Chapter 16
Fast Fourier Transform (FFT) on GPUs

Yash Ukidave, Gunar Schirner, and David Kaeli

16.1 Introduction to FFT

The Fast Fourier Transform (FFT) is one of the most common algorithms used in signal processing to transform a signal from the time domain to the frequency domain and vice-versa. It is named after the French mathematician and physicist named Joseph Fourier.

The FFT is a fast and efficient algorithm to compute the discrete fourier transform (DFT) and the inverse discrete fourier transform (IDFT). There are many distinct FFT algorithms involving a wide range of mathematics, from simple complex-number arithmetic, to group theory and number theory. Considerable research effort has been devoted towards optimizing FFT algorithms over the past four decades. The algorithm presented by Cooley and Tukey [2] reduced the algorithmic complexity when computing the DFT to $O(NlogN)$ from $O(N^2)$, which is viewed as a turning point for applications using the fourier transform.

16.2 How Does the FFT Work?

Every discrete function $f(x)$ of finite length can be described as a set of potentially infinite sinusoidal functions. This representation is known as the frequency domain representation of the function, defined as $F(u)$. The relationship between these two functions is represented by the Discrete Fourier Transform (DFT).

Y. Ukidave • G. Schirner • D. Kaeli (✉)
Northeastern University, Boston, MA, USA
e-mail: yukidave@ece.neu.edu; schirner@ece.neu.edu; kaeli@ece.neu.edu

V. Kindratenko (ed.), *Numerical Computations with GPUs*,
DOI 10.1007/978-3-319-06548-9_16, © Springer International Publishing Switzerland 2014

$$\mathscr{F}\{f(x)\} = F(u) = \sum_{x=0}^{N-1} f(x) W_N^{ux} \tag{16.1}$$

$$\mathscr{F}^{-1}\{F(u)\} = f(x) = \frac{1}{N} \sum_{u=0}^{N-1} F(u) W_N^{-ux} \tag{16.2}$$

where $W_N = e^{\frac{-j2\pi}{N}}$ is the *Twiddle Factor* and N is the number of input data points.

Equation (16.1) defines the forward DFT computation over a finite time signal $f(x)$ and the inverse DFT computation over a frequency function $F(u)$, as shown in Eq. (16.2).

The Fast Fourier Transform refers to a class of algorithms that uses divide-and-conquer techniques to efficiently compute the Discrete Fourier Transform of the input signal. For a one-dimensional (1D) array of input data size N, the FFT algorithm breaks the array into a number of equal-sized sub-arrays and performs computation on these sub-arrays. The process of dividing the input sequence in the FFT algorithm is called decimation. Two major classifications of the FFT algorithm cover all the variations of the computation based on the technique of decimation.

- **Decimation in Time (DIT)**:
  The decimation in time (DIT) algorithm splits the N-point data sequence into two $\frac{N}{2}$-point data sequences $f_1(N)$ and $f_2(N)$, for even and odd numbered input samples, respectively.
- **Decimation in Frequency (DIF)**:
  The decimation in frequency (DIF) algorithm splits the N-point data sequence into two sequences, of first $\frac{N}{2}$ data points and last $\frac{N}{2}$ data points respectively. The DIF algorithm does not consider the decimation as even and odd data points.

In this process of decimation, we exploit both the symmetry and the periodicity of the complex exponential $W_N = e^{\frac{-j2\pi}{N}}$, known as the *Twiddle Factor*. The algorithm using the divide-and-conquer technique was proposed by Cooley and Tukey. Algorithm 1 shows the pseudocode of the basic Cooley-Tukey FFT.

The Radix-2 Cooley-Tukey algorithm recursively divides the N-point DFT into two N/2-point DFTs, with a complex multiplication (the Twiddle Factor) in between. The division is done into two half length DFTs with even indexed and odd indexed samples, as described in Eq. (16.3). The recursive division continues until we achieve N 2-point signals.

$$\mathscr{F}\{f(x)\} = F(u) = \sum_{x(even)} F(x) W_N^{ux} + \sum_{x(odd)} F(x) W_N^{ux} \tag{16.3}$$

The butterfly computation transforms two complex input points to two complex output points to compute the FFT. The 2-point, Radix-2 FFT butterfly is derived by expanding the formula of the DFT for two point signal (N = 2), as shown in Eqs. (16.4)–(16.6). The 2-point, Radix-2 butterfly is also shown in Fig. 16.1.

Algorithm 1 Cooley-Tukey Radix-2 FFT

```
Require: Input data A[0,..., $n-1$]
Recursive-FFT(A)
$n = length[A]$
if n = 1
    return $A$
end
$w_n = e^{2i\pi/n}$
$w = 1$
$A^{even} = (A[0], A[2], \ldots, A[n-2])$
$A^{odd} = (A[1], A[3], \ldots, A[n-1])$
$y^{even}$ = Recursive-FFT($A^{even}$)
$y^{odd}$ = Recursive-FFT($A^{odd}$)
for k = 0 → $\frac{n}{2}$ - 1
    $y_k = y_k^{even} + wy_k^{odd}$
    $y_{k+n/2} = y_k^{even}$ - $wy_k^{odd}$
    $w = ww_n$
end
return $y$
```

$$\mathscr{F}\{f(x)\} = F(u) = \sum_{x=0}^{1} f(x) W_2^{ux} \tag{16.4}$$

$$F(0) = f(0) + f(1) \tag{16.5}$$

$$F(1) = f(0) - f(1) \tag{16.6}$$

16.2.1 FFT as a Heterogeneous Application

Graphics Processing Units (GPUs) have been effectively used for accelerating a number of general-purpose computation. The high performance community has been able to effectively exploit the inherent parallelism on these devices, leveraging their impressive floating-point performance and high memory bandwidth of GPU. FFTs were one of the first algorithms ported to GPU [3]. The original GPU-based FFT code was implemented as a shader code executed in the graphics pipeline. Since

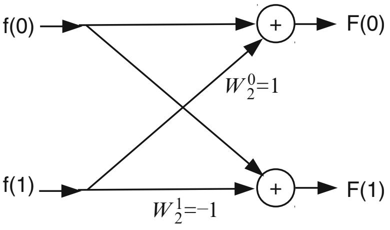


Fig. 16.1 Radix-2 FFT butterfly for a 2-point computation. The butterfly requires two complex additions and two complex multiplications

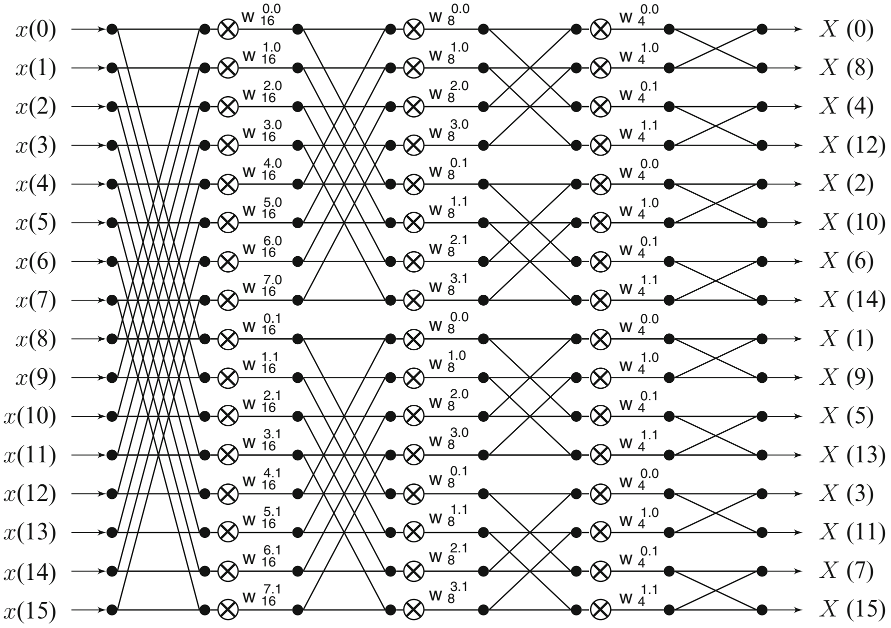


Fig. 16.2 Radix-2 based DIF FFT with 16 input data points

this time, GPU computing has moved forward aggressively, overcoming many of the limitations of programming in a shader-based language. The basic structure of the FFT makes it a good candidate for parallel execution on heterogeneous platforms. The use of heterogeneous programming models available in OpenCL and CUDA simplify the porting of mathematical computations such as FFT to a GPU [4, 6].

The FFT is a non-causal algorithm, since it is solely dependent of the data inputs. Thus, the FFT can be evaluated in parallel over all N input points by forming smaller subsequences of the problem. Each subsequence carries out the required number of FFT compute stages, which can be mapped to different computational cores (i.e., compute units) on the heterogeneous device. Figure 16.2 shows the computation of a Radix-2 DIF FFT, computed over 16 input data points. The input data points are transformed over four stages of compute using the FFT algorithm. This computation can be parallelized effectively on a GPU by creating many threads, each of which works on 2 input points at a given time for every stage. For the example presented in Fig. 16.2, the problem can be divided over stages using 8 (N/2) threads. For a large input data set, the FFT can be computed using a large number of threads. Thus, the parallel hardware present on a GPU can be effectively leveraged to efficiently compute the FFT.

Heterogeneous implementations of the FFT algorithm can be designed using OpenCL programming model. OpenCL provides functionality to represent the FFT computation for each work-item (i.e., thread). The memory model of OpenCL allows the programmer to map the entire input data for the FFT on the device for parallel execution on the available compute units (CUs) on the GPU.

16.3 Implementing FFT on GPU Using OpenCL

Next, we discuss a Radix-2 FFT implementation using the Cooley-Tukey algorithm [2]. We focus on the Radix-2 variant of the Cooley-Tukey algorithm due to its ease of implementation and numerical accuracy. The strategies presented here can be easily extended for higher-Radix cases.

Based on the structure of the Radix-2 Cooley-Tukey algorithm, we will design a OpenCL kernel for the FFT computation. We begin by describing our methodology to decompose the FFT into a number of simpler computations. Then we can use these simple operations as building blocks to produce the final result.

The code block shown in Sect. 16.2 presented an implementation of a Radix-2 FFT computation using a 2-point butterfly. The kernel takes as input complex inputs, which are represented using `float2`, a vector data type provided in OpenCL. The `x` and `y` components of the `float2` vector represent the *real* and *imaginary* parts of the complex number, respectively. The kernel retrieves 2 input data points from global memory of the GPU and performs a 2-point FFT computation over them.

```
__kernel void FFT_2_pt( __global float2* d_in,
      __global float2* d_out)
{
    int gid = get_global_id(0);  // Global id of thread
    float2  in0, in1;
    in_0 = d_in[gid];            // Input data #0
    in_1 = d_in[gid+1];          // Input data #1

    /* 2-point FFT computation begins */
    float2  Var;
    Var = in_0;
    in_0 = Var + in_1;
    in_1 = Var - in_1;
    /* 2-point FFT computation ends */

    d_out[gid] = in_0;           // Output data #0
    d_out[gid+1] = in_1;         // Output data #1
}
```

The 2-point FFT butterfly computation can be extended to a 4-point butterfly representation. The 4-point butterfly is shown in Fig. 16.3a. To produce results in the proper order, the input data points of the FFT computation are bit-reversed, as seen in Fig. 16.3a. To obtain coalesced memory accesses in the 4-point butterfly, the 2-point butterfly structures have to be rearranged in order to preserve spatial locality in the address stream, as shown in Fig. 16.3b. The 4-point FFT can compute two stages of the Radix-2 computation in one single pass. Hence, an 8-point FFT can be implemented using one pass of a 4-point FFT computation followed by one pass

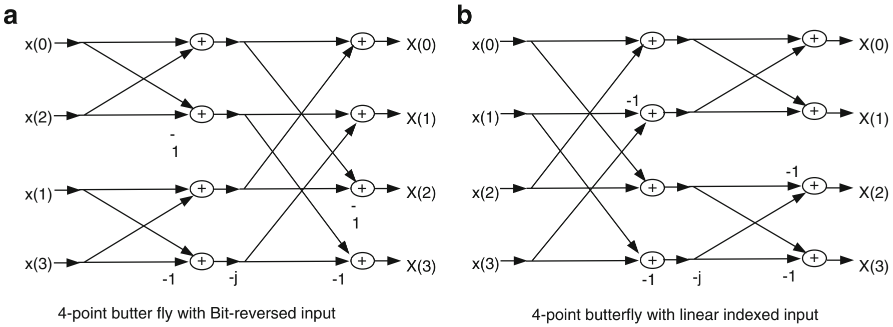


Fig. 16.3 A Radix-2 FFT using a 4-point butterfly with (**a**) bit-reversed input indexes and (**b**) linear input indexes

0	4	2	6	1	5	3	7

Fig. 16.4 Bit-reversed input indexes of a 8-point FFT

Thread 0				Thread 1			
0	2	4	6	1	3	5	7

Fig. 16.5 Input data mapping of a 8-point FFT computation to threads of a 4-point FFT kernel

of a 2-point FFT computation. If the computation is performed using only a 2-point FFT, it would require 3 passes of the 2-point FFT kernel. This would increase the kernel launch overhead on GPU and would also increase the number of accesses to global memory. Both of these factors can cause performance degradation. Hence, a large-scale FFT is computed using combinations of small FFTs.

To obtain coalesced memory accesses, the inputs to the FFT kernels should be indexed in a linear order. We have to develop a generic indexing scheme for all FFT kernels, since we want this code to work for different input sizes. We consider the example of the 8-point FFT computation using 4-point FFT and 2-point FFT kernels. The first two stages of the compute are implemented with 2 threads using a 4-point FFT. The last stage of the compute is implemented using 4 threads each computing a 2-point FFT. The inputs for a bit-reversed 8-point FFT are shown in Fig. 16.4.

Figure 16.5 shows how inputs are mapped to the two threads that compute the 4-point FFT to implement the 8-point FFT. We observe that the first data element required by each thread is located at an input index equal to the global id (gid) of the particular thread. We define a set of parameters in the kernel code to help us develop the desired indexing.

1. ***gid***: Global Id of a thread
2. ***gbl_size***: Work size (Total number of threads)
3. ***N***: Number of input data points(complex)

4. ***out_id*** : The index for each output data point
5. ***num_stage*** : Parameter that defines output index
6. ***kern_size***: Kernel Size

We use these parameters and our observations of the 4-point FFT to develop the input indexing of a 4-point FFT kernel. The code section for the input indexing is given below:

```
int gid = get_global_id(0);
int gbl_size = N/4;
float2 in_0, in_1, in_2, in_3;
uint kern_size = 4;

in_0 = d_in[(0*gbl_size)+gid];
in_1 = d_in[(1*gbl_size)+gid];
in_2 = d_in[(2*gbl_size)+gid];
in_3 = d_in[(3*gbl_size)+gid];
```

The output index order of the kernel needs to be sequential. To insure preserving spatial locality, we develop a generic indexing scheme for storing output data. The output index (`out_id`) is computed using the parameters defined above. The `num_stage` parameter must be set by the host code while executing the kernel. The value of `num_stage` is $2^{Stages completed by FFT}$ (i.e., the value of `num_stage` after the completion of the 4-point kernel is 4). After careful study of the 2-point and 4-point FFT computations, the following observations are made:

- In the first stage of the FFT computation, the output index `out_id` of each thread is computed as the product of the kernel-size(`kern_size`) and global id `gid` of that thread.
- For every stage of the FFT computation, each thread writes `kern_size` output points to the global memory with a stride equal to `num_stage`
- For the last FFT stage, the output index `out_id` of each work item is equal to the global id `gid` of the particular thread.

The value of the `out_id` can thus be calculated as follows for each FFT kernel:

```
out_id = (gid / num_stage) * num_stage * kern_size
         + (gid%num_stage);
```

A key portion of the FFT computation is the calculation of the Twiddle factor, which was defined in Sect. 16.2. The Twiddle factor depends on the size of the FFT computation and has to be computed at runtime during kernel execution.

$$W_N^{kn} = e^{\frac{-j2kn\pi}{N}} = cos\left(\frac{-2kn\pi}{N}\right) + jsin\left(\frac{-2kn\pi}{N}\right) \qquad (16.7)$$

The code used for computing the Twiddle factor using Eq. (16.7) is given below:

```
void twidle_factor(int k, float angle, float2 ip)
{
      float2 twiddle, var;
      twiddle.x = native_cos(k*angle);
      twiddle.y = native_sin(k*angle);

      var.x = twiddle.x*ip.x - twiddle.y*ip.y;
      var.y = twiddle.x*ip.y + twiddle.y*ip.x;

      ip.x = var.x;
      ip.y = var.y;
}
```

Twiddle factors are sinusoidal components, and are symmetrical by definition. Hence, developers can choose to hardcode the values of the Twiddle factors required by the FFT computation in the OpenCL kernel code.

The kernel code for the 2-point, 4-point and 8-point kernels is given below. The kernels are specialized implementations and use the input and output indexing scheme as described earlier.

Kernel Code for 2-Point FFT:

```
void FFT2_comp(float2 in_0, float2 in_1)
{
  float2 v0;
  v0 = in0;
  in0 = v0 + in1;
  in1 = v0 - in1;
}


__kernel void FFT_MS_2(__global float2* d_in,
                       __global float2* d_out,
                       __global uint* comp_stage,
                       uint num_data)
{
  uint N = num_data;
  uint gid = get_global_id(0);
  uint gbl_size = N/2;
  float2 in_0, in1;
  uint kern_size = 2;

  in_0 = d_in[(0*gbl_size)+gid];
  in_1 = d_in[(1*gbl_size)+gid];

  uint num_stage = comp_stage[0];
  if (num_stage!=1)
  {
    float angle = -2*PI*(gid)/(N);
    twidle_factor(1, angle, in1);
  }

  FFT2_comp(in_0, in_1);

  uint I_dout = (gid/num_stage)*num_stage*kern_size
  +(gid%num_stage);
  d_out[(0*num_stage)+I_dout] = in_0;
  d_out[(1*num_stage)+I_dout] = in_1;

}
```

Kernel Code for 4-Point FFT:

```
void FFT4_comp(float2 in_0, float2 in_1, float2 in_2,
float2 in_3) {
float2 v_0, v_1, v_2, v_3;
v_0 = in_0 + in_2;
v_1 = in_1 + in_3;
v_2 = in_0 - in_2;
v_3.x = in_1.y - in_3.y;
v_3.y = in_3.x - in_1.x;
in_0 = v_0 + v_1;
in_2 = v_0 - v_1;
in_1 = v_2 + v_3;
in_3 = v_2 - v_3;
}


__kernel void  FFT_MS_4( __global float2* Data_in,
      __global float2* Data_out,
      __global uint* comp_stage,
      uint num_data)
{
  uint N = num_data;
  int gid = get_global_id(0);
  int gbl_size = N/4;
  float2 in_0, in_1, in_2, in_3;
  uint kern_size = 4;
  in_0 = d_in[(0*gbl_size)+gid];
  in_1 = d_in[(1*gbl_size)+gid];
  in_2 = d_in[(2*gbl_size)+gid];
  in_3 = d_in[(3*gbl_size)+gid];
  uint num_stage = comp_stage[0];
  if (num_stage!=1)
  {

  float angle = -2*PI*(gid)/(N);
  twidle_factor(1, angle, in_1);
  twidle_factor(2, angle, in_2);
  twidle_factor(3, angle, in_3);
      }

  FFT4(in_0, in_1, in_2, in_3);
  uint I_dout = (gid/num_stage)*num_stage*kern_size+
  (gid%num_stage);
  d_out[(0*num_stage)+I_dout] = in_0;
  d_out[(1*num_stage)+I_dout] = in_1;
  d_out[(2*num_stage)+I_dout] = in_2;
  d_out[(3*num_stage)+I_dout] = in_3;

}
```

Kernel Code for 8-Point FFT:

```
void FFT8(in_0, in_1, in_2, in_3, in_4, in_5, in_6,
in_7) {
float2 v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7;
float2 s0, s1, s2, s3, s4, s5, s6, s7;
v_0 = in_0 + in_4;
v_1 = in_1 + in_5;
v_2 = in_2 + in_6;
v_3 = in_3 + in_7;
v_4 = in_0 - in_4;
v_5 = in_1 - in_5;
v_6.x = in_2.y - in_6.y;
v_6.y = in_6.x - in_2.x;
v_7.x = in_3.y - in_7.y;
v_7.y = in_7.x - in_3.x;
s0 = v_0 + v_2;
s1 = v_1 + v_3;
s2 = v_0 - v_3;
s3 = v_3 - v_1;
s4 = v_4 + v_6;
s5.x = v_5.y - v_7.y;
s5.y = v_7.x - v_5.x;
s6 = v_3 - v_6;
```

```
s7.x = v_7.y - v_5.y;
s7.y = v_5.x - v_7.x;
in_0 = s0 + s1;
in_1 = s0 - s1;
in_2 = s2 + s3;
in_3 = s2 - s3;
in_4 = s4 + s5;
in_5 = s4 - s5;
in_6 = s6 + s7;
in_7 = s6 - s7;
}


__kernel void  FFT_MS_8( __global float2* d_in,
      __global float2* d_out,
      __global uint* comp_stage,
      uint num_data)
{
  uint N = num_data;
  int gid = get_global_id(0);
  int gbl_size = N/8;
  float2 in_0, in_1, in_2, in_3, in_4, in_5, in_6, in_7;

  uint num_stage = comp_stage[0];
  uint kern_size = 8;
  in_0 = d_in[(0*gbl_size)+gid];
  in_1 = d_in[(1*gbl_size)+gid];
  in_2 = d_in[(2*gbl_size)+gid];
  in_3 = d_in[(3*gbl_size)+gid];
  in_4 = d_in[(4*gbl_size)+gid];
  in_5 = d_in[(5*gbl_size)+gid];
  in_6 = d_in[(6*gbl_size)+gid];
  in_7 = d_in[(7*gbl_size)+gid];

  if (num_stage!=1)
    {
    float angle = -2*PI*(gid%num_stage)/(num_stage*
    kern_size);
    twidle_factor(1, angle, in_1);
    twidle_factor(2, angle, in_2);
    twidle_factor(3, angle, in_3);
    twidle_factor(4, angle, in_4);
    twidle_factor(5, angle, in_5);
    twidle_factor(6, angle, in_6);
    twidle_factor(7, angle, in_7);
    }
  FFT8_comp(in_0, in_1, in_2, in_3, in_4, in_5, in_6,
  in_7);
  uint I_dout = (gid/num_stage)*num_stage*kern_size+
  (gid%num_stage);

  d_out[(0*num_stage)+I_dout] = in_0;
  d_out[(1*num_stage)+I_dout] = in_1;
  d_out[(2*num_stage)+I_dout] = in_2;
  d_out[(3*num_stage)+I_dout] = in_3;
  d_out[(4*num_stage)+I_dout] = in_4;
  d_out[(5*num_stage)+I_dout] = in_5;
  d_out[(6*num_stage)+I_dout] = in_6;
  d_out[(7*num_stage)+I_dout] = in_7;
}
```

When using hard-coded FFT kernels (e.g., 16-point and 32-point) for large data sets, each kernel will require a large number of registers per thread on the GPU. This can cause a reduction in number of active threads scheduled on the GPU due to unavailability of resources, which leads to degradation in execution performance of the application.

Developers should track the number of registers per compute unit consumed by the kernel, especially when using large-sized FFT kernels.

16.4 Implementation of 2D FFT

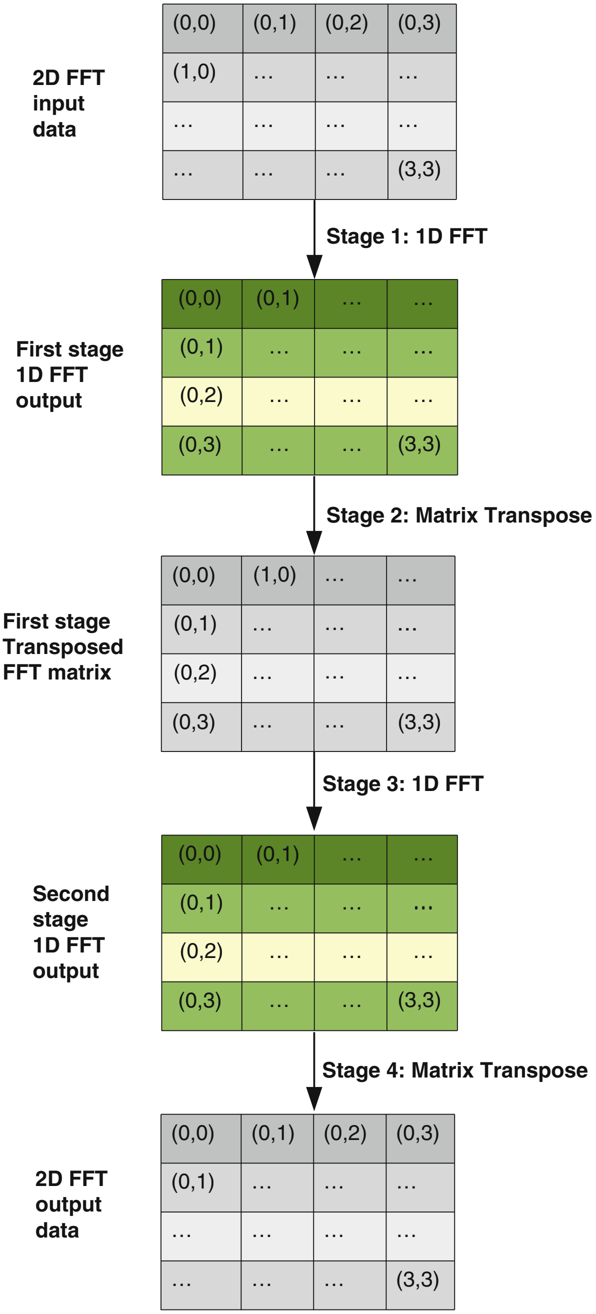
We have learned about the implementation of a 1D-FFT kernel on the GPU in Sect. 16.3. Next, we proceed to implement the 2D-FFT computation kernel.

2D FFT computations are generally used in media applications, including video and image processing. Such applications have input data arranged in a 2D matrix format, with data stored using either *row-major* or *column-major* format. A 2D FFT is defined as follows:

$$\mathscr{F}\{f(x,y)\} = F(u,v) = \frac{1}{MN}\sum_{y=0}^{M-1}\sum_{x=0}^{N-1} f(x,y)W_N^{ux}W_M^{vy} \tag{16.8}$$

where M and N are the dimensions of the input matrix.

Equation (16.8) describes the 2D FFT computation. As observed, the 2D-FFT requires two summations over two dimensions. This computation can be implemented leveraging the 1D-FFT kernel as a building block. This method of computing the 2D-FFT is shown in Fig. 16.6.


2D FFT input data
Stage 1: 1D FFT
First stage 1D FFT output
Stage 2: Matrix Transpose
First stage Transposed FFT matrix
Stage 3: 1D FFT
Second stage 1D FFT output
Stage 4: Matrix Transpose
2D FFT output data

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	…	…	…
…	…	…	…
…	…	…	(3,3)

(0,0)	(0,1)	…	…
(0,1)	…	…	…
(0,2)	…	…	…
(0,3)	…	…	(3,3)

(0,0)	(1,0)	…	…
(0,1)	…	…	…
(0,2)	…	…	…
(0,3)	…	…	(3,3)

(0,0)	(0,1)	…	…
(0,1)	…	…	…
(0,2)	…	…	…
(0,3)	…	…	(3,3)

(0,0)	(0,1)	(0,2)	(0,3)
(0,1)	…	…	…
…	…	…	…
…	…	…	(3,3)

Fig. 16.6 Stages to compute a 2D FFT using a 1D FFT and a matrix transpose as building blocks

Figure 16.6 outlines the process of computing a 2D FFT for a 4 × 4 matrix. The first stage of the transform performs an in-place 1D FFT for each row of the input data matrix. The second stage performs a transpose over the output of the first stage. At this point we have a transposed 1D FFT matrix. The third stage performs another in-place 1D FFT over the rows of the matrix. The fourth stage again performs a transpose over the matrix using the output of the third stage, producing the resultant matrix. The output matrix has 2D FFT data output in its natural order (non-transposed).

The matrix transpose is an additional step that must be performed to compute a 2D FFT. This transpose can be performed on the GPU device using the same data buffers used for computing the FFT stages. This avoids the overhead of moving data back and forth between the CPU and the GPU. A basic matrix transpose kernel is provided below.

Kernel Code for Matrix Transpose:

```
#define BLOCK_SIZE 16
__kernel void  Matrix_Transpose(__global float2* d_in,
                                __global float2* d_out,
                                int width,
                                int height)
{

  // Read the matrix block into local memory

  __local float2 block[BLOCK_SIZE * (BLOCK_SIZE + 1)]

  unsigned int id_x = get_global_id(0);
  unsigned int id_y = get_global_id(1);

  if((id_x < width) && (id_y < height))
  {
    unsigned int in_index = id_y * width + id_x;
    int I_din = get_local_id(1)*(BLOCK_SIZE+1)
              + get_local_id(0);
    block[I_din]=  d_in[in_index];
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  // Write transposed matrix block back to global
  memory

  id_x = get_group_id(1) * BLOCK_SIZE + get_local_id
  (0);
  id_y = get_group_id(0) * BLOCK_SIZE + get_local_id
  (1);
```

```
 if((id_x < height) && (id_y < width))
 {
  unsigned int index_out = id_y * height + id_x;
  int I_dout = get_local_id(0)*(BLOCK_SIZE+1)
               + get_local_id(1);
  d_out[index_out] = block[I_dout];
 }
}
```

The use of **double precision** data types is supported by setting a `pragma` directive in the OpenCL kernel code. Data types such as `double, double2, double4, double8` and `double16` can be used by specifying `#pragma OPENCL EXTENSION cl_khr_fp64 : enable` in the kernel code. Use of double precision data types can affect the execution performance of the FFT application when compared to using the single precision version.

16.5 Performance Evaluation of FFT

In this section, we provide a performance evaluation of the FFT computation described in the previous sections, as run on both AMD and Nvidia GPUs.

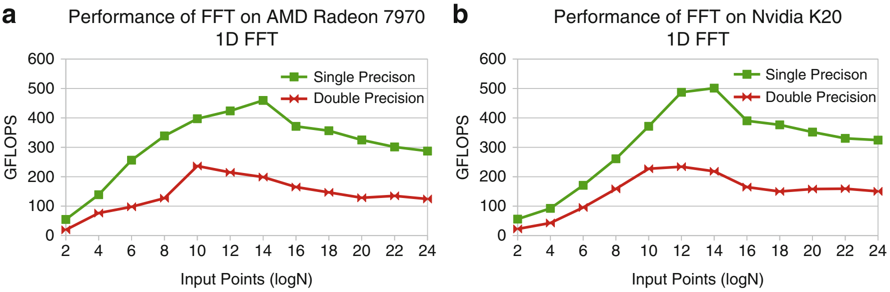


Fig. 16.7 Execution performance (in GFLOPS) for the 1D FFT, as run on a (**a**) AMD Radeon 7970 GPU and a (**b**) Nvidia K20 GPU

The evaluation is performed on a system consisting of an Intel Core i7 3770k processor as the CPU host. The AMD Radeon 7970 discrete GPU and Nvidia K20 discrete GPU are used to accelerate execution of FFT. The performance of the GPU is measured in Floating Point Operations per Second *FLOPS*. For an input of size N, the FFT performs (5×N$\log_2$(N)) floating point operations (FLOPs) [7].

Figure 16.7a shows the performance of the 1D FFT using single and double precision on AMD platform. Figure 16.7b shows the same for the Nvidia platform.

Figure 16.8 shows the performance of the 2D FFT as run on a Nvidia K20 and a AMD Radeon GPU. The 2D FFT uses 2 1D FFT computations and 2

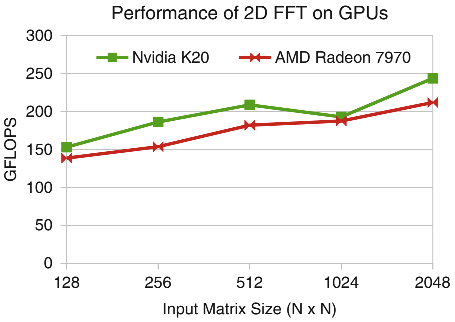


Fig. 16.8 Execution performance of 2D FFT (Single Precision) on Nvidia K20 and AMD Radeon 7970 GPU

transpose computations to carry out the transform. We can notice the added overhead of launching the transpose in the kernels for the 2D FFT, as compared to the performance of the 1D FFT.

GPU vendors provide a highly optimized FFT library for their devices. The library provides customized APIs for computing 1D and 2D FFT on the GPU. Developers can choose to use these optimized libraries to reduce code production time. **cuFFT** by Nvidia and **clAmdFFT** by AMD are optimized libraries developed in CUDA and OpenCL, respectively [1, 5].

In this chapter was have presented 1D and 2D implementations of an FFT. We have discussed some of the tradeoffs when mapping these implementations to GPUs. We have also provided samples runs of these codes developed in OpenCL when run on state-of-the-art GPUs.

Appendix

Host code for FFT application

```
 /* FFT.c */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include <CL/cl.h>

/* Function to check and handle OpenCL errors */
int check_status(int status, char* message) {
  if (status != CL_SUCCESS) {
    printf(message);
    printf("\n ERR: %d\n", status);
    fflush(NULL);
    exit(-1);
  }
}

/* Define custom constants*/
#define MAX_SOURCE_SIZE (10000000)

int main(int argc, char** argv) {
  /* OpenCL Variables */
  cl_uint num_data = 0;
  cl_uint num_stage = 0;
  cl_uint stages_reqd = 0;
  cl_float2* input = NULL;
  cl_float2* output = NULL;
  cl_float* coeff = NULL;
  cl_float* temp_output = NULL;
  cl_platform_id platform_id = NULL;
  cl_device_id device_id = NULL;
  cl_context context;
  cl_command_queue command_queue;
  cl_uint ret_num_platforms;
  cl_uint ret_num_devices;
  cl_mem input_buffer;
  cl_mem output_buffer;
  cl_mem comp_stage_buffer;
  cl_kernel kernel_2;
  cl_kernel kernel_4;
  cl_kernel kernel_8;
  cl_event event;
  cl_int ret;

  /* Custom Variables */
  unsigned int i = 0;
  unsigned int count = 1;
  unsigned int local;
  unsigned int launch_8, launch_4, launch_2;
  unsigned int iters;
  char *platformVendor;
  size_t platInfoSize;

  /* Command Line based Initialization */
  if (argc > 1) {
    num_data = atoi(argv[1]);
    iters = atoi(argv[2]);
```

```
}

/* Allocate input memory */
input = (cl_float2 *) malloc(
    num_data * sizeof(cl_float2));
output = (cl_float2 *) malloc(
    num_data * sizeof(cl_float2));

/* Generate Input Data */
for (i = 0; i < num_data; i++) {
  input[i].x = Random();
  input[i].y = Random();
}

/* Load the kernel source code into the array source_str */
FILE *fp;
char *source_str;
size_t source_size;

fp = fopen("FFT.cl", "r");
if (!fp) {
  fprintf(stderr, "Failed to load kernel.\n");
  exit(1);
}
source_str = (char*) malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

/* Get platform id */
ret = clGetPlatformIDs(1, &platform_id,
    &ret_num_platforms);
check_status(ret, "Error: Platform ID\n");

/* Get device id */

ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1,
    &device_id, &ret_num_devices);
check_status(ret, "Error: Create Program\n");
printf("\nNo of Devices %d", ret_num_platforms);

/* Get platform info */
clGetPlatformInfo(platform_id, CL_PLATFORM_VENDOR, 0,
    NULL, &platInfoSize);
platformVendor = (char*) malloc(platInfoSize);
clGetPlatformInfo(platform_id, CL_PLATFORM_VENDOR,
    platInfoSize, platformVendor, NULL);
printf("\nVendor: %s\n", platformVendor);
free(platformVendor);

/* Create an OpenCL context */
context = clCreateContext(NULL, 1, &device_id, NULL,
    NULL, &ret);

/* Create a command queue */
command_queue = clCreateCommandQueue(context, device_id,
    CL_QUEUE_PROFILING_ENABLE, &ret);

/* Define and create memory buffers on the device for each vector */
input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY,
    sizeof(cl_float2) * num_data, NULL, &ret);
output_buffer = clCreateBuffer(context,
    CL_MEM_READ_WRITE, sizeof(cl_float2) * num_data,
    NULL, &ret);
comp_stage_buffer = clCreateBuffer(context,
    CL_MEM_READ_ONLY, sizeof(cl_uint), NULL, &ret);

/* Create a program from the kernel source */
program = clCreateProgramWithSource(context, 1,
    (const char **) &source_str,
    (const size_t *) &source_size, &ret);
check_status(ret, "Error: Create Program\n");

/* Build program */
ret = clBuildProgram(program, 1, &device_id, NULL, NULL,
    NULL);

/* Dump the program build info to a string */
size_t paramValueSize = 1024 * 1024,
    param_value_size_ret;
char *paramValue;
paramValue = (char*) calloc(paramValueSize,
    sizeof(char));
ret = clGetProgramBuildInfo(program, device_id,
    CL_PROGRAM_BUILD_LOG, paramValueSize, paramValue,
    &param_value_size_ret);
printf(" \n\n %s \n\n", paramValue);
check_status(ret, "Error: Build Program\n");

/* Create the OpenCL kernel for 8-point, 4-point and 2-point FFT */
kernel_8 = clCreateKernel(program, "FFT_MS_8", &ret);
```

```
check_status(ret,
    "Error: Create kernel 8. (clCreateKernel)\n");

kernel_4 = clCreateKernel(program, "FFT_MS_4", &ret);
check_status(ret,
    "Error: Create kernel 4. (clCreateKernel)\n");

kernel_2 = clCreateKernel(program, "FFT_MS_2", &ret);
check_status(ret,
    "Error: Create kernel 2. (clCreateKernel)\n");

/* Set the arguments of the kernel_8, kernel_4, kernel_2 */
ret = clSetKernelArg(kernel_8, 0, sizeof(cl_mem),
    (void *) &input_buffer);
ret = clSetKernelArg(kernel_8, 1, sizeof(cl_mem),
    (void *) &output_buffer);
ret = clSetKernelArg(kernel_8, 2, sizeof(cl_mem),
    (void *) &comp_stage_buffer);
ret = clSetKernelArg(kernel_8, 3, sizeof(cl_uint),
    (void *) &num_data);

ret = clSetKernelArg(kernel_4, 0, sizeof(cl_mem),
    (void *) &input_buffer);
ret = clSetKernelArg(kernel_4, 1, sizeof(cl_mem),
    (void *) &output_buffer);
ret = clSetKernelArg(kernel_4, 2, sizeof(cl_mem),
    (void *) &comp_stage_buffer);
ret = clSetKernelArg(kernel_4, 3, sizeof(cl_uint),
    (void *) &num_data);

ret = clSetKernelArg(kernel_2, 0, sizeof(cl_mem),
    (void *) &input_buffer);
ret = clSetKernelArg(kernel_2, 1, sizeof(cl_mem),
    (void *) &output_buffer);
ret = clSetKernelArg(kernel_2, 2, sizeof(cl_mem),
    (void *) &comp_stage_buffer);
ret = clSetKernelArg(kernel_2, 3, sizeof(cl_uint),
    (void *) &num_data);

/* Initialize Memory Buffer */
ret = clEnqueueWriteBuffer(command_queue, input_buffer,
    1, 0, num_data * sizeof(cl_float2), input, 0, 0,
    &event);
check_status(ret, "Error: write input Buffer\n");
```

```
eventList->add(event);

/* Decide the local group formation for kernel_8 */
size_t globalThreads_8[0] = { num_data / 8 };

/* Decide the local group formation for kernel_4 */
size_t globalThreads_4[0] = { num_data / 4 };

/* Decide the local group formation for kernel_2 */
size_t globalThreads_2[0] = { num_data / 2 };

/* Setup Multi-stage FFT launch counters */
stages_reqd = log2(num_data);
launch_8 = stages_reqd;
launch_4 = 0;
launch_2 = 0;

/* Calculate 8-point FFT kernel launch count */
while (launch_8) {
  if ((launch_8) % 3) {
    launch_8--;
  } else
    break;
}
launch_8 = launch_8 / 3;

/* Calculate 4-point and 2-point FFT kernel launch count */
switch (stages_reqd - launch_8) {
case (2):
  launch_4 = 2;
  break;
case (1):
  launch_2 = 1;
  break;
default:
  break;
}

/* FFT compute Loop */
while (count < stages_reqd) {
  num_stage = (1 << count);
  if (launch_8) {
    ret = clEnqueueWriteBuffer(command_queue,
        comp_stage_buffer, 1, 0, sizeof(cl_uint),
        &num_stage, 0, 0, &event);
  check_status(ret, "Error: write Buffer\n");

  /* Execute the 8-point FFT kernel*/
  ret = clEnqueueNDRangeKernel(command_queue,
      kernel_8, 1, NULL, globalThreads_8,
      localThreads_8, 0, NULL, &event);
  check_status(ret, "Error: Run kernel 8\n");
  count += 3;
  launch_8--;
 } else if (launch_4) {
  ret = clEnqueueWriteBuffer(command_queue,
      comp_stage_buffer, 1, 0, sizeof(cl_uint),
      &num_stage, 0, 0, &event);
  check_status(ret, "Error: write Buffer\n");

  /* Execute the 4-point FFT kernel */
  ret = clEnqueueNDRangeKernel(command_queue,
      kernel_4, 1, NULL, globalThreads_4,
      localThreads_4, 0, NULL, &event);
  check_status(ret, "Error: Run kernel 4\n");
  count += 2;
  launch_4--;
 } else if (launch_2) {
  ret = clEnqueueWriteBuffer(command_queue,
      comp_stage_buffer, 1, 0, sizeof(cl_uint),
      &num_stage, 0, 0, &event);
  check_status(ret, "Error: write Buffer\n");
  /* Execute the 2-point FFT kernel */
  ret = clEnqueueNDRangeKernel(command_queue,
      kernel_2, 1, NULL, globalThreads_2,
      localThreads_2, 0, NULL, &event);
  check_status(ret, "Error: Run kernel 2\n");
  count += 1;
  launch_2--;
 }
}

/* Get the output buffer */
ret = clEnqueueReadBuffer(command_queue, output_buffer,
  CL_TRUE, 0, num_data * sizeof(cl_float2), output, 0,
  NULL, &event);

ret = clFlush(command_queue);
ret = clFinish(command_queue);
```

```
  /* Release all resources */
  ret = clReleaseKernel(kernel_8);
  ret = clReleaseKernel(kernel_4);
  ret = clReleaseKernel(kernel_2);
  ret = clReleaseProgram(program);
  ret = clReleaseMemObject(input_buffer);
  ret = clReleaseMemObject(output_buffer);
  ret = clReleaseMemObject(comp_stage_buffer);
  ret = clReleaseCommandQueue(command_queue);
  ret = clReleaseContext(context);
  free(input);
  free(output);
  return 0;
}



}
```

References

1. AMD: clAmdfft, OpenCL FFT library from AMD (2013)
2. Cooley, J., Tukey, J.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**, 297–301 (1965)
3. Moreland, K., Angel, E.: The FFT on a GPU. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 112–120 (2003)
4. Munshi, A.: The OpenCL 1.2 Specification. Khronos OpenCL Working Group, Beaverton (2012)
5. Nvidia: Cufft library (2010)
6. NVIDIA: CUDA Programming Guide, Version 5 (2012)
7. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. Society for Industrial and Applied Mathematics (1992). doi:10.1137/1.9781611970999. http://epubs.siam.org/doi/book/10.1137/1.9781611970999