GLOpenCL: Compiler and Run-Time Support for OpenCL on Hardware- and Software-Managed Cache Multicores

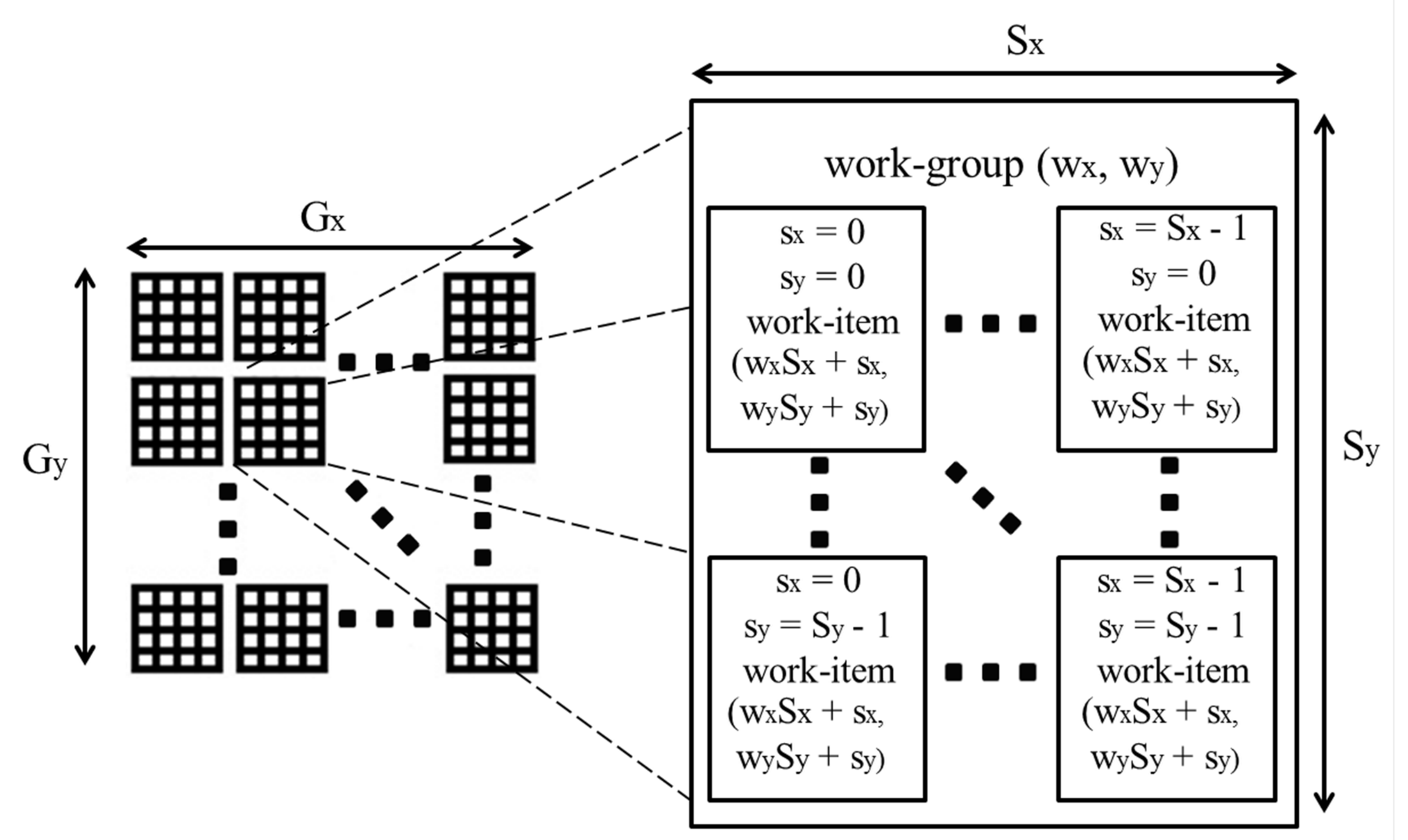
Konstantis Daloukas, Christos D. Antonopoulos and Nikolaos Bellas
University of Thessaly, Greece

1 Introduction - Motivation

- Technology advances have enabled the development of a plethora of computation substrates for parallel, high performance computing.
  - A paradigm shift towards parallel programming.
- A multitude of programming models have been proposed that aim at handling most of the technicalities, thus allowing the programmer to focus on algorithmic issues. However, none of them target both homogeneous and heterogeneous multicores as well as accelerators.
- **OpenCL** presents itself as a unified programming standard and framework for heterogeneous multicores that integrate both CPUs and accelerators, such as the Cell BE or GPUs. Although, currently available, vendor specific implementations target either only multicores or accelerators, due to their vast architectural differences.
- **Motivation**: Design and develop **GLOpenCL**, a unified framework to enable native execution of OpenCL applications on both hardware- as well as software-controlled cache multicores. The framework comprises a compiler and a run-time library that shares the same basic architecture across all platforms.

2 OpenCL Programming Model

- OpenCL models the underlying parallel architecture as a host and a number of compute devices. Each compute device integrates a number of compute units, each one divided into processing elements.



- Efficient exploitation and mapping of the available parallelism is not a trivial task.

3 Compiler Support

- Source-to-source transformations that aim at coarsening the granularity of a kernel function from a per-logical-thread to a per-work-group basis. These transformations enable an efficient mapping of the available parallelism on multicores without hardware support for fine-grained threading.
  - Logical thread serialization has the additional implicit effect of potentially reducing the local storage that is required by each work-item.

Logical Thread Serialization

```
__kernel void BlackScholes(
   __global float *d_Call,
   __global float *d_Put,
   __global float *d_S,
   __global float *d_X,
   __global float *d_T,
   float R, float V,
   unsigned int optN ){

   unsigned int opt;
   for(opt = get_global_id(0); opt < optN;
       opt += get_global_size(0))
     BlackScholesBody(
        &d_Call[opt],
        &d_Put[opt],
        d_S[opt],
        d_X[opt],
        d_T[opt],
        R,
        V
     );
}
```

```
__kernel void BlackScholes (
   __global float *d_Call,
   __global float *d_Put,
   __global float *d_S,
   __global float *d_X,
   __global float *d_T,
   float R, float V,
   unsigned int optN ) {

  int ___kernel_indices[3];
  unsigned int opt;

  ___kernel_indices[2] = 0;
  while (___kernel_indices[2] < get_local_size (2)) {
   ___kernel_indices[1] = 0;
   while (___kernel_indices[1] < get_local_size (1)) {
     ___kernel_indices[0] = 0;
     while (___kernel_indices[0] < get_local_size (0)) {

       for (opt = get_global_id (0); opt < optN;
            opt += get_global_size (0) )
          BlackScholesBody (&d_Call[opt], &d_Put[opt],
               d_S[opt], d_X[opt], d_T[opt], R, V);

       ___kernel_indices[0]++;
     }
     ___kernel_indices[1]++;
   }
   ___kernel_indices[2]++;
  }
}
```

Elimination Of Synchronization Operations

```
__kernel void transpose(__global float *odata, __global float *idata,
                        int width, int height, __local float* block) {

     // read the matrix tile into shared memory
     unsigned int xIndex = get_global_id(0);
     unsigned int yIndex = get_global_id(1);
     unsigned int index_in, index_out;
     if((xIndex < width) && (yIndex < height)){
        index_in = yIndex * width + xIndex;
        block[get_local_id(1)*(BLOCK_DIM+1)+get_local_id(0)] =
         idata[index_in];
     }

     barrier(CLK_LOCAL_MEM_FENCE);

     // write the transposed matrix tile to global memory
     xIndex = get_group_id(1) * BLOCK_DIM + get_local_id(0);
     yIndex = get_group_id(0) * BLOCK_DIM + get_local_id(1);
     if((xIndex < height) && (yIndex < width)){
        index_out = yIndex * height + xIndex;
        odata[index_out] =
         block[get_local_id(0)*(BLOCK_DIM+1)+get_local_id(1)];
     }
}
```

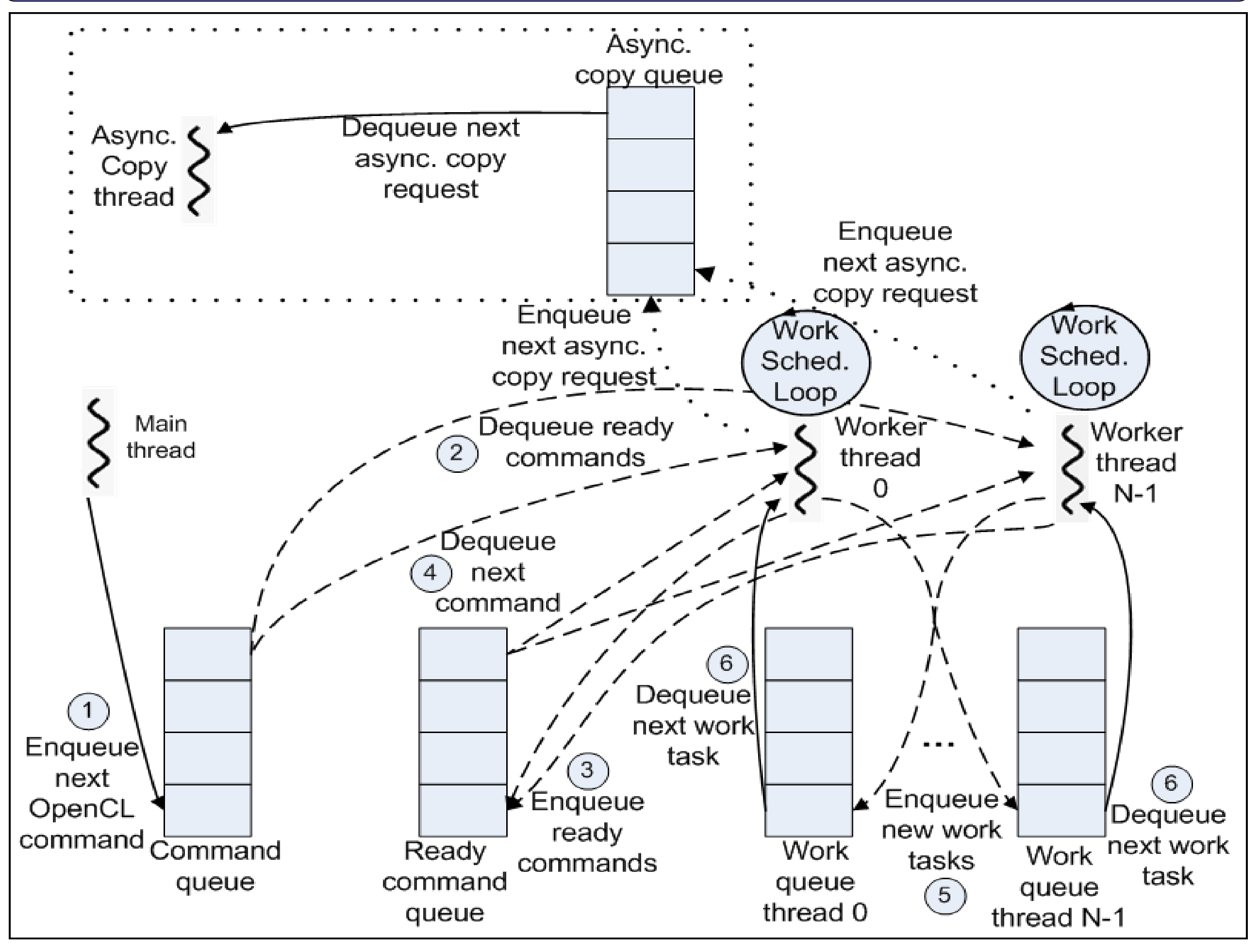
```
__kernel void transpose (__global float *odata, __global float *idata,
                         int width, int height, __local float *block) {

    int ___kernel_indices[3];
    unsigned int xIndex, yIndex, index_in, index_out;

    triple_nested_loop {
       xIndex = (get_global_id (0) + ___kernel_indices[0]);
       yIndex = (get_global_id (1) + ___kernel_indices[1]);
       if ((xIndex < width) && (yIndex < height)) {
          index_in = yIndex * width + xIndex;
          block[get_local_id(1)*(BLOCK_DIM+1)+get_local_id(0)] =
           idata[index_in];
       }
    }
    //barrier (CLK_LOCAL_MEM_FENCE);
    triple_nested_loop {
       xIndex = get_group_id (1) * BLOCK_DIM + get_local_id (0);
       yIndex = get_group_id (0) * BLOCK_DIM + get_local_id (1);

       if ((xIndex < height) && (yIndex < width)) {
          index_out = yIndex * height + xIndex;
          odata[index_out] =
           block[get_local_id(0)*(BLOCK_DIM+1)+get_local_id(1)];
       }
    }
}
```

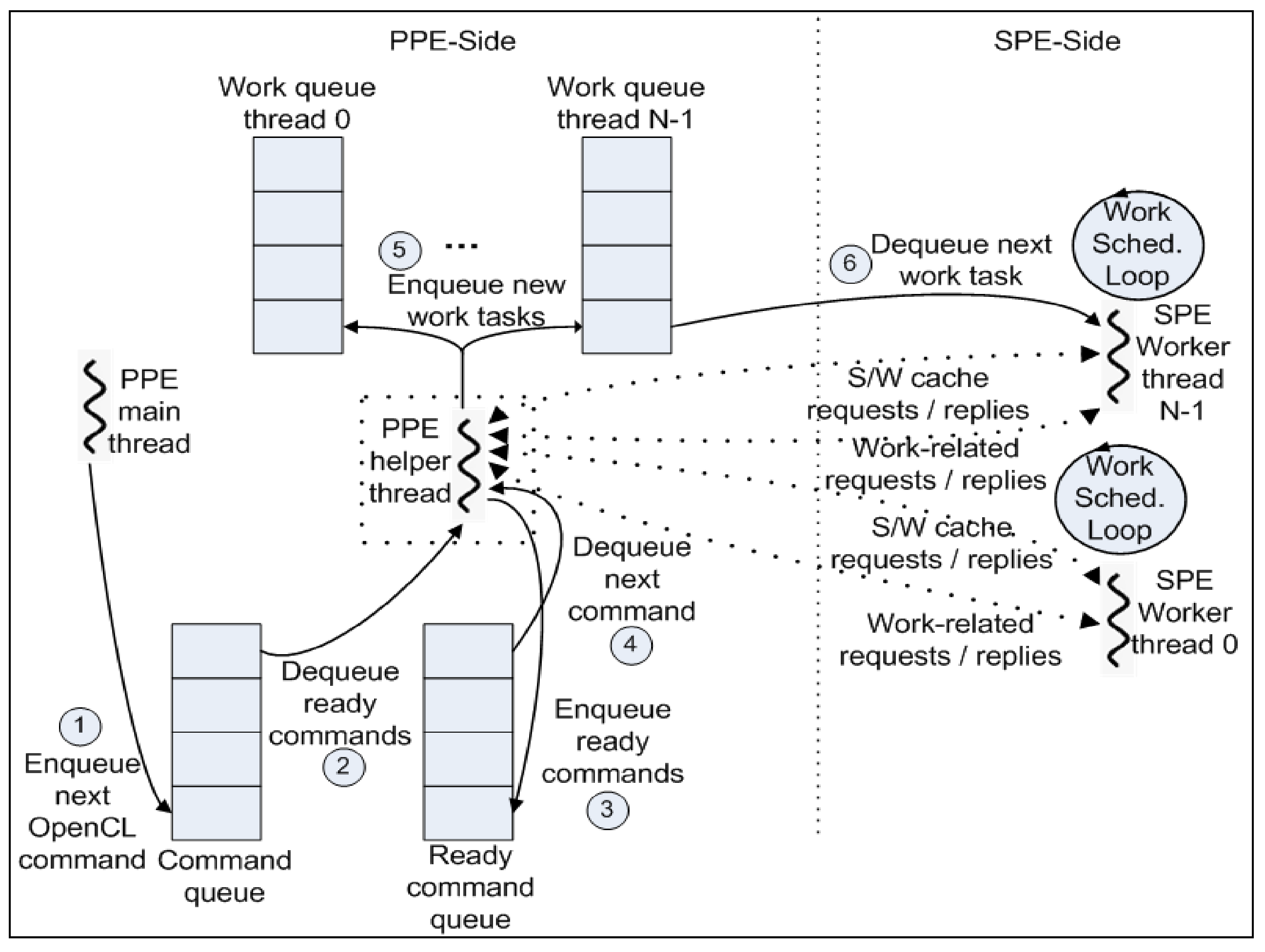
Variable Privatization

```
__kernel void matrixMul( __global float *A, __global float *B, __global float *C,
                         __local float *As, __local float *Bs, int Awidth, int Bwidth ) {

     int a, b, c, aEnd, k;
     float Csub;

     triple_nested_loop {
          aEnd = Awidth * get_local_id(1) + get_local_id(0);
          a = Awidth * BLOCK_SIZE * get_group_id(1) + aEnd;
          b = a + BLOCK_SIZE * get_group_id(0);
          aEnd = a + Awidth;
          Csub = 0;
     }
     while( a < aEnd ) {
          triple_nested_loop {
               As[ get_local_id(1) * BLOCK_SIZE + get_local_id(0) ] = A[a];
               Bs[ get_local_id(1) * BLOCK_SIZE + get_local_id(0) ] = B[b];
               a += BLOCK_SIZE;
               b += BLOCK_SIZE * Bwidth;
          }
          triple_nested_loop {
               for( k = 0; k < BLOCK_SIZE; k++ ) {
                    Csub += As[ get_local_id(1) * BLOCK_SIZE + k ] *
                         Bs[ k * BLOCK_SIZE + get_local_id(0) ];
               }
          }
     }
     triple_nested_loop {
          c = get_global_id(1) * get_global_size(0) + get_global_id(0);
          C[c] = Csub;
     }
}
```

```
__kernel void matrixMul( __global float *A, __global float *B, __global float *C,
                         __local float *As, __local float *Bs, int Awidth, int Bwidth ) {

     int a[], b[], aEnd[], c, k;
     float Csub[];

     triple_nested_loop {
          aEnd[thread_id] = Awidth * get_local_id(1) + get_local_id(0);
          a[thread_id] = ( Awidth * BLOCK_SIZE ) * get_group_id(1) + aEnd[thread_id];
          b[thread_id] = a[thread_id] + BLOCK_SIZE * get_group_id(0);
          aEnd[thread_id] = a[thread_id] + Awidth;
          Csub[thread_id] = 0;
     }
     while( a[thread_id] < aEnd[thread_id] ) {
          triple_nested_loop {
               As[ get_local_id(1) * BLOCK_SIZE + get_local_id(0) ] = A[a[thread_id]];
               Bs[ get_local_id(1) * BLOCK_SIZE + get_local_id(0) ] = B[b[thread_id]];
               a[thread_id] += BLOCK_SIZE;
               b[thread_id] += BLOCK_SIZE * Bwidth;
          }
          triple_nested_loop {
               for( k = 0; k < BLOCK_SIZE; k++ ) {
                    Csub[thread_id] += As[ get_local_id(1) * BLOCK_SIZE + k ] *
                         Bs[ ( k * BLOCK_SIZE ) + get_local_id(0) ];
               }
          }
     }
     triple_nested_loop {
          c = get_global_id(1) * get_global_size(0) + get_global_id(0);
          C[c[thread_id]] = Csub[thread_id];
     }
}
```

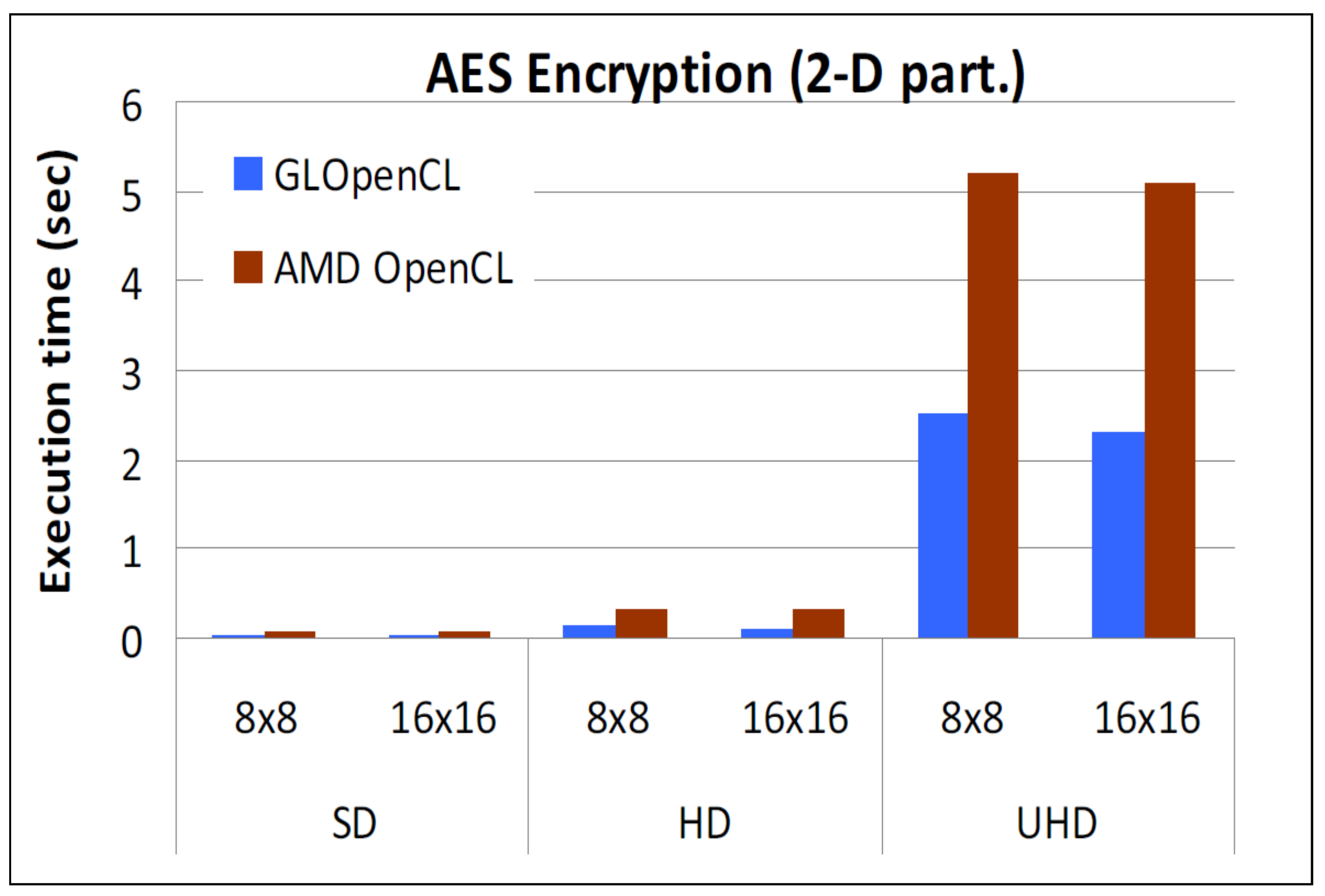
4 Run-time Support

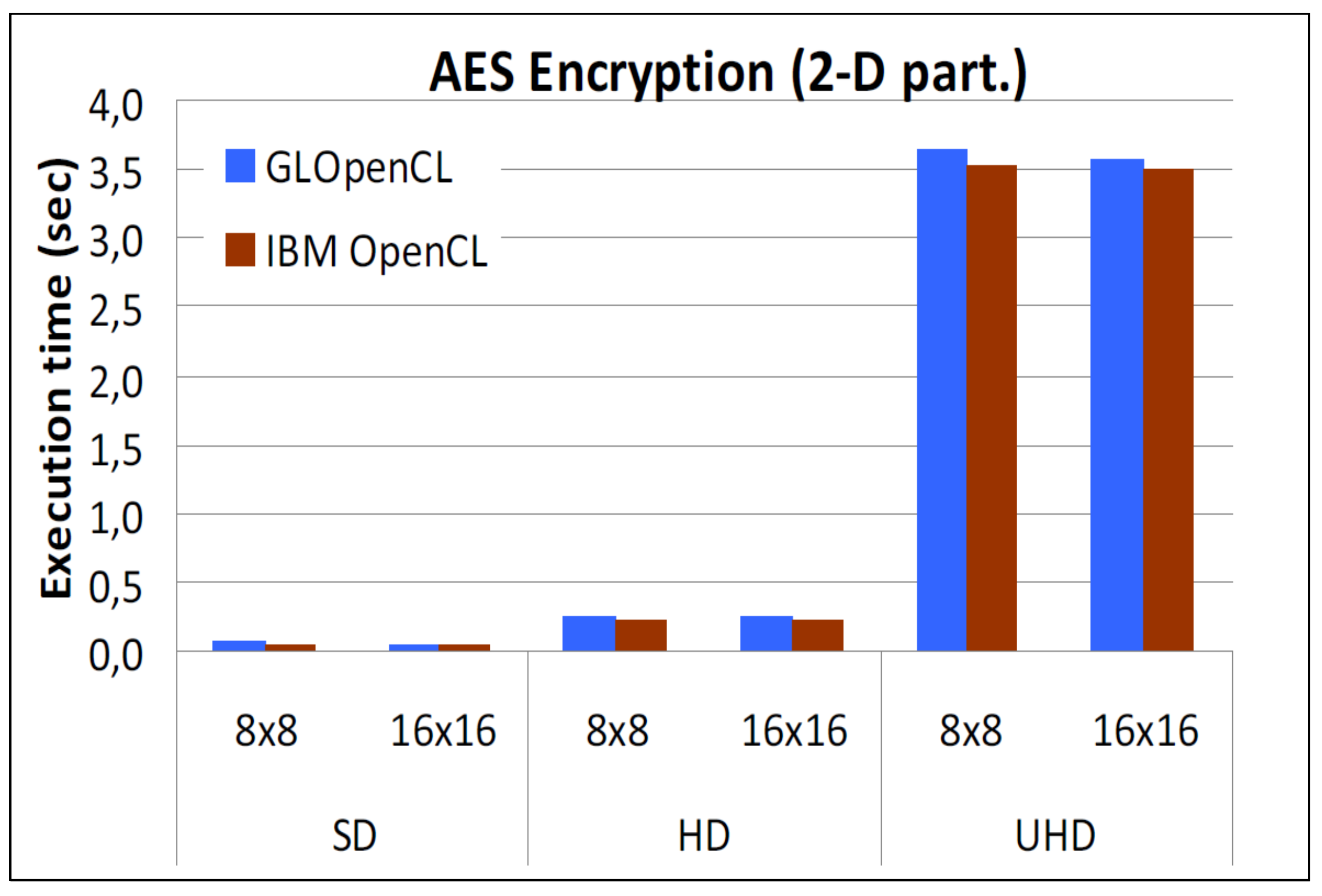




5 Performance Evaluation

- Evaluate the framework's performance using a series of six benchmark applications on the Intel E5520 i7 and the Cell BE processors, and compare with the SDKs provided by ATI/AMD and IBM.


AES Encryption (2-D part.)
Execution time (sec)
GLOpenCL
AMD OpenCL
8x8
16x16
SD
HD
UHD


AES Encryption (2-D part.)
Execution time (sec)
GLOpenCL
IBM OpenCL
8x8
16x16
SD
HD
UHD