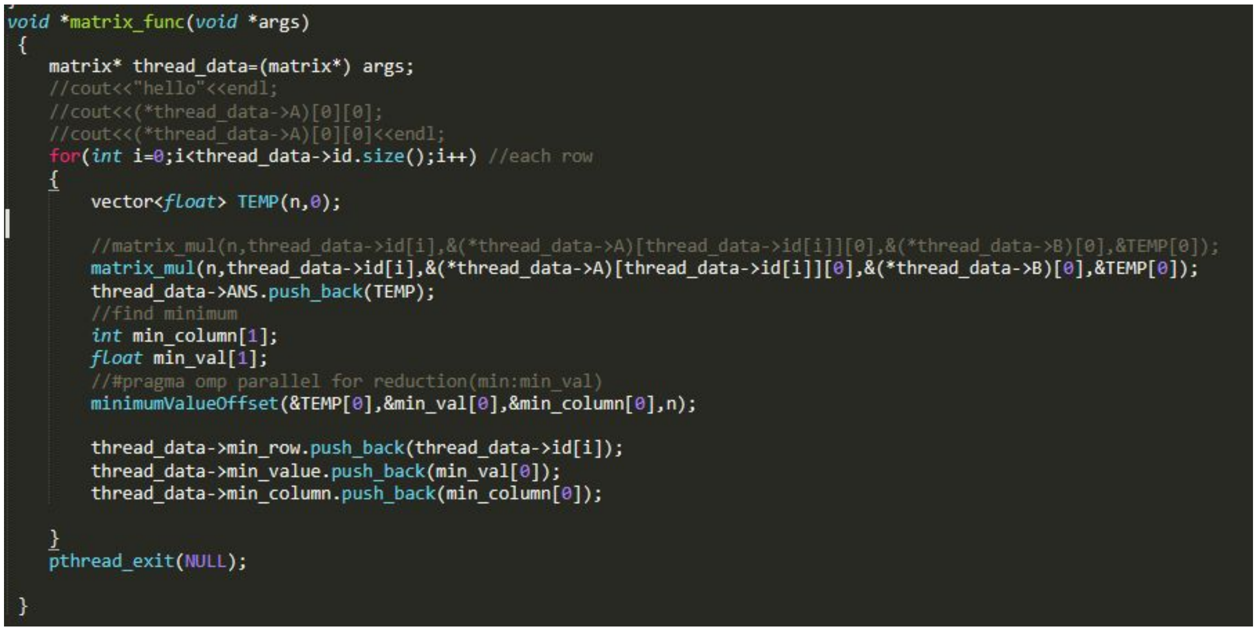


```
export void find_mini(
  uniform int count,
  uniform float ANS[],
  uniform float min[],
  uniform int index[]
  )
{

    int minValueIndex = 0;
    float minValue = ANS[0];
    for (int i = programIndex + 1; i < count; i += programCount) {
      if (ANS[i] < minValue) {
      minValue = ANS[i];
      minValueIndex = i;
      }
    }
    uniform int minValueIndexU = extract(minValueIndex, 0);
    uniform float minValueU = extract(minValue, 0);
    for (uniform int i = 1; i < programCount; ++i) {
      if (extract(minValue, i) < minValueU) {
        minValueU = extract(minValue, i);
        minValueIndexU = extract(minValueIndex, i);
      }
    }
    index[0]=minValueIndexU;
    min[0]=minValueU;
}
```

Find minimum value and index

After the previous ISPC task function call for minimum value finding, the returned array contains "num_of_thread" minimum values and indexes, the program needs to run another minimum finding function to find the global minimum value and index.This function finds minimum value and index with simple SIMD.

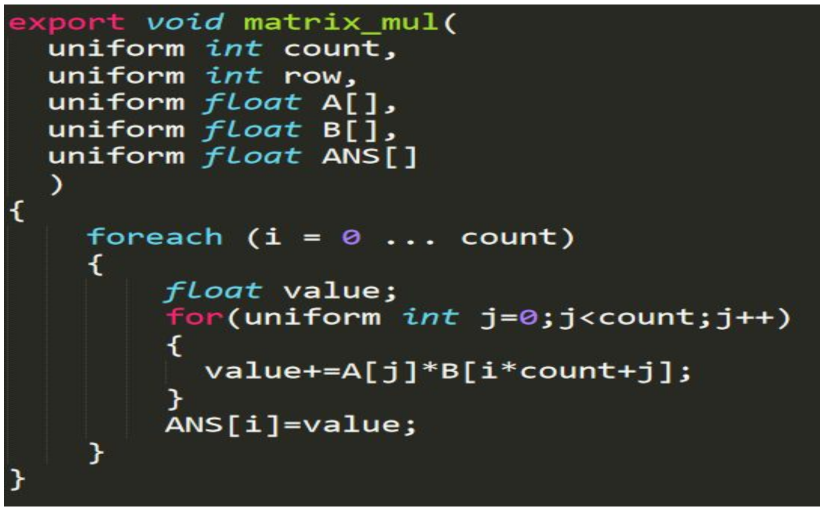
Pthread program with ISPC SIMD:



```
void *matrix_func(void *args)
 {
    matrix* thread_data=(matrix*) args;
    //cout<<"hello"<<endl;
    //cout<<(*thread_data->A)[0][0];
    //cout<<(*thread_data->A)[0][0]<<endl;
    for(int i=0;i<thread_data->id.size();i++) //each row
    {
        vector<float> TEMP(n,0);

        //matrix_mul(n,thread_data->id[i],&(*thread_data->A)[thread_data->id[i]][0],&(*thread_data->B)[0],&TEMP[0]);
        matrix_mul(n,thread_data->id[i],&(*thread_data->A)[thread_data->id[i]][0],&(*thread_data->B)[0],&TEMP[0]);
        thread_data->ANS.push_back(TEMP);
        //find minimum
        int min_column[1];
        float min_val[1];
        //#pragma omp parallel for reduction(min:min_val)
        minimumValueOffset(&TEMP[0],&min_val[0],&min_column[0],n);

        thread_data->min_row.push_back(thread_data->id[i]);
        thread_data->min_value.push_back(min_val[0]);
        thread_data->min_column.push_back(min_column[0]);

    }
    pthread_exit(NULL);

 }
```

Pthread function that calls ISPC function.

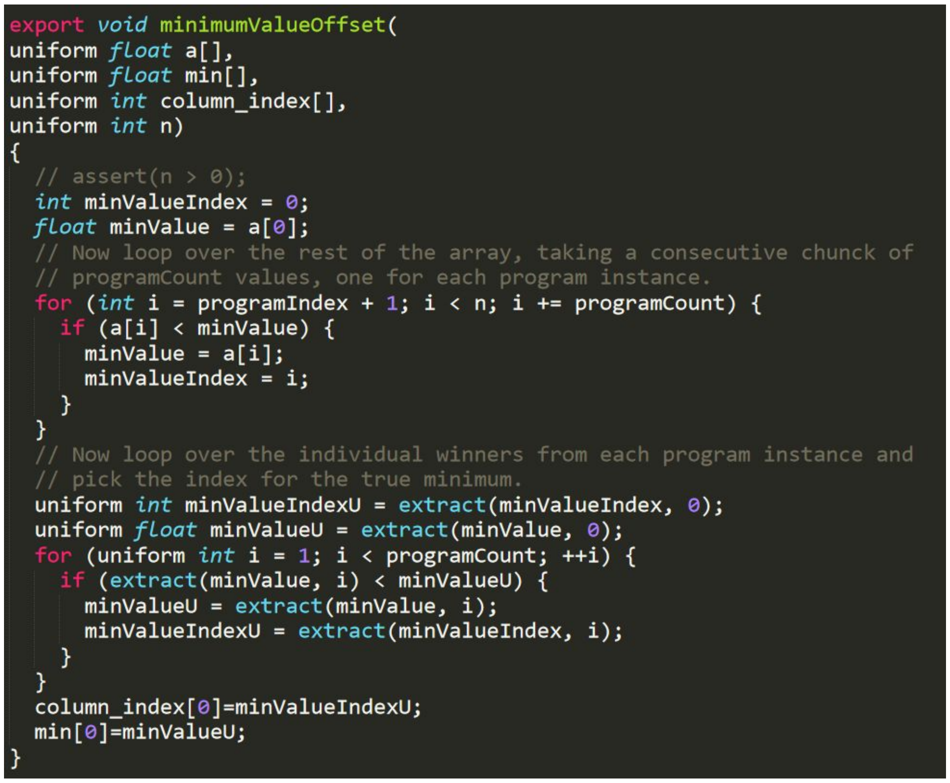
This function is very similar to the pthread function from a pthread program with cache optimization, but in here, all the matrix calculation and minimum finding is given to the ISPC function.



```
export void matrix_mul(
   uniform int count,
   uniform int row,
   uniform float A[],
   uniform float B[],
   uniform float ANS[]
   )
{
    foreach (i = 0 ... count)
    {
        float value;
        for(uniform int j=0;j<count;j++)
        {
            value+=A[j]*B[i*count+j];
        }
        ANS[i]=value;
    }
}
```

ISPC Code for matrix multiplication

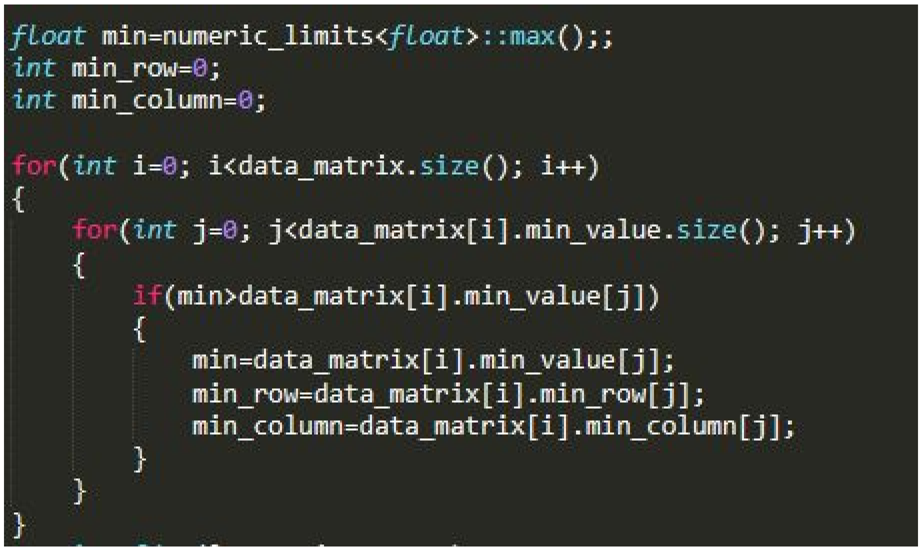
This function calculates a row of dot product for the final matrix with ISPC SIMD.



```
export void minimumValueOffset(
uniform float a[],
uniform float min[],
uniform int column_index[],
uniform int n)
{
  // assert(n > 0);
  int minValueIndex = 0;
  float minValue = a[0];
  // Now loop over the rest of the array, taking a consecutive chunck of
  // programCount values, one for each program instance.
  for (int i = programIndex + 1; i < n; i += programCount) {
    if (a[i] < minValue) {
      minValue = a[i];
      minValueIndex = i;
    }
  }
  // Now loop over the individual winners from each program instance and
  // pick the index for the true minimum.
  uniform int minValueIndexU = extract(minValueIndex, 0);
  uniform float minValueU = extract(minValue, 0);
  for (uniform int i = 1; i < programCount; ++i) {
    if (extract(minValue, i) < minValueU) {
      minValueU = extract(minValue, i);
      minValueIndexU = extract(minValueIndex, i);
    }
  }
  column_index[0]=minValueIndexU;
  min[0]=minValueU;
}
```

ISPC code for minimum finding

This function finds the minimum value and index for one row of the final matrix with ISPC SIMD.

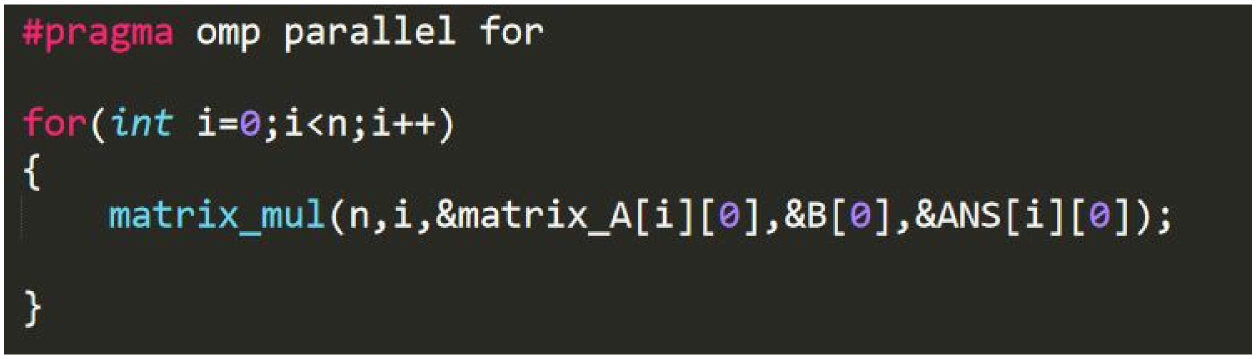


```
float min=numeric_limits<float>::max();;
int min_row=0;
int min_column=0;

for(int i=0; i<data_matrix.size(); i++)
{
    for(int j=0; j<data_matrix[i].min_value.size(); j++)
    {
        if(min>data_matrix[i].min_value[j])
        {
            min=data_matrix[i].min_value[j];
            min_row=data_matrix[i].min_row[j];
            min_column=data_matrix[i].min_column[j];
        }
    }
}
```

Sequential minimum finding

This function finds the global minimum value and index for the whole matrix by looking over the minimum value array by the pthread just like pthread program with cache optimization.

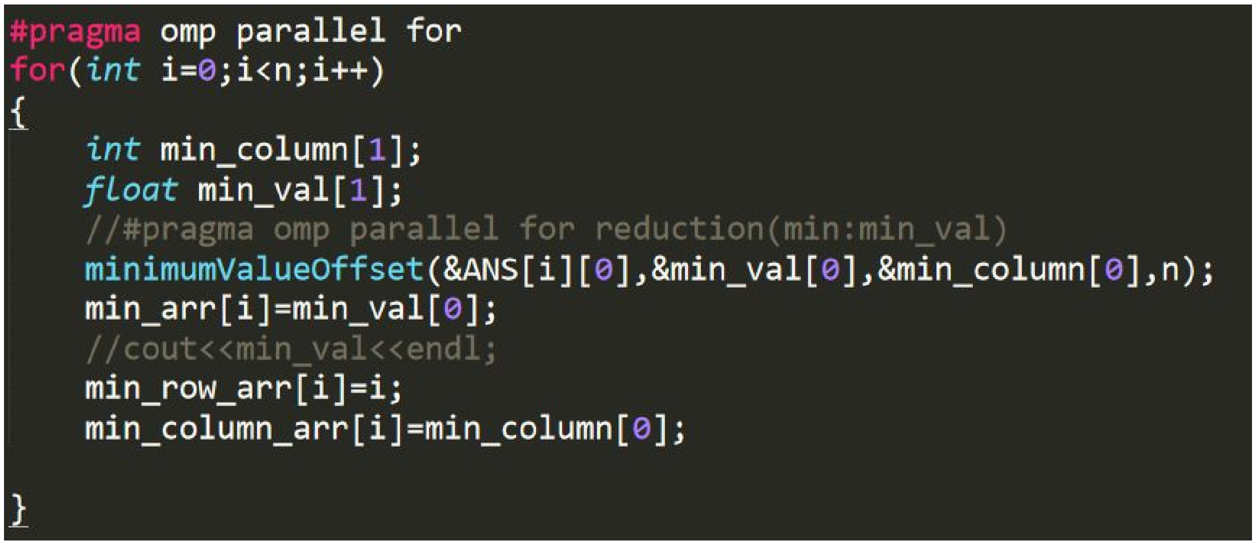
OPENMP program with ISPC SIMD:



```
#pragma omp parallel for

for(int i=0;i<n;i++)
{
    matrix_mul(n,i,&matrix_A[i][0],&B[0],&ANS[i][0]);

}
```

OPENMP program that calculates matrix multiplication with ISPC function.

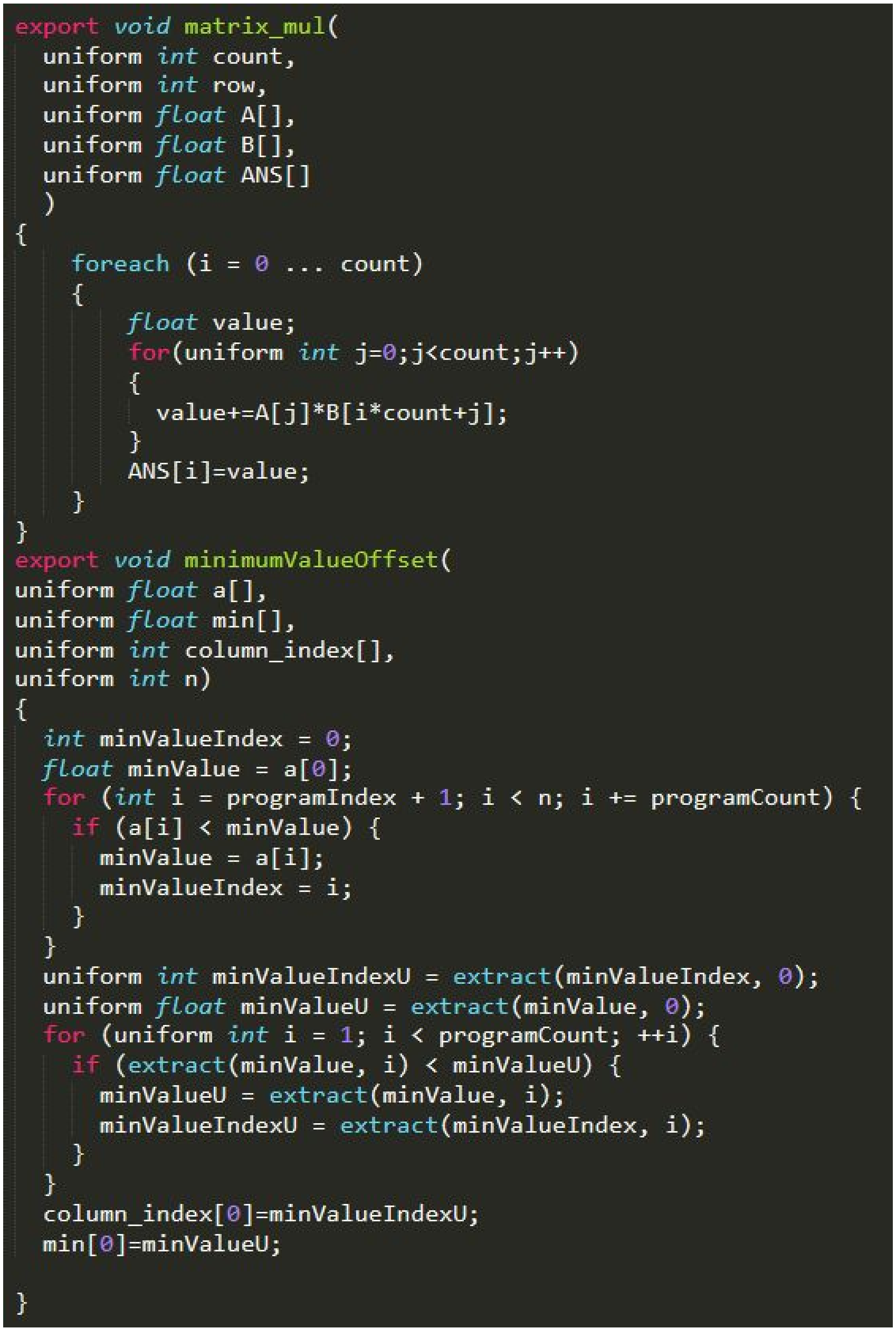
By giving the matrix multiplication calculation to the ISPC function, we can use SIMD to speed up the matrix multiplication even further.



```
#pragma omp parallel for
for(int i=0;i<n;i++)
{
    int min_column[1];
    float min_val[1];
    //#pragma omp parallel for reduction(min:min_val)
    minimumValueOffset(&ANS[i][0],&min_val[0],&min_column[0],n);
    min_arr[i]=min_val[0];
    //cout<<min_val<<endl;
    min_row_arr[i]=i;
    min_column_arr[i]=min_column[0];

}
```

OPENMP program that calculates minimum value and index for each row of the result matrix with ISPC function.

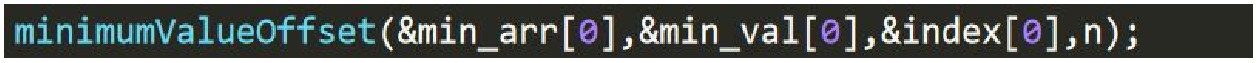
By giving the matrix multiplication calculation to the ISPC function, we can use SIMD to also speed up the minimum finding even further.



```
export void matrix_mul(
  uniform int count,
  uniform int row,
  uniform float A[],
  uniform float B[],
  uniform float ANS[]
  )
{
    foreach (i = 0 ... count)
    {
        float value;
        for(uniform int j=0;j<count;j++)
        {
          value+=A[j]*B[i*count+j];
        }
        ANS[i]=value;
    }
}
export void minimumValueOffset(
uniform float a[],
uniform float min[],
uniform int column_index[],
uniform int n)
{
  int minValueIndex = 0;
  float minValue = a[0];
  for (int i = programIndex + 1; i < n; i += programCount) {
    if (a[i] < minValue) {
      minValue = a[i];
      minValueIndex = i;
    }
  }
  uniform int minValueIndexU = extract(minValueIndex, 0);
  uniform float minValueU = extract(minValue, 0);
  for (uniform int i = 1; i < programCount; ++i) {
    if (extract(minValue, i) < minValueU) {
      minValueU = extract(minValue, i);
      minValueIndexU = extract(minValueIndex, i);
    }
  }
  column_index[0]=minValueIndexU;
  min[0]=minValueU;

}
```

ISPC functions for matrix multiplication and minimum finding.

The same ISPC functions are used in the pthread program with ISPC.



```
minimumValueOffset(&min_arr[0],&min_val[0],&index[0],n);
```

ISPC function that finds the minimum value and index

After we find minimum value and index for each row of the result matrix, we can call the minimum finding function again to find the global minimum value and index with ISPC SIMD.

Speed up table

Simple matrix multiplication	1795	1	
matrix multiplication with cache optimization	914.268	1.963319289	
matrix multiplication with cache optimization with ISPC	42.0983	42.63830131	
matrix multiplication with cache optimization with pthread(1)	822	2.183698297	1
matrix multiplication with cache optimization with pthread(2)	426.16	4.212033039	1.928853013
matrix multiplication with cache optimization with pthread(4)	213.44	8.409857571	3.8511994
matrix multiplication with cache optimization with pthread(20)	54.167	33.13825761	15.17529123
matrix multiplication with cache optimization with openmp(1)	689.32	2.604015552	1
matrix multiplication with cache optimization with openmp(2)	343.335	5.228129961	2.007718409
matrix multiplication with cache optimization with openmp(4)	175.965	10.20089222	3.917369932
matrix multiplication with cache optimization with openmp(20)	44.66	40.19256605	15.43484102
matrix multiplication with cache optimization with ISPC task(1)	34.602	51.87561413	1
matrix multiplication with cache optimization with ISPC task(2)	20.531	87.42876626	1.685353855
matrix multiplication with cache optimization with ISPC task(4)	11.051	162.4287395	3.131119356
matrix multiplication with cache optimization with ISPC task(20)	4.1686	430.6002015	8.300628508

matrix multiplication with cache optimization with pthread with ISPC(1)	33.029	54.34618063	1
matrix multiplication with cache optimization with pthread with ISPC(2)	18.8563	95.19364881	1.751616171
matrix multiplication with cache optimization with pthread with ISPC(4)	9.6146	186.6952343	3.43529632
matrix multiplication with cache optimization with pthread with ISPC(20)	4.0563	442.5215098	8.142642309
matrix multiplication with cache optimization with openmp with ISPC(1)	36.3583	49.36974501	1
matrix multiplication with cache optimization with openmp with ISPC(2)	19.7703	90.79275479	1.839036332
matrix multiplication with cache optimization with openmp with ISPC(4)	10.314	174.0352918	3.525140586
matrix multiplication with cache optimization with openmp with ISPC(20)	3.6363	493.6336386	9.998707477
	second	speed up	speed up(multithreading/multitasking)

Some points: Amdahl's law shows its power in multithreading/multitasking programs, with 20 threads, we only have around 8 to 15 speeds up. And one good feature for openmp is that the user doesn't need to write their own partition algorithm.

1. What is the best way (program) to speed up the execution of the benchmark program? What is the total speedup compared to sequential version S?

A:The best result is from the openmp program with ISPC, with a speedup of roughly 493 times compared to the sequential program.

2. For the best program vs. sequential version S, what are the speedup factors due to

- o cache data optimization?
- o SIMD extensions?
- o multithreading/multitasking?

A:Theoretically, if we just multiply the speedup of each technique, we should get the speedup, if we do that, cache optimization will give us a speed up of roughly 1.96, SIMD speed up of 21.7, and openmp just runs 1.32 time faster with one thread compares to sequential program with cache optimization, and 20 thread speed up of 10 with SIMD, the result speedup should be 1.96*21.7*1.32*10=561, which is close to our practical result of 493.

https://matrix.reshish.com/multiplication.php is used to help me verify my matrix multiplication part of the program.