The OpenCL Programming Book

6.1 FFT (Fast Fourier Transform)

The first application we will look at is a program that will perform band-pass filtering on an image. We will start by first explaining the process known as a Fourier Transform, which is required to perform the image processing.

Fourier Transform

"Fourier Transform" is a process that takes in samples of data, and outputs its frequency content. Its general application can be summarized as follows.

- Take in an audio signal and find its frequency content
- Take in an image data and find its spatial frequency content

The output of the Fourier Transform contains all of its input. A process known as the Inverse Fourier Transform can be used to retrieve the original signal.

The Fourier Transform is a process commonly used in many fields. Many programs use this procedure which is required for an equalizer, filter, compressor, etc.

The mathematical formula for the Fourier Transform process is shown below

$$X(\omega) = \int_{-\infty}^{\infty} x(t) \exp(-i\omega t) dt$$

The "i" is an imaginary number, and ω is the frequency in radians.

As you can see from its definition, the Fourier Transform operates on continuous data. However, continuous data means it contains infinite number of points with infinite precision. For this processing to be practical, it must be able to process a data set that contains a finite number of elements. Therefore, a process known as the Discrete Fourier Transform (DFT) was developed to estimate the Fourier Transform, which operates on a finite data set. The mathematical formula is shown below.

$$X_j = \sum_{k=0}^{n-1} x_k \exp(-\frac{2\pi i}{n} jk) \qquad j = 0, 1, \ldots, n-1$$

This formula now allows processing of digital data with a finite number of samples. The problem with this method, however, is that its O(N^2). As the number of points is increased, the processing time grows by a power of 2.

Fast Fourier Transform

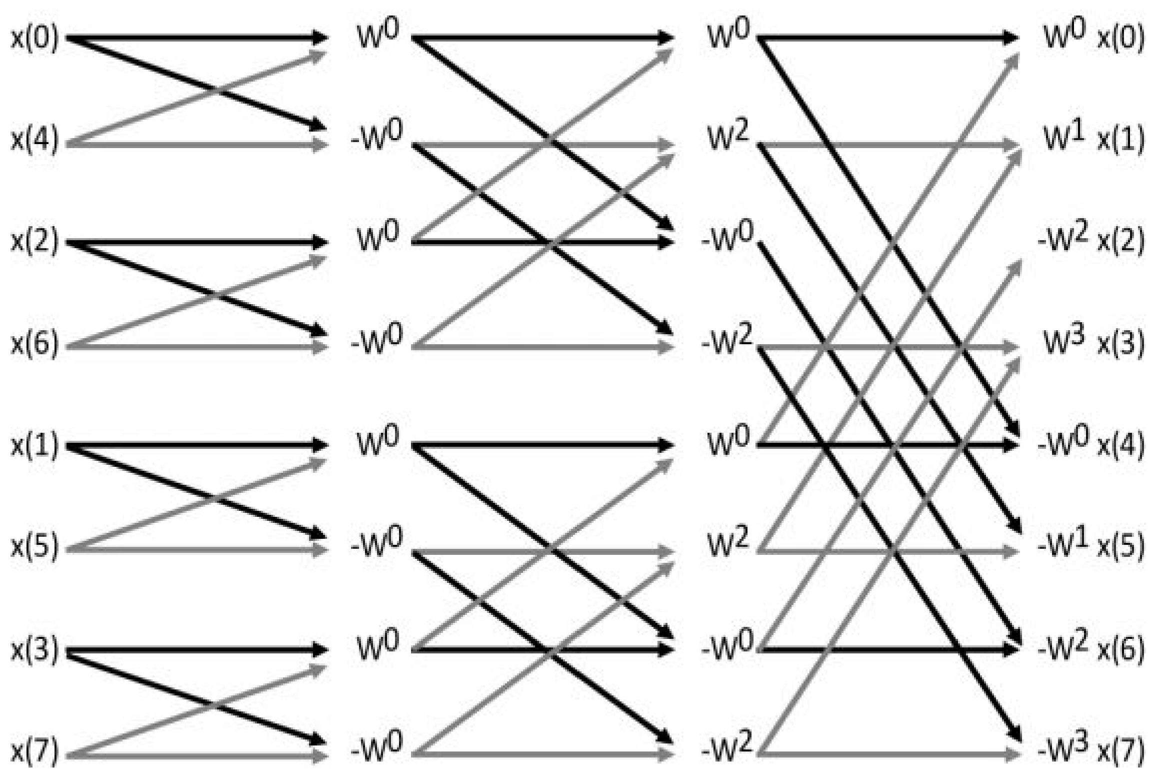
There exists an optimized implementation of the DFT, called the "Fast Fourier Transform (FFT)".

Many different implementations of FFT exist, so we will concentrate on the most commonly used Cooley-Tukey FFT algorithm. An entire book can be dedicated to explaining the FFT algorithm, so we will only explain the minimal amount required to implement the program.

The Cooley-Tukey algorithm takes advantage of the cyclical nature of the Fourier Transform, and solves the problem in O(N log N), by breaking up the DFT into smaller DFTs. The limitation with this algorithm is that the number of input samples must to be a power of 2. This limitation can be overcome by padding the input signal with zeros, or use in conjunction with another FFT algorithm that does not have this requirement. For simplicity, we will only use input signals whose length is a power of 2.

The core computation in this FFT algorithm is what is known as the "Butterfly Operation". The operation is performed on a pair of data samples at a time, whose signal flow graph is shown in Figure 6.2 below. The operation got its name due to the fact that each segment of this flow graph looks like a butterfly.

Figure 6.2: Butterfly Operation



The "W" seen in the signal flow graph is defined as below.

$$W = \exp(-\frac{2\pi i}{n})$$

Looking at Figure 6.2, you may notice the indices of the input are in a seemingly random order. We will not get into details on why this is done in this text except that it is an optimization method, but note that what is known as "bit-reversal" is performed on the indices of the input. The input order in binary is (000, 100, 010, 110, 001, 101, 011, 111). Notice that if you reverse the bit ordering of (100), you get (001). So the new input indices are in numerical order, except that the bits are reversed.

2-D FFT

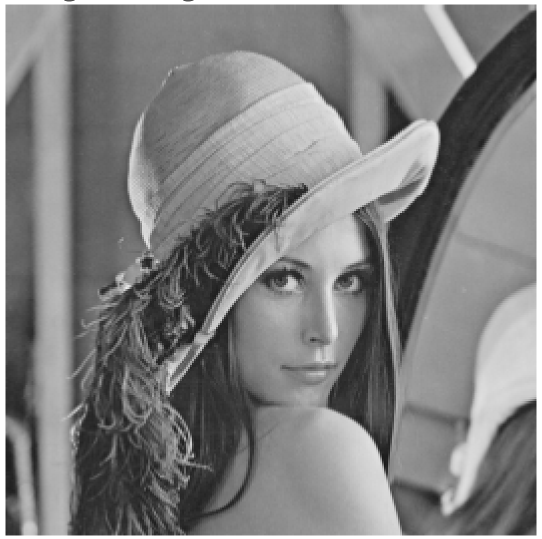
As the previous section shows, the basic FFT algorithm is to be performed on 1 dimensional data. In order to take the FFT of an image, an FFT is taken row-wise and column-wise. Note that we are not dealing with time any more, but with spatial location.

When 2-D FFT is performed, the FFT is first taken for each row, transposed, and the FFT is again taken on this result. This is done for faster memory access, as the data is stored in row-major form. If transposition is not performed, interleaved accessing of memory occurs, which can greatly decrease speed of performance as the size of the image increases.

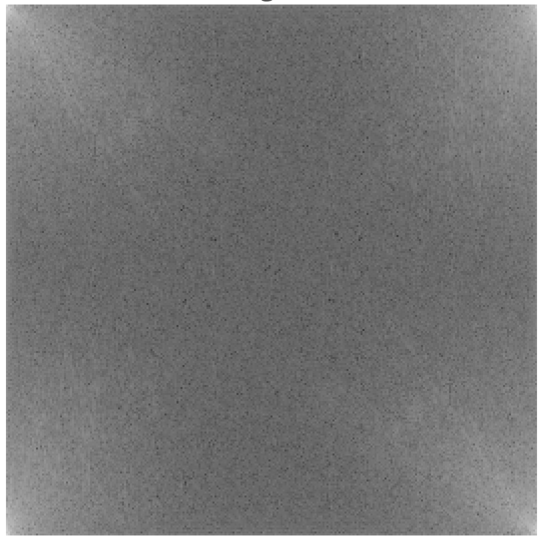
Figure 6.3(b) shows the result of taking a 2-D FFT of Figure 6.3(a).

Figure6.3: 2-D FFT

(a) Original Image



(b) The result of taking a 2-D FFT

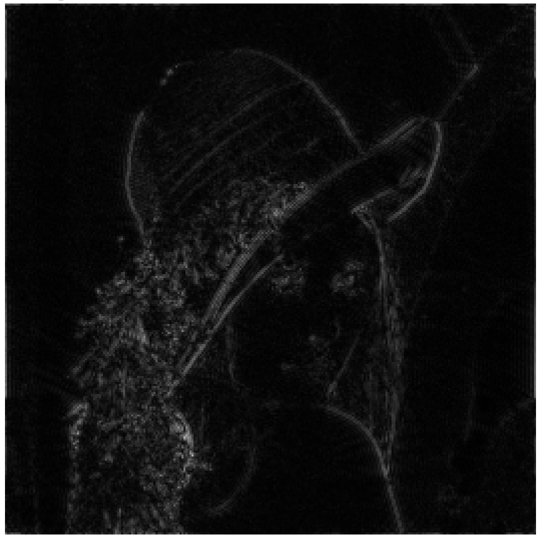


Bandpass Filtering and Inverse Fourier Transform

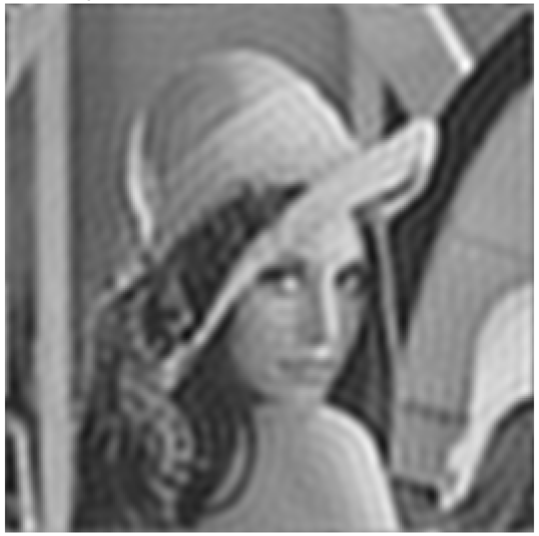
As stated earlier, the signal that has been transformed to the frequency domain via Fourier Transform can be transformed back using the Inverse Fourier Transform. Using this characteristic, it is possible to perform frequency-based filtering while in the frequency domain and transform back to the original domain. For example, the low-frequency components can be cut, which leaves the part of the image where a sudden change occurs. This is known as an "Edge Filter", and its result is shown in Figure 6.4a. If the high-frequency components are cut instead, the edges will be blurred, resulting in an image shown in Figure 6.4b. This is known as a "low-pass filter".

Figure 6.4:Edge Filter and Low-pass Filter

(a) Edged Filter



(b) Low-pass Filter



The mathematical formula for Inverse Discrete Fourier Transform is shown below.

$$x_j = \frac{1}{n}\sum_{k=0}^{n-1} X_k \exp(\frac{2\pi i}{n} jk) \qquad j = 0, 1, ..., n \quad 1$$

Note its similarity with the DFT formula. The only differences are:

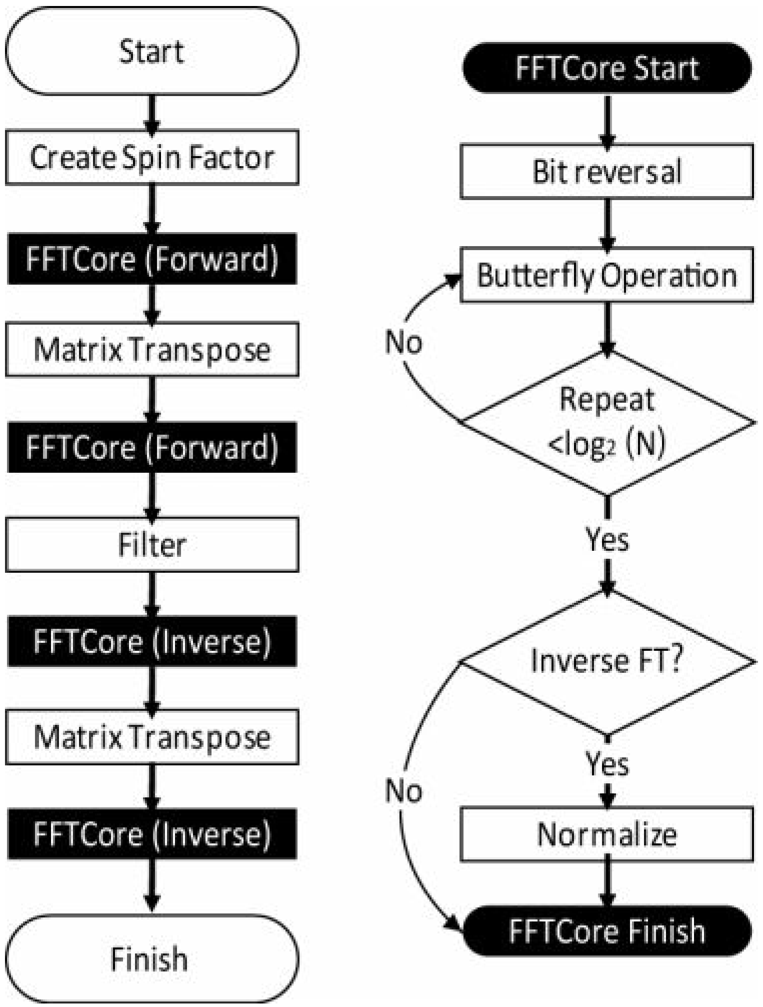
- Must be normalized by the number of samples
- The term within the exp() is positive.

The rest of the procedure is the same. Therefore, the same kernel can be used to perform either operation.

Overall Program Flow-chart

The overall program flow-chart is shown in Figure 6.5 below.

Figure 6.5: Program flow-chart



Each process is dependent on the previous process, so each of the steps must be followed in sequence. A kernel will be written for each of the processes in Figure 6.5.

Source Code Walkthrough

We will first show the entire source code for this program. List 6.1 is the kernel code, and List 6.2 is the host code.

List 6.1: Kernel Code

```
#define PI 3.14159265358979323846
#define PI_2 1.57079632679489661923

__kernel void spinFact(__global float2* w, int n)
{
unsigned int i = get_global_id(0);

float2 angle = (float2)(2*i*PI/(float)n,(2*i*PI/(float)n)+PI_2);
w[i] = cos(angle);
}

__kernel void bitReverse(__global float2 *dst, __global float2 *src, int m, int n)
{
unsigned int gid = get_global_id(0);
unsigned int nid = get_global_id(1);

unsigned int j = gid;
j = (j & 0x55555555) << 1 | (j & 0xAAAAAAAA) >> 1;
j = (j & 0x33333333) << 2 | (j & 0xCCCCCCCC) >> 2;
j = (j & 0x0F0F0F0F) << 4 | (j & 0xF0F0F0F0) >> 4;
j = (j & 0x00FF00FF) << 8 | (j & 0xFF00FF00) >> 8;
j = (j & 0x0000FFFF) << 16 | (j & 0xFFFF0000) >> 16;

j >>= (32-m);

dst[nid*n+j] = src[nid*n+gid];
}

__kernel void norm(__global float2 *x, int n)
{
unsigned int gid = get_global_id(0);
unsigned int nid = get_global_id(1);

x[nid*n+gid] = x[nid*n+gid] / (float2)((float)n, (float)n);
}

__kernel void butterfly(__global float2 *x, __global float2* w, int m, int n, int iter, uint flag)
{
unsigned int gid = get_global_id(0);
unsigned int nid = get_global_id(1);

int butterflySize = 1 << (iter-1);
int butterflyGrpDist = 1 << iter;
int butterflyGrpNum = n >> iter;
int butterflyGrpBase = (gid >> (iter-1))*(butterflyGrpDist);
int butterflyGrpOffset = gid & (butterflySize-1);

int a = nid * n + butterflyGrpBase + butterflyGrpOffset;
int b = a + butterflySize;

int l = butterflyGrpNum * butterflyGrpOffset;

float2 xa, xb, xbxx, xbyy, wab, wayx, wbyx, resa, resb;

xa = x[a];
xb = x[b];
xbxx = xb.xx;
xbyy = xb.yy;

wab = as_float2(as_uint2(w[l]) ^ (uint2)(0x0, flag));
wayx = as_float2(as_uint2(wab.yx) ^ (uint2)(0x80000000, 0x0));
wbyx = as_float2(as_uint2(wab.yx) ^ (uint2)(0x0, 0x80000000));

resa = xa + xbxx*wab + xbyy*wayx;
resb = xa - xbxx*wab + xbyy*wbyx;

x[a] = resa;
x[b] = resb;
}
```

```

__kernel void transpose(__global float2 *dst, __global float2* src, int n)
{
unsigned int xgid = get_global_id(0);
unsigned int ygid = get_global_id(1);

unsigned int iid = ygid * n + xgid;
unsigned int oid = xgid * n + ygid;

dst[oid] = src[iid];
}

__kernel void highPassFilter(__global float2* image, int n, int radius)
{
unsigned int xgid = get_global_id(0);
unsigned int ygid = get_global_id(1);

int2 n_2 = (int2)(n>>1, n>>1);
int2 mask = (int2)(n-1, n-1);

int2 gid = ((int2)(xgid, ygid) + n_2) & mask;

int2 diff = n_2 - gid;
int2 diff2 = diff * diff;
int dist2 = diff2.x + diff2.y;

int2 window;

if (dist2 < radius*radius) {
window = (int2)(0L, 0L);
} else {
window = (int2)(-1L, -1L);
}
900  7:00:00 AM
image[ygid*n+xgid] = as_float2(as_int2(image[ygid*n+xgid]) & window);
}
```

List 6.2: Host Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#include "pgm.h"

#define PI 3.14159265358979

#define MAX_SOURCE_SIZE (0x100000)

#define AMP(a, b) (sqrt((a)*(a)+(b)*(b)))

cl_device_id device_id = NULL;
cl_context context = NULL;
cl_command_queue queue = NULL;
cl_program program = NULL;

enum Mode {
forward = 0,
inverse = 1
};

int setWorkSize(size_t* gws, size_t* lws, cl_int x, cl_int y)
{
switch(y) {
case 1:
gws[0] = x;
gws[1] = 1;
lws[0] = 1;
lws[1] = 1;
break;
default:
gws[0] = x;
gws[1] = y;
lws[0] = 1;
lws[1] = 1;
break;
}

return 0;
}

int fftCore(cl_mem dst, cl_mem src, cl_mem spin, cl_int m, enum Mode direction)
{
cl_int ret;

cl_int iter;
cl_uint flag;

cl_int n = 1<<m;

cl_event kernelDone;

cl_kernel brev = NULL;
cl_kernel bfly = NULL;
cl_kernel norm = NULL;

brev = clCreateKernel(program, "bitReverse", &ret);
bfly = clCreateKernel(program, "butterfly", &ret);
norm = clCreateKernel(program, "norm", &ret);

size_t gws[2];
size_t lws[2];

switch (direction) {
case forward:flag = 0x00000000; break;
case inverse:flag = 0x80000000; break;
}

ret = clSetKernelArg(brev, 0, sizeof(cl_mem), (void *)&dst);
ret = clSetKernelArg(brev, 1, sizeof(cl_mem), (void *)&src);
ret = clSetKernelArg(brev, 2, sizeof(cl_int), (void *)&m);
ret = clSetKernelArg(brev, 3, sizeof(cl_int), (void *)&n);

ret = clSetKernelArg(bfly, 0, sizeof(cl_mem), (void *)&dst);
ret = clSetKernelArg(bfly, 1, sizeof(cl_mem), (void *)&spin);
ret = clSetKernelArg(bfly, 2, sizeof(cl_int), (void *)&m);
ret = clSetKernelArg(bfly, 3, sizeof(cl_int), (void *)&n);
ret = clSetKernelArg(bfly, 5, sizeof(cl_uint), (void *)&flag);

ret = clSetKernelArg(norm, 0, sizeof(cl_mem), (void *)&dst);
ret = clSetKernelArg(norm, 1, sizeof(cl_int), (void *)&n);

/* Reverse bit ordering */
setWorkSize(gws, lws, n, n);
ret = clEnqueueNDRangeKernel(queue, brev, 2, NULL, gws, lws, 0, NULL, NULL);

/* Perform Butterfly Operations*/
setWorkSize(gws, lws, n/2, n);
for (iter=1; iter <= m; iter++){
ret = clSetKernelArg(bfly, 4, sizeof(cl_int), (void *)&iter);
ret = clEnqueueNDRangeKernel(queue, bfly, 2, NULL, gws, lws, 0, NULL, &kernelDone);

ret = clWaitForEvents(1, &kernelDone);
}

if (direction == inverse) {
setWorkSize(gws, lws, n, n);
ret = clEnqueueNDRangeKernel(queue, norm, 2, NULL, gws, lws, 0, NULL, &kernelDone);

ret = clWaitForEvents(1, &kernelDone);
}

ret = clReleaseKernel(bfly);
ret = clReleaseKernel(brev);
ret = clReleaseKernel(norm);

return 0;
}

int main()
```

```
{
cl_mem xmobj = NULL;
cl_mem rmobj = NULL;
cl_mem wmobj = NULL;
cl_kernel sfac = NULL;
cl_kernel trns = NULL;
cl_kernel hpfl = NULL;

cl_platform_id platform_id = NULL;

cl_uint ret_num_devices;
cl_uint ret_num_platforms;

cl_int ret;

cl_float2 *xm;
cl_float2 *rm;
cl_float2 *wm;

pgm_t ipgm;
pgm_t opgm;

FILE *fp;
const char fileName[] = "./fft.cl";
size_t source_size;
char *source_str;
cl_int i, j;
cl_int n;
cl_int m;

size_t gws[2];
size_t lws[2];

/* Load kernel source code */
fp = fopen(fileName, "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 );

/* Read image */
readPGM(&ipgm, "lena.pgm");

n = ipgm.width;
m = (cl_int)(log((double)n)/log(2.0));

xm = (cl_float2 *)malloc(n * n * sizeof(cl_float2));
rm = (cl_float2 *)malloc(n * n * sizeof(cl_float2));
wm = (cl_float2 *)malloc(n / 2 * sizeof(cl_float2));

for (i=0; i < n; i++) {
for (j=0; j < n; j++) {
((float*)xm)[(2*n*j)+2*i+0] = (float)ipgm.buf[n*j+i];
((float*)xm)[(2*n*j)+2*i+1] = (float)0;
}
}

/* Get platform/device  */
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &ret_num_
devices);

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

/* Create Command queue */
queue = clCreateCommandQueue(context, device_id, 0, &ret);

/* Create Buffer Objects */
xmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, n*n*sizeof(cl_float2), NULL, &re
t);
rmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, n*n*sizeof(cl_float2), NULL, &re
t);
wmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, (n/2)*sizeof(cl_float2), NULL,
&ret);
```

```

/* Transfer data to memory buffer */
ret = clEnqueueWriteBuffer(queue, xmobj, CL_TRUE, 0, n*n*sizeof(cl_float2), xm, 0,
NULL, NULL);

/* Create kernel program from source */
program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const
size_t *)&source_size, &ret);

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

/* Create OpenCL Kernel */
sfac = clCreateKernel(program, "spinFact", &ret);
trns = clCreateKernel(program, "transpose", &ret);
hpfl = clCreateKernel(program, "highPassFilter", &ret);

/* Create spin factor */
ret = clSetKernelArg(sfac, 0, sizeof(cl_mem), (void *)&wmobj);
ret = clSetKernelArg(sfac, 1, sizeof(cl_int), (void *)&n);
setWorkSize(gws, lws, n/2, 1);
ret = clEnqueueNDRangeKernel(queue, sfac, 1, NULL, gws, lws, 0, NULL, NULL);

/* Butterfly Operation */
fftCore(rmobj, xmobj, wmobj, m, forward);

/* Transpose matrix */
ret = clSetKernelArg(trns, 0, sizeof(cl_mem), (void *)&xmobj);
ret = clSetKernelArg(trns, 1, sizeof(cl_mem), (void *)&rmobj);
ret = clSetKernelArg(trns, 2, sizeof(cl_int), (void *)&n);
setWorkSize(gws, lws, n, n);
ret = clEnqueueNDRangeKernel(queue, trns, 2, NULL, gws, lws, 0, NULL, NULL);

/* Butterfly Operation */
fftCore(rmobj, xmobj, wmobj, m, forward);

/* Apply high-pass filter */
cl_int radius = n/8;
ret = clSetKernelArg(hpfl, 0, sizeof(cl_mem), (void *)&rmobj);
ret = clSetKernelArg(hpfl, 1, sizeof(cl_int), (void *)&n);
ret = clSetKernelArg(hpfl, 2, sizeof(cl_int), (void *)&radius);
setWorkSize(gws, lws, n, n);
ret = clEnqueueNDRangeKernel(queue, hpfl, 2, NULL, gws, lws, 0, NULL, NULL);

/* Inverse FFT */

/* Butterfly Operation */
fftCore(xmobj, rmobj, wmobj, m, inverse);

/* Transpose matrix */
ret = clSetKernelArg(trns, 0, sizeof(cl_mem), (void *)&rmobj);
ret = clSetKernelArg(trns, 1, sizeof(cl_mem), (void *)&xmobj);
setWorkSize(gws, lws, n, n);
ret = clEnqueueNDRangeKernel(queue, trns, 2, NULL, gws, lws, 0, NULL, NULL);

/* Butterfly Operation */
fftCore(xmobj, rmobj, wmobj, m, inverse);

/* Read data from memory buffer */
ret = clEnqueueReadBuffer(queue, xmobj, CL_TRUE, 0, n*n*sizeof(cl_float2), xm, 0, N
ULL, NULL);

/*  */
float* ampd;
ampd = (float*)malloc(n*n*sizeof(float));
for (i=0; i < n; i++) {
for (j=0; j < n; j++) {
ampd[n*((i))+((j))] = (AMP(((float*)xm)[(2*n*i)+2*j], ((float*)xm)
[(2*n*i)+2*j+1]));
}
}
opgm.width = n;
opgm.height = n;
normalizeF2PGM(&opgm, ampd);
free(ampd);

/* Write out image */
writePGM(&opgm, "output.pgm");
```

```

/* Finalizations*/
ret = clFlush(queue);
ret = clFinish(queue);
ret = clReleaseKernel(hpfl);
ret = clReleaseKernel(trns);
ret = clReleaseKernel(sfac);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(xmobj);
ret = clReleaseMemObject(rmobj);
ret = clReleaseMemObject(wmobj);
ret = clReleaseCommandQueue(queue);
ret = clReleaseContext(context);

destroyPGM(&ipgm);
destroyPGM(&opgm);

free(source_str);
free(wm);
free(rm);
free(xm);

return 0;
}
```

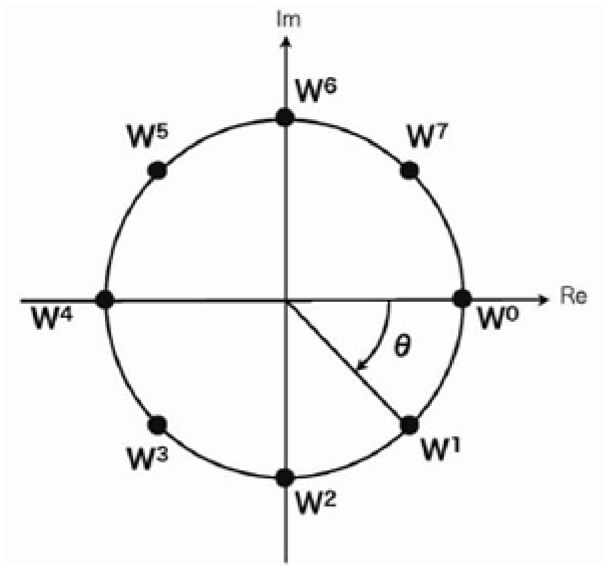
We will start by taking a look at each kernel.

List 6.3: Create Spin Factor

```
__kernel void spinFact(__global float2* w, int n)
{
unsigned int i = get_global_id(0);

float2 angle = (float2)(2*i*PI/(float)n,(2*i*PI/(float)n)+PI_2);
w[i] = cos(angle);
}
```

The code in List 6.3 is used to pre-compute the value of the spin factor "w", which gets repeatedly used in the butterfly operation. The "w" is computed for radian angles that are multiples of (2π/n), which is basically the real and imaginary components on the unit circle using cos() and -sin(). Note the shift by PI/2 in line 8 allows the cosine function to compute -sin(). This is done to utilize the SIMD unit on the OpenCL device if it has one.

Figure 6.6: Spin factor for n=8



The pre-computing of the values for "w" creates what is known as a "lookup table", which stores values to be used repeatedly on the memory. On some devices, such as the GPU, it may prove to be faster if the same operation is performed each time, as it may be more expensive to access the memory.

List 6.4: Bit reversing

```
__kernel void bitReverse(__global float2 *dst, __global float2 *src, int m, int n)
{
unsigned int gid = get_global_id(0);
unsigned int nid = get_global_id(1);

unsigned int j = gid;
j = (j & 0x55555555) << 1 | (j & 0xAAAAAAAA) >> 1;
j = (j & 0x33333333) << 2 | (j & 0xCCCCCCCC) >> 2;
j = (j & 0x0F0F0F0F) << 4 | (j & 0xF0F0F0F0) >> 4;
10. j = (j & 0x00FF00FF) << 8 | (j & 0xFF00FF00) >> 8;
11. j = (j & 0x0000FFFF) << 16 | (j & 0xFFFF0000) >> 16;
12.
13. j >>= (32-m);
14.
15. dst[nid*n+j] = src[nid*n+gid];
16. }
```

List 6.4 shows the kernel code for reordering the input data such that it is in the order of the bit-reversed index. Lines 18~22 performs the bit reversing of the inputs. The indices are correctly shifted in line 24, as the max index would otherwise be 2^32-1.

Also, note that a separate memory space must be allocated for the output on the global memory. These types of functions are known as an out-of-place function. This is done since the coherence of the data cannot be guaranteed if the input gets overwritten each time after processing. An alternative solution is shown in List 6.5, where each work item stores the output locally until all work items are finished, at which point the locally stored data is written to the input address space.

List 6.5: Bit reversing (Using synchronization)

```
__kernel void bitReverse(__global float2 *x, int m, int n)
{
unsigned int gid = get_global_id(0);
unsigned int nid = get_global_id(1);

unsigned int j = gid;
j = (j & 0x55555555) << 1 | (j & 0xAAAAAAAA) >> 1;
j = (j & 0x33333333) << 2 | (j & 0xCCCCCCCC) >> 2;
j = (j & 0x0F0F0F0F) << 4 | (j & 0xF0F0F0F0) >> 4;
j = (j & 0x00FF00FF) << 8 | (j & 0xFF00FF00) >> 8;
j = (j & 0x0000FFFF) << 16 | (j & 0xFFFF0000) >> 16;

j >>= (32-m);

float2 val = x[nid*n+gid];

SYNC_ALL_THREAD /* Synchronize all work-items */

x[nid*n+j] = val;
}
```

However, OpenCL does not currently require the synchronization capability in its specification. It may be supported in the future, but depending on the device, these types of synchronization, especially when processing large amounts of data, can potentially decrease performance. If there is enough space on the device, the version on List 6.4 should be used.

List 6.6: Normalizing by the number of samples

```
__kernel void norm(__global float2 *x, int n)
{
unsigned int gid = get_global_id(0);
unsigned int nid = get_global_id(1);

x[nid*n+gid] = x[nid*n+gid] / (float2)((float)n, (float)n);
}
```

The code in List 6.6 should be self-explanatory. It basically just dives the input by the value of "n". The operation is performed on a float2 type. Since the value of "n" is limited to a power of 2, you may be tempted to use shifting, but division by shifting is only possible for integer types. Shifting of a float value will result in unwanted results.

List 6.7: Butterfly operation

```
__kernel void butterfly(__global float2 *x, __global float2* w, int m, int n, int iter, uint flag)
{
unsigned int gid = get_global_id(0);
unsigned int nid = get_global_id(1);

int butterflySize = 1 << (iter-1);
int butterflyGrpDist = 1 << iter;
int butterflyGrpNum = n >> iter;
int butterflyGrpBase = (gid >> (iter-1))*(butterflyGrpDist);
int butterflyGrpOffset = gid & (butterflySize-1);
11.
12. int a = nid * n + butterflyGrpBase + butterflyGrpOffset;
13. int b = a + butterflySize;
14.
15. int l = butterflyGrpNum * butterflyGrpOffset;
16.
17. float2 xa, xb, xbxx, xbyy, wab, wayx, wbyx, resa, resb;
18.
19. xa = x[a];
20. xb = x[b];
21. xbxx = xb.xx;
22. xbyy = xb.yy;
23.
24. wab = as_float2(as_uint2(w[l]) ^ (uint2)(0x0, flag));
25. wayx = as_float2(as_uint2(wab.yx) ^ (uint2)(0x80000000, 0x0));
26. wbyx = as_float2(as_uint2(wab.yx) ^ (uint2)(0x0, 0x80000000));
27.
28. resa = xa + xbxx*wab + xbyy*wayx;
29. resb = xa - xbxx*wab + xbyy*wbyx;
30.
31. x[a] = resa;
32. x[b] = resb;
33. }
```

The kernel for the butterfly operation, which performs the core of the FFT algorithm, is shown in List 6.7 above. Each work item performs one butterfly operation for a pair of inputs. Therefore, (n * n)/2 work items are required.

Refer back to the signal flow graph for the butterfly operation in Figure 6.2. As the graph shows, the required inputs are the two input data and the spin factor, which can be derived from the "gid". The intermediate values required in mapping the "gid" to the input and output indices are computed in lines 42-46.

First, the variable "butterflySize" represents the difference in the indices to the data for the butterfly operation to be performed on. The "butterflySize" is 1 for the first iteration, and this value is doubled for each iteration.

Next, we need to know how the butterfly operation is grouped. Looking at the signal flow graph, we see that the crossed signal paths occur within independent groups. In the first iteration, the number of groups is the same as the number of butterfly operation to perform, but in the 2nd iteration, it is split up into 2 groups. This value is stored in the variable butterflyGrpNum.

The differences of the indices between the groups are required as well. This is stored in the variable butterflyGrpDistance.

Next, we need to determine the indices to read from and to write to. The butterflyGrpBase variable contains the index to the first butterfly operation within the group. The butterflyGropOffset is the offset within the group. These are determined using the following formulas.

```
butterflyGrpBase = (gid / butterflySize) * butterflyGrpDistance);
butterflyGrpOffset = gid % butterflySize;
```

For our FFT implementation, we can replace the division and the mod operation with bit shifts, since we are assuming the value of n to be a power of 2.

Now the indices to perform the butterfly operation and the spin factor can be found. We will now go into the actual calculation.

Lines 55 ~ 65 are the core of the butterfly operation. Lines 60 ~ 62 takes the sign of the spin factor into account to take care of the computation for the real and imaginary components, as well as the FFT and IFFT. Lines 64 ~ 65 are the actual operations, and Lines 67 ~ 68 stores the processed data.

List 6.8: Matrix Transpose

```
__kernel void transpose(__global float2 *dst, __global float2* src, int n)
{
unsigned int xgid = get_global_id(0);
unsigned int ygid = get_global_id(1);

unsigned int iid = ygid * n + xgid;
unsigned int oid = xgid * n + ygid;

dst[oid] = src[iid];
}
```

List 6.8 shows a basic implementation of the matrix transpose algorithm. We will not go into optimization of this algorithm, but this process can be speed up significantly by using local memory and blocking.

List 6.9: Filtering

```
__kernel void highPassFilter(__global float2* image, int n, int radius)
{
unsigned int xgid = get_global_id(0);
unsigned int ygid = get_global_id(1);

int2 n_2 = (int2)(n>>1, n>>1);
int2 mask = (int2)(n-1, n-1);

int2 gid = ((int2)(xgid, ygid) + n_2) & mask;

int2 diff = n_2 - gid;
int2 diff2 = diff * diff;
int dist2 = diff2.x + diff2.y;

int2 window;

if (dist2 < radius*radius) {
window = (int2)(0L, 0L);
} else {
window = (int2)(-1L, -1L);
}
900  7:00:00 AM
image[ygid*n+xgid] = as_float2(as_int2(image[ygid*n+xgid]) & window);
}
```

List 6.9 is a kernel that filters an image based on frequency. As the kernel name suggests, the filter passes high frequencies and gets rid of the lower frequencies.

The spatial frequency obtained from the 2-D FFT shows the DC (direct current) component on the 4 edges of the XY coordinate system. A high pass filter can be created by cutting the frequency within a specified radius that includes these DC components. The opposite can be performed to create a low pass filter. In general, a high pass filter extracts the edges, and a low pass filter blurs the image.

Next, we will go over the host program. Most of what is being done is the same as for the OpenCL programs that we have seen so far. The main differences are:

- Multiple kernels are implemented
- Multiple memory objects are used, requiring appropriate data flow construction

Note that when a kernel is called repeatedly, the clSetKernelArg() only need to be changed for when an argument value changes from the previous time the kernel is called. For example, consider the butterfly operations being called in line 94 on the host side.

```
        /* Perform butterfly operations */
        setWorkSize(gws, lws, n/2, n);
        for (iter=1; iter <= m; iter++){
                ret = clSetKernelArg(bfly, 4, sizeof(cl_int), (void *)&iter);
                ret = clEnqueueNDRangeKernel(queue, bfly, 2, NULL, gws, lws, 0, NULL, &kernelDone);
                ret = clWaitForEvents(1, &kernelDone);
        }
```

This butterfly operation kernel is executed log_2(n) times. The kernel must have its iteration number passed in as an argument. The kernel uses this value to compute which data to perform the butterfly operation on.

In this program, the data transfers between kernels occur via the memory objects. The types of kernels used can be classified to either in-place kernel or out-of-place kernel based on the data flow.

The in-place kernel uses the same memory object for both input and output, where the output data is written over the address space of the input data. The out-of-place kernel uses separate memory objects for input and output. The problem with these types of kernel is that it would require too much memory space on the device, and that a data transfer must occur between memory objects. Therefore, it would be wise to use as few memory objects as possible.

For this program, a memory object is required to store the pre-computed values of the spin factors. Since an out-of-place operation such as the matrix transposition exist, at least 2 additional memory objects are required. In fact, only these 3 memory objects are required for this program to run without errors due to race conditions.

To do this, the arguments to the kernel must be appropriately set using clSetKernelArg() for each call to the kernel. For example, when calling the out-of-place transpose operation which is called twice, the pointer to the memory object must be reversed the second time around.

The kernels in this program is called using clEnqueueNDRangeKernel() to operate on the data in a data parallel manner. When this is called, the number work items, whose values differ depending on the kernel used, must be set beforehand. To reduce careless errors and to make the code more readable, a setWorkSize() function is implemented in this program.

```
        int setWorkSize(size_t* gws, size_t* lws, cl_int x, cl_int y)
```

The program contains a set of procedures that are repeated numerous times, namely the bit reversal and the butterfly operation, for both FFT and IFFT. These procedures are all grouped into one function fftCore().

```
        int fftCore(cl_mem dst, cl_mem src, cl_mem spin, cl_int m, enum Mode direct
ion)
```

This function takes memory objects for the input, output, and the spin factor, the sample number normalized by the log of 2, and the FFT direction. This function can be used for 1-D FFT if the arguments are appropriately set.

Lastly, we will briefly explain the outputting of the processed data to an image. The format used for the image is PGM, which is a gray-scale format that requires 8 bits for each pixel. The data structure is quite simple and intuitive to use. We will add a new file, called "pgm.h". This file will define numerous functions and structs to be used in our program. First is the pgm_t struct.

```
typedef struct _pgm_t {
   int width;
   int height;
   unsigned char *buf;
} pgm_t;
```

The width and the height gets the size of the image, and buf gets the image data. This can read in or written to a file using the following functions.

```
readPGM(pgm_t* pgm, const char*, filename);
writePGM(pgm_t* pgm, const char* filename);
```

Since each pixel of the PGM is stored as unsigned char, a conversion would need to be performed to represent the pixel information in 8 bits.

```
normalizePGM(pgm_t* pgm, double* data);
```

The full pgm.h file is shown below in List 6.10.

List 6.10: pgm.h

```
#ifndef _PGM_H_
#define _PGM_H_

#include <math.h>
#include <string.h>

#define PGM_MAGIC "P5"

#ifdef _WIN32
#define STRTOK_R(ptr, del, saveptr) strtok_s(ptr, del, saveptr)
#else
#define STRTOK_R(ptr, del, saveptr) strtok_r(ptr, del, saveptr)
#endif

typedef struct _pgm_t {
int width;
int height;
unsigned char *buf;
} pgm_t;

int readPGM(pgm_t* pgm, const char* filename)
{
char *token, *pc, *saveptr;
char *buf;
size_t bufsize;
char del[] = " \t\n";
unsigned char *dot;

long begin, end;
```

```
int filesize;
int i, w, h, luma, pixs;


FILE* fp;
if ((fp = fopen(filename, "rb"))==NULL) {
fprintf(stderr, "Failed to open file\n");
return -1;
}

fseek(fp, 0, SEEK_SET);
begin = ftell(fp);
fseek(fp, 0, SEEK_END);
end = ftell(fp);
filesize = (int)(end - begin);

buf = (char*)malloc(filesize * sizeof(char));
fseek(fp, 0, SEEK_SET);
bufsize = fread(buf, filesize * sizeof(char), 1, fp);

fclose(fp);

token = (char *)STRTOK_R(buf, del, &saveptr);
if (strncmp(token, PGM_MAGIC, 2) != 0) {
return -1;
}

token = (char *)STRTOK_R(NULL, del, &saveptr);
if (token[0] == '#' ) {
token = (char *)STRTOK_R(NULL, "\n", &saveptr);
token = (char *)STRTOK_R(NULL, del, &saveptr);
}

w = strtoul(token, &pc, 10);
token = (char *)STRTOK_R(NULL, del, &saveptr);
h = strtoul(token, &pc, 10);
token = (char *)STRTOK_R(NULL, del, &saveptr);
luma = strtoul(token, &pc, 10);

```

```
token = pc + 1;
pixs = w * h;

pgm->buf = (unsigned char *)malloc(pixs * sizeof(unsigned char));

dot = pgm->buf;

for (i=0; i< pixs; i++, dot++) {
*dot = *token++;
}

pgm->width = w;
pgm->height = h;

return 0;
}

int writePGM(pgm_t* pgm, const char* filename)
{
int i, w, h, pixs;
FILE* fp;
unsigned char* dot;

w = pgm->width;
h = pgm->height;
pixs = w * h;

if ((fp = fopen(filename, "wb+")) ==NULL) {
fprintf(stderr, "Failed to open file\n");
return -1;
}

fprintf (fp, "%s\n%d %d\n255\n", PGM_MAGIC, w, h);

dot = pgm->buf;

for (i=0; i<pixs; i++, dot++) {
putc((unsigned char)*dot, fp);
}
```

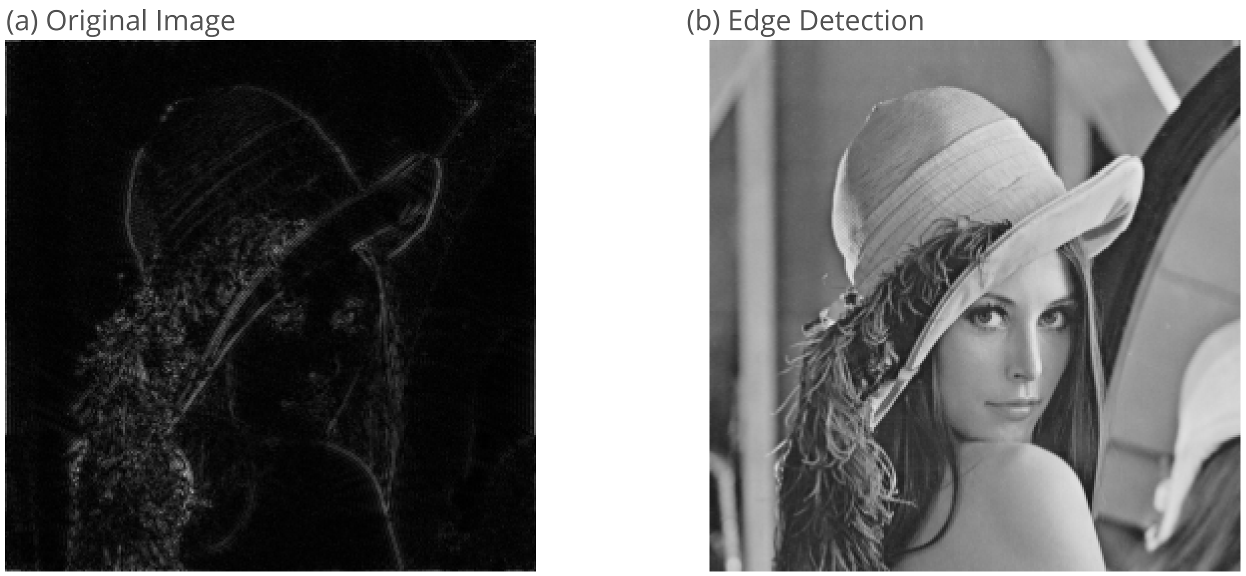
```

fclose(fp);

return 0;
}

int normalizeD2PGM(pgm_t* pgm, double* x)
{
int i, j, w, h;

w = pgm->width;
h = pgm->height;

pgm->buf = (unsigned char*)malloc(w * h * sizeof(unsigned char));

double min = 0;
double max = 0;
for (i=0; i < h; i++) {
for (j=0; j < w; j++) {
if (max < x[i*w+j])
max = x[i*w+j];
if (min > x[i*w+j])
min = x[i*w+j];
}
}

for (i=0; i < h; i++) {
for (j=0; j < w; j++) {
if((max-min)!=0)
pgm->buf[i*w+j] = (unsigned char)(255*(x[i*w+j]-min)/(max-min));
else
pgm->buf[i*w+j]= 0;
}
}

return 0;
}

int normalizeF2PGM(pgm_t* pgm, float* x)
```

```
{
int i, j, w, h;

w = pgm->width;
h = pgm->height;

pgm->buf = (unsigned char*)malloc(w * h * sizeof(unsigned char));

float min = 0;
float max = 0;
for (i=0; i < h; i++) {
for (j=0; j < w; j++) {
if (max < x[i*w+j])
max = x[i*w+j];
if (min > x[i*w+j])
min = x[i*w+j];
}
}

for (i=0; i < h; i++) {
for (j=0; j < w; j++) {
if((max-min)!=0)
pgm->buf[i*w+j] = (unsigned char)(255*(x[i*w+j]-min)/(max-min));
else
pgm->buf[i*w+j]= 0;
}
}

return 0;
}

int destroyPGM(pgm_t* pgm)
{
if (pgm->buf) {
free(pgm->buf);
}

return 0;
}
```

```

#endif /* _PGM_H_ */
```

When all the sources are compiled and executed on an image, the picture shown in Figure 6.7(a) becomes the picture shown in Figure 6.7(b). The edges in the original picture become white, while everything else becomes black.

Figure 6.7: Edge Detection



Measuring Execution Time

OpenCL is an abstraction layer that allows the same code to be executed on different platforms, but this only guarantee that program can be executed. The speed of execution is dependent on the device, as well as the type of parallelism used. Therefore, in order to get the maximum performance, a device and parallelism dependent tuning must be performed.

In order to tune a program, the execution time must be measured, since it would otherwise be very difficult to see the result. We will now show how this can be done within the OpenCL framework for portability. Time measurement can be done in OpenCL, which is triggered by event objects associated with certain clEnqueue-type commands. This code is shown in List 6.11.

List 6.11: Time measurement using event objects

```
cl_context context;
cl_command_queue queue;
cl_event event;
cl_ulong start;
cl_ulong end;
…
queue = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &ret);
…
ret = clEnqueueWriteBuffer(queue, mobj, CL_TRUE, 0, MEM_SIZE, m, 0, NULL, &event);
…
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
printf(" memory buffer write: %10.5f [ms]\n", (end - start)/1000000.0);
```

The clWaitForEvents keeps the next line of the code to be executed until the specified events in the event list has finished its execution. The first argument gets the number of events to wait for, and the 2nd argument gets the pointer to the event list.

You should now be able to measure execution times using OpenCL.

Index Parameter Tuning

Recall that the number of work items and work groups had to be specified when executing data parallel kernels. This section focuses on what these values should be set to for optimal performance.

There is quite a bit of freedom when setting these values. For example, 512 work items can be split up into 2 work groups each having 256 work items, or 512 work groups each having 1 work item.

This raises the following questions:

1. What values are allowed for the number of work groups and work items?

2. What are the optimal values to use for the number of work groups and work items?

The first question can be answered using the clGetDeviceInfo() introduced in Section 5.1. The code is shown in List 6.13 below.

List 6.13: Get maximum values for the number of work groups and work items

```
ret = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, gws, lws, 0, NULL, &event);
clWaitForEvents(1, &event);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong),
&start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NU
LL);
```

The first clGetDeviceInfo() gets the maximum number of dimensions allowed for the work item. This returns either 1, 2 or 3.

The second clGetDeviceInfo() gets the maximum values that can be used for each dimensions of the work item. The smallest value is [1,1,1].

The third clGetDeviceInfo() gets the maximum work item size that can be set for each work group. The smallest value for this is 1.

Running the above code using NVIDIA OpenCL would generate the results shown below.

```
Max work-item dimensions : 3
Max work-item sizes       : 512 512 64
Max work-group size       : 512
```

As mentioned before, the number of work items and the work groups must be set before executing a kernel. The global work item index and the local work item index each correspond to the work item ID of all the submitted jobs, and the work item ID within the work group.

For example, if the work item is 512 x 512, the following combination is possible. (gws=global work-item size, lws=local work-item size).

```
gws[] = {512,512,1}
lws[] = {1,1,1}
```

The following is also possible:

```
gws[] = {512,512,1}
lws[] = {16,16,1}
```

Yet another combination is:

```
gws[] = {512,512,1}
lws[] = {256,1,1}
```

The following example seems to not have any problems at a glance, but would result in an error, since the size of the work-group size exceeds that of the allowed size (32*32 = 1024 > 512).

```
gws[] = {512,512,1}
lws[] = {32,32,1}
```

Hopefully you have found the above to be intuitive. The real problem is figuring out the optimal combination to use.

At this point, we need to look at the hardware architecture of the actual device. As discussed in Chapter 2, the OpenCL architecture assumes the device to contain compute unit(s), which is made up of several processing elements. The work-item corresponds to the processing element, and the work group corresponds to the compute unit. In other words, a work group gets executed on a compute unit, and a work-item gets executed on a processing element.

This implies that the knowledge of the number of compute units and processing elements is required in deducing the optimal combination to use for the local work group size. These can be found using clGetDeviceInfo(), as shown in List 6.14.

List 6.14: Find the number of compute units

```
cl_uint comput_unit = 0;
```

```
ret = clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint), &com
pute_unit, NULL);
```

In NVIDIA GPUs, a compute unit corresponds to what is known as Streaming Multi-processor (SM). GT 200-series GPUs such as Tesla C1060 and GTX285 contain 30 compute units. A processing element corresponds to what is called CUDA cores. Each compute unit contains 8 processing elements, but 32 processes can logically be performed in parallel.

The following generations can be made from the above information:

- Processor elements can be used efficiently if the number of work items within a work group is a multiple of 32.
- All compute units can be used if the number of work groups is greater than 30.

We will now go back to the subject of FFT. We will vary the number of work-items per work group (local work-group size), and see how it affects the processing time. For simplicity, we will use a 512 x 512 image. Execution time was measured for 1, 16, 32, 64, 128, 256, and 512 local work-group size (Table 6.1).

Table 6.1: Execution time when varying lws (units in ms)

Process	1	16	32	64	128	256	512
membuf write	0.54	0.53	0.36	0.52	0.53	0.54	0.53
spinFactor	0.01	0.01	0.01	0.01	0.01	0.01	0.01
bitReverse	6.54	0.60	0.47	0.47	0.49	0.50	0.51
butterfly	37.75	3.12	2.87	3.02	3.41	3.60	3.88
normalize	2.86	0.20	0.12	0.09	0.09	0.10	0.10
transpose	2.83	1.53	1.49	1.42	1.07	0.95	0.96
highPassFilter	1.51	0.10	0.07	0.06	0.06	0.06	0.07
membuf read	0.62	0.57	0.57	0.61	0.58	0.56	0.58

As you can see, the optimal performance occurs when the local work-group size is a multiple of 32. This is due to the fact that 32 processes can logically be performed in parallel for each compute unit. The performance does not suffer significantly when the local work-group size is increased, since the NVIDIA GPU hardware performs the thread switching.

The FFT algorithm is a textbook model of a data parallel algorithm. We performed parameter tuning specifically for the NVIDIA GPU, but the optimal value would vary depending on the architecture of the device. For example, Apple's OpenCL for multi-core CPUs only allows 1 work-item for each work group. We can only assume that the parallelization is performed by the framework itself. At the time of this writing (December 2009), Mac OS X Snow Leopard comes with an auto-parallelization framework called the "Grand Central Dispatch", which lead us to assume a similar auto-parallelization algorithms are implemented to be used within the OpenCL framework.

We will now conclude our case study of the OpenCL implementation of FFT. Note that the techniques used so far are rather basic, but when combined wisely, a complex algorithm like the FFT can be implemented to be run efficiently over an OpenCL device. The implemented code should work over any platform where an OpenCL framework exists.

Previous | Next