OpenCL Sorting

Eric Bainville - June 2011

Introduction

This page relates some experiments I made on OpenCL sorting algorithms.

Sorting algorithms are not the best suited for the GPU: they usually require lots of comparisons, and access memory through irregular patterns. However, since sorting is a basic building block of many algorithms, it may be desirable to have a GPU implementation. Some GPU implementations are actually faster than the CPU (assuming data is already resident in the GPU).

Contents

1. Introduction - Introduction (this page).
2. Parallel selection - Parallel selection sort, some variations on a simple algorithm.
3. Parallel selection, local - Partial parallel selection sort inside a workgroup.
4. Parallel bitonic, local - Partial bitonic sort inside a workgroup.
5. Parallel bitonic I - Full bitonic sort, working in global memory and registers.
6. Parallel bitonic II - Full bitonic sort, adding local memory.
7. Parallel merge, local - Partial merge sort inside a workgroup.

References

Writing this article was an excellent opportunity to re-open the third volume of the Art of Computer Programming by Donald E. Knuth. Googling for GPU sorting will return the most interesting references on the subject. The Wikipedia Sorting algorithm page provides some useful information too. Efficient algorithms for GPU sorting can be found in Designing Efficient Sorting Algorithms for Manycore GPUs by Satish, Harris, and Garland. The publications of Duane Merrill describe more recent high performance sorting on GPU (reaching 775 Mkey/s on a NVIDIA GTX 480 for Key+Value sorting).

Downloads

The package contains all host and kernel code, with projects for Linux and Visual Studio 2010. It includes a small OpenCL wrapper providing basic useful functions (MiniCL). The test program will check all the kernels (it takes some time).

OpenCLsorting-20110625.zip (24 KB), June 2011, all kernels up to Parallel Bitonic II.

Benchmarks

We will sort an input array of **N** 32-bit unsigned integers keys, with and without an associated 32-bit integer value. For a given **N** we measure the elapsed wall clock time **T**, and performance $P=10^{-6}.N/T$ in Mkey/s (million keys per second). To simplify the code, **N** will always be a power of 2. The measurement loop repeats the sort until a minimal elapsed time is reached, as in the following:

```
double t0 = getRealTime(); // wallclock time (s)
double dt = 0;
double nit = 0; // total number of calls to sort()
// Double number of iterations until we reach a minimum elapsed time (0.5s here)
for (int it=1;it<=1<<20;it<<=1)
{
  for (int i=0;i<it;i++)
  {
    ok &= algo.sort(c,targetDevice,n,inBuffer,outBuffer); // sort
    c->finish(targetDevice); // wait for kernels to terminate
  }
  dt = getRealTime() - t0; // elapsed time (s)
  nit += (double)it;
  if (dt > 0.5) break; // min time reached?
}
dt /= nit;  // elapsed time per algorithm invocation
```

The machine used for our experiments has the following configuration: **Core i7 950 @4000MHz 12GB RAM, AMD HD6970 Cayman @880Mhz 2GB RAM @1375MHz, Windows 7 64-bit and AMD Catalyst 11.6**.

Upper bound of the sorting rate

To get an upper bound of the expected performance, let's imagine we will run **K** successive kernels, each run reading and writing the entire key array, **(4+4).N** bytes. Assuming we can reach the advertised 176 GB/s for the HD6970, we get **rate < 22,000/K Mkey/s**.

OpenCL Sorting

Eric Bainville - June 2011

> BEALTO
Home
Eric Bainville

> FPGA
FPGA Simple UART

> OpenCL
OpenCL Sorting
• Introduction
• Parallel selection
• Parallel selection, local
• Parallel bitonic, local
• Parallel bitonic I
• Parallel bitonic II
• Parallel merge, local
OpenCL FFT
OpenCL FFT (OLD)
OpenCL GEMV
OpenCL Multiprecision
GPU Mandelbrot Set
GPU Benchmarks

> Other
SSE gradient
SSE dot product
AMD64 Multiprecision
Intel64 Multiprecision
POV-Ray Buttons/Logos
Projective Geometry
Quaternions

Sitemap

Sign In

Parallel Selection Sort

Basic implementation

Let's begin with a very simple algorithm. I could not find a better name for it than "parallel selection sort"; contact me if you have a better suggestion for the naming. We run **N** threads. Thread **i** iterates on the entire input vector to find the output position **pos** of value $\mathbf{in_i}$. Finally, thread **i** and writes $\mathbf{in_i}$ into $\mathbf{out_{pos}}$:

```
__kernel void ParallelSelection(__global const data_t * in,__global data_t * out)
{
  int i = get_global_id(0); // current thread
  int n = get_global_size(0); // input size
  data_t iData = in[i];
  uint iKey = keyValue(iData);
  // Compute position of in[i] in output
  int pos = 0;
  for (int j=0;j<n;j++)
  {
    uint jKey = keyValue(in[j]); // broadcasted
    bool smaller = (jKey < iKey) || (jKey == iKey && j < i);  // in[j] < in[i] ?
    pos += (smaller)?1:0;
  }
  out[pos] = iData;
}
```

ParallelSelection		
$\log_2(N)$	Key Only	Key+Value
8	1.00	0.99
9	1.23	1.22
10	1.36	1.36
11	1.44	1.44
12	1.50	1.50
13	1.48	1.49
14	1.17	1.31
15	0.68	0.72
16	0.37	0.38
17	0.22	0.22
18	0.12	0.12
19	0.06	0.06

Performance of the ParallelSelection kernel, Mkey/s.

This algorithm is obviously highly ineffective. Considering the total I/O is $\mathbf{2*N+N^2}$ records, the total memory throughput increases, to reach 127 GB/s for $\mathbf{N=2^{19}}$. Note that there is no difference in having 32-bit or 64-bit records here.

Using local memory

Instead of having all threads read values from global memory, we could try to preload blocks of values in local memory, and use them in all threads inside a workgroup. In the following code, we load `BLOCK_FACTOR` input records for each thread: each workgroup will load the input data by blocks of `BLOCK_FACTOR * workgroup_size` records.

```
__kernel void ParallelSelection_Blocks(__global const data_t * in,__global data_t * out,__local uint * aux)
{
  int i = get_global_id(0); // current thread
  int n = get_global_size(0); // input size
  int wg = get_local_size(0); // workgroup size
  data_t iData = in[i]; // input record for current thread
  uint iKey = keyValue(iData); // input key for current thread
  int blockSize = BLOCK_FACTOR * wg; // block size

  // Compute position of iKey in output
  int pos = 0;
  // Loop on blocks of size BLOCKSIZE keys (BLOCKSIZE must divide N)
  for (int j=0;j<n;j+=blockSize)
  {
    // Load BLOCKSIZE keys using all threads (BLOCK_FACTOR values per thread)
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int index=get_local_id(0);index<blockSize;index+=wg)
      aux[index] = keyValue(in[j+index]);
```

```
    // Loop on all values in AUX
    for (int index=0;index<blockSize;index++)
    {
      uint jKey = aux[index]; // broadcasted, local memory
      bool smaller = (jKey < iKey) || ( jKey == iKey && (j+index) < i ); // in[j] < in[i] ?
      pos += (smaller)?1:0;
    }
  }
  out[pos] = iData;
}
```

Note that **both** **`barrier`** instructions are required, the first one ensures all threads have finished the processing loop before loading a new block, and the second one ensures the block is entirely loaded before starting the next processing loop.

ParallelSelection_Blocks						
	BLOCK_FACTOR value					
$\log_2(N)$	**1**	**2**	**4**	**8**	**16**	**32**
8	1.80					
9	2.66	2.65				
10	3.41	3.42	3.34			
11	3.91	3.94	3.96	3.79		
12	4.24	4.28	4.27	4.16	4.15	
13	2.36	2.37	2.37	2.38	2.39	2.28
14	1.61	1.61	1.60	1.59	1.62	1.55
15	0.82	0.82	0.82	0.79	0.81	0.78
16	0.45	0.45	0.45	0.45	0.45	0.43
17	0.22	0.23	0.23	0.23	0.23	0.21
18	0.12	0.12	0.12	0.12	0.12	0.11
19	0.06	0.06	0.06	0.06	0.06	0.05

Performance of the ParallelSelection_Blocks kernel (Key+Value sort), Mkey/s.

Performance is a little better, and reaches **4.28** Mkey/s thanks to the higher speed of the local memory, but we still are very far from good.

OpenCL Sorting

Eric Bainville - June 2011

Parallel selection, local

Let's now focus on sorting in local memory. We execute **N** threads in workgroups of **WG** threads, and each workgroup sorts a segment of **WG** input records. The output contains **N/WG** ordered sequences. The following kernel is adapted from the ParallelSelection_Blocks kernel, and uses the same algorithm to sort each subsequence of length **WG**.

```
__kernel void ParallelSelection_Local(__global const data_t * in,__global data_t * out,__local data_t * aux)
{
  int i = get_local_id(0); // index in workgroup
  int wg = get_local_size(0); // workgroup size = block size

  // Move IN, OUT to block start
  int offset = get_group_id(0) * wg;
  in += offset; out += offset;

  // Load block in AUX[WG]
  data_t iData = in[i];
  aux[i] = iData;
  barrier(CLK_LOCAL_MEM_FENCE);

  // Find output position of iData
  uint iKey = getKey(iData);
  int pos = 0;
  for (int j=0;j<wg;j++)
  {
    uint jKey = getKey(aux[j]);
    bool smaller = (jKey < iKey) || ( jKey == iKey && j < i ); // in[j] < in[i] ?
    pos += (smaller)?1:0;
  }

  // Store output
  out[pos] = iData;
}
```

ParallelSelection_Local	
WG	**Top speed**
1	133
2	209
4	287
8	336
16	367
32	384
64	392
128	199
256	100

Performance of the ParallelSelection_Local kernel (Key+Value sort), Mkey/s.

Each thread performs **O(1)** global memory accesses, and **O(WG)** local memory accesses. When **WG** doubles, the work quantity doubles too, meaning the processing rate is expected to be halved. This is verified for **WG>32** here.

OpenCL Sorting : Parallel selection | Top of Page | OpenCL Sorting : Parallel bitonic, local

> BEALTO
Home
Eric Bainville

> FPGA
FPGA Simple UART

> OpenCL
OpenCL Sorting
• Introduction
• Parallel selection
• Parallel selection, local
• Parallel bitonic, local
• Parallel bitonic I
• Parallel bitonic II
• Parallel merge, local
OpenCL FFT
OpenCL FFT (OLD)
OpenCL GEMV
OpenCL Multiprecision
GPU Mandelbrot Set
GPU Benchmarks

> Other
SSE gradient
SSE dot product
AMD64 Multiprecision
Intel64 Multiprecision
POV-Ray Buttons/Logos
Projective Geometry
Quaternions

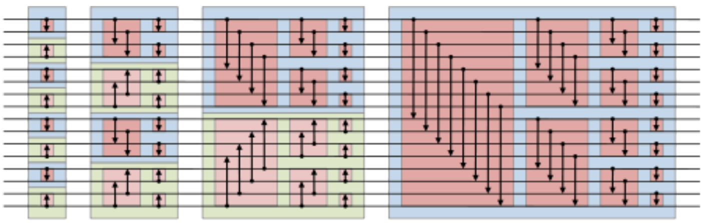
Sitemap

Sign In

Parallel bitonic, local

Here again, we sort in local memory. We execute **N** threads in workgroups of **WG** threads, and each workgroup sorts a segment of **WG** input records. The output contains **N/WG** ordered sequences.

Ken Batcher's bitonic sorting network is described in Knuth's book, and nicely illustrated in the Wikipedia page:



Bitonic sorter network (from Wikipedia).

The code below is a direct translation of the network pictured in the figure, where each thread computes exactly one element (i.e. we have two threads for each comparator).

```
__kernel void ParallelBitonic_Local(__global const data_t * in,__global data_t * out,__local data_t * aux)
{
  int i = get_local_id(0); // index in workgroup
  int wg = get_local_size(0); // workgroup size = block size, power of 2

  // Move IN, OUT to block start
  int offset = get_group_id(0) * wg;
  in += offset; out += offset;

  // Load block in AUX[WG]
  aux[i] = in[i];
  barrier(CLK_LOCAL_MEM_FENCE); // make sure AUX is entirely up to date

  // Loop on sorted sequence length
  for (int length=1;length<wg;length<<=1)
  {
    bool direction = ((i & (length<<1)) != 0); // direction of sort: 0=asc, 1=desc
    // Loop on comparison distance (between keys)
    for (int inc=length;inc>0;inc>>=1)
    {
      int j = i ^ inc; // sibling to compare
      data_t iData = aux[i];
      uint iKey = getKey(iData);
      data_t jData = aux[j];
      uint jKey = getKey(jData);
      bool smaller = (jKey < iKey) || ( jKey == iKey && j < i );
      bool swap = smaller ^ (j < i) ^ direction;
      barrier(CLK_LOCAL_MEM_FENCE);
      aux[i] = (swap)?jData:iData;
      barrier(CLK_LOCAL_MEM_FENCE);
    }
  }

  // Write output
  out[i] = aux[i];
}
```

We don't need to read **iData** at each step, since the thread can keep track of its value when it is changed. The modified version is a little faster, measured as follows:

ParallelBitonic_Local	
WG	**Top speed**
1	191
2	163
4	166
8	192
16	246
32	341
64	496
128	380
256	291

> BEALTO
Home
Eric Bainville

> FPGA
FPGA Simple UART

> OpenCL
OpenCL Sorting
• Introduction
• Parallel selection
• Parallel selection, local
• Parallel bitonic, local
• Parallel bitonic I
• Parallel bitonic II
• Parallel merge, local
OpenCL FFT
OpenCL FFT (OLD)
OpenCL GEMV
OpenCL Multiprecision
GPU Mandelbrot Set
GPU Benchmarks

> Other
SSE gradient
SSE dot product
AMD64 Multiprecision
Intel64 Multiprecision
POV-Ray Buttons/Logos
Projective Geometry
Quaternions

Sitemap

Sign In

OpenCL Sorting

Eric Bainville - June 2011

> BEALTO
Home
Eric Bainville

> FPGA
FPGA Simple UART

> OpenCL
OpenCL Sorting
• Introduction
• Parallel selection
• Parallel selection, local
• Parallel bitonic, local
• Parallel bitonic I
• Parallel bitonic II
• Parallel merge, local
OpenCL FFT
OpenCL FFT (OLD)
OpenCL GEMV
OpenCL Multiprecision
GPU Mandelbrot Set
GPU Benchmarks

> Other
SSE gradient
SSE dot product
AMD64 Multiprecision
Intel64 Multiprecision
POV-Ray Buttons/Logos
Projective Geometry
Quaternions

Sitemap

Sign In

Parallel bitonic

The simplicity and regularity of the bitonic sort make it an ideal candidate for experiments. Let's transform the inner loop of the ParallelBitonic_Local kernel into a kernel operating on global memory.

```
__kernel void ParallelBitonic_A(__global const data_t * in,__global data_t * out,int inc,int dir)
{
  int i = get_global_id(0); // thread index
  int j = i ^ inc; // sibling to compare

  // Load values at I and J
  data_t iData = in[i];
  uint iKey = getKey(iData);
  data_t jData = in[j];
  uint jKey = getKey(jData);

  // Compare
  bool smaller = (jKey < iKey) || ( jKey == iKey && j < i );
  bool swap = smaller ^ (j < i) ^ ((dir & i) != 0);

  // Store
  out[i] = (swap)?jData:iData;
}
```

The loop on **length** and **inc** is now in the host code: we call **$O(\log^2(N))$** kernels, swapping input/output buffers and enqueuing a barrier at each invocation.

ParallelBitonic_A	
$\log_2(N)$	Key+Value
9	1.1
10	1.9
13	10
14	17
17	37
18	37
21	30
22	28
25	21

Performance of the ParallelBitonic_A kernel, Mkey/s.

Each thread makes 3 global memory accesses, and we execute **$L*(L+1)/2$** kernels, where **$L=\log_2(N)$**. For **$L=22$**, we reach an impressive 169 GB/s of global memory I/O rate.

In the above kernel, since we run one thread per value, we perform each comparison **i<>j** two times: one in thread **i**, and the other one in thread **j**. We can make this comparison in a single thread, which will manage both values: 2 read, 2 write, and 1 comparison instead of 4 read, 2 write, and 2 comparisons. This leads to the following kernel, requiring **N/2** threads at each pass, and modifying data in-place:

```
#define ORDER(a,b) { bool swap = reverse ^ (getKey(a)<getKey(b)); \
  data_t auxa = a; data_t auxb = b; a = (swap)?auxb:auxa; b = (swap)?auxa:auxb; }

__kernel void ParallelBitonic_B2(__global data_t * data,int inc,int dir)
{
  int t = get_global_id(0); // thread index
  int low = t & (inc - 1); // low order bits (below INC)
  int i = (t<<1) - low; // insert 0 at position INC
  bool reverse = ((dir & i) == 0); // asc/desc order
  data += i; // translate to first value

  // Load
  data_t x0 = data[  0];
  data_t x1 = data[inc];

  // Sort
  ORDER(x0,x1)

  // Store
  data[0  ] = x0;
  data[inc] = x1;
}
```

This kernel runs at 49 Mkey/s for Key+Value, exceeding 160 GB/s of global memory I/O rate, and 73 Mkey/s for Key sorting only. The speed limitation comes from the large number of kernels invoked. For example for

We could collapse two consecutive calls of the kernel with **inc** and **inc/2**. Each thread would process 4 values in-place, performing 4 comparisons. This is illustrated in the following kernel:

```
__kernel void ParallelBitonic_B4(__global data_t * data,int inc,int dir)
{
  inc >> 1;
  int t = get_global_id(0); // thread index
  int low = t & (inc - 1); // low order bits (below INC)
  int i = ((t - low) << 2) + low; // insert 00 at position INC
  bool reverse = ((dir & i) == 0); // asc/desc order
  data += i; // translate to first value

  // Load
  data_t x0 = data[    0];
  data_t x1 = data[  inc];
  data_t x2 = data[2*inc];
  data_t x3 = data[3*inc];

  // Sort
  ORDER(x0,x2)
  ORDER(x1,x3)
  ORDER(x0,x1)
  ORDER(x2,x3)

  // Store
  data[    0] = x0;
  data[  inc] = x1;
  data[2*inc] = x2;
  data[3*inc] = x3;
}
```

Using the B2 and B4 kernels, parallel bitonic sort reaches 78 Mkey/s for Key+Value, and 115 Mkey/s for Key only. Let's continue in this direction with kernels processing 8 and 16 inputs at a time:

```
#define ORDERV(x,a,b) { bool swap = reverse ^ (getKey(x[a])<getKey(x[b])); \
      data_t auxa = x[a]; data_t auxb = x[b]; \
      x[a] = (swap)?auxb:auxa; x[b] = (swap)?auxa:auxb; }
#define B2V(x,a) { ORDERV(x,a,a+1) }
#define B4V(x,a) { for (int i4=0;i4<2;i4++) { ORDERV(x,a+i4,a+i4+2) } B2V(x,a) B2V(x,a+2) }
#define B8V(x,a) { for (int i8=0;i8<4;i8++) { ORDERV(x,a+i8,a+i8+4) } B4V(x,a) B4V(x,a+4) }
#define B16V(x,a) { for (int i16=0;i16<8;i16++) { ORDERV(x,a+i16,a+i16+8) } B8V(x,a) B8V(x,a+8) }

__kernel void ParallelBitonic_B8(__global data_t * data,int inc,int dir)
{
  inc >> 2;
  int t = get_global_id(0); // thread index
  int low = t & (inc - 1); // low order bits (below INC)
  int i = ((t - low) << 3) + low; // insert 000 at position INC
  bool reverse = ((dir & i) == 0); // asc/desc order
  data += i; // translate to first value

  // Load
  data_t x[8];
  for (int k=0;k<8;k++) x[k] = data[k*inc];

  // Sort
  B8V(x,0)

  // Store
  for (int k=0;k<8;k++) data[k*inc] = x[k];
}

__kernel void ParallelBitonic_B16(__global data_t * data,int inc,int dir)
{
  inc >> 3;
  int t = get_global_id(0); // thread index
  int low = t & (inc - 1); // low order bits (below INC)
  int i = ((t - low) << 4) + low; // insert 0000 at position INC
  bool reverse = ((dir & i) == 0); // asc/desc order
  data += i; // translate to first value

  // Load
  data_t x[16];
  for (int k=0;k<16;k++) x[k] = data[k*inc];

  // Sort
  B16V(x,0)

  // Store
  for (int k=0;k<16;k++) data[k*inc] = x[k];
}
```

The use of loops does not slow down the generated code. Using B2, B4, and B8 kernels, we can reach 103 Mkey/s for Key+Value, and 160 Mkey/s for Key only. Unfortunately, a probable bug in the driver (Catalyst 11.5 and 11.6) does not allow to execute the B16 kernel without errors in the output. It runs correctly on a NVIDIA GPU.

ParallelBitonic		
Kernels	**Key+Value**	**Key**
B2	49	73
B2+B4	78	115
B2+B4+B8	103	160
B2+B4+B8+B16	FAIL	FAIL

When **inc** becomes small enough, all sorting is done locally, and a single workgroup processes the same data. It is then possible to terminate the **inc** loop inside a single kernel, inserting barriers to synchronize the threads. The second step would be to replace global memory I/O by local memory I/O for intermediate states. We will implement this in the next page: Parallel bitonic II.

OpenCL Sorting

Eric Bainville - June 2011

> BEALTO
Home
Eric Bainville

> FPGA
FPGA Simple UART

> OpenCL
OpenCL Sorting
• Introduction
• Parallel selection
• Parallel selection, local
• Parallel bitonic, local
• Parallel bitonic I
• Parallel bitonic II
• Parallel merge, local
OpenCL FFT
OpenCL FFT (OLD)
OpenCL GEMV
OpenCL Multiprecision
GPU Mandelbrot Set
GPU Benchmarks

> Other
SSE gradient
SSE dot product
AMD64 Multiprecision
Intel64 Multiprecision
POV-Ray Buttons/Logos
Projective Geometry
Quaternions

Sitemap

Sign In

Parallel bitonic, local memory acceleration

We will now use all three levels of the GPU memory system: global, local, and registers. We will operate in global memory until **inc** becomes smaller than the workgroup size. From this point, we can terminate the loop on **inc** inside a single kernel. The following kernel C2(pre) does this, each thread processing two values at each iteration (it is based on kernel B2). To sort the entire input vector, we now call a mix of B2, B4, B8,... and C2(pre) to terminate each **inc** loop when **inc ≤ WG**.

```
__kernel void ParallelBitonic_C2_pre(__global data_t * data,int inc,int dir)
{
  int t = get_global_id(0); // thread index

  // Terminate the INC loop inside the workgroup
  for ( ;inc>0;inc>>=1)
  {
    int low = t & (inc - 1); // low order bits (below INC)
    int i = (t<<1) - low; // insert 0 at position INC
    bool reverse = ((dir & i) == 0); // asc/desc order

    barrier(CLK_GLOBAL_MEM_FENCE);

    // Load
    data_t x0 = data[i];
    data_t x1 = data[i+inc];

    // Sort
    ORDER(x0,x1)

    barrier(CLK_GLOBAL_MEM_FENCE);

    // Store
    data[i] = x0;
    data[i+inc] = x1;
  }
}
```

This kernel runs at 40 Mkey/s, a little slower than kernel B2. Now let's replace all intermediate storage in global memory by local memory accesses. This is kernel C2:

```
__kernel void ParallelBitonic_C2(__global data_t * data,int inc0,int dir,__local data_t * aux)
{
  int t = get_global_id(0); // thread index
  int wgBits = 2*get_local_size(0) - 1; // bit mask to get index in local memory AUX (size is 2*WG)

  for (int inc=inc0;inc>0;inc>>=1)
  {
    int low = t & (inc - 1); // low order bits (below INC)
    int i = (t<<1) - low; // insert 0 at position INC
    bool reverse = ((dir & i) == 0); // asc/desc order
    data_t x0,x1;

    // Load
    if (inc == inc0)
    {
      // First iteration: load from global memory
      x0 = data[i];
      x1 = data[i+inc];
    }
    else
    {
      // Other iterations: load from local memory
      barrier(CLK_LOCAL_MEM_FENCE);
      x0 = aux[i & wgBits];
      x1 = aux[(i+inc) & wgBits];
    }

    // Sort
    ORDER(x0,x1)

    // Store
    if (inc == 1)
    {
      // Last iteration: store to global memory
      data[i] = x0;
      data[i+inc] = x1;
    }
    else
    {
      // Other iterations: store to local memory
      barrier(CLK_LOCAL_MEM_FENCE);
      aux[i & wgBits] = x0;
      aux[(i+inc) & wgBits] = x1;
    }
  }
}
```

The local memory array **aux** has dimension 2*WG since each thread of the workgroup handles 2 values. Kernel C2 runs at 92 Mkey/s (combined with B2+B4+B8 for larger **inc** values) for Key+Value sorting, and 108 Mkey/s for Key only. Let's extend this to a C4 kernel, processing 4 values per thread, and called only when **inc** is 8, 32, 128:

```
__kernel void ParallelBitonic_C4(__global data_t * data,int inc0,int dir,__local data_t * aux)
{
  int t = get_global_id(0); // thread index
  int wgBits = 4*get_local_size(0) - 1; // bit mask to get index in local memory AUX (size is 4*WG)
  int inc,low,i;
  bool reverse;
  data_t x[4];

  // First iteration, global input, local output
  inc = inc0>>1;
  low = t & (inc - 1); // low order bits (below INC)
  i = ((t - low) << 2) + low; // insert 00 at position INC
  reverse = ((dir & i) == 0); // asc/desc order
  for (int k=0;k<4;k++) x[k] = data[i+k*inc];
  B4V(x,0);
  for (int k=0;k<4;k++) aux[(i+k*inc) & wgBits] = x[k];
  barrier(CLK_LOCAL_MEM_FENCE);

  // Internal iterations, local input and output
  for ( ;inc>1;inc>>=2)
  {
    low = t & (inc - 1); // low order bits (below INC)
    i = ((t - low) << 2) + low; // insert 00 at position INC
    reverse = ((dir & i) == 0); // asc/desc order
    for (int k=0;k<4;k++) x[k] = aux[(i+k*inc) & wgBits];
    B4V(x,0);
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int k=0;k<4;k++) aux[(i+k*inc) & wgBits] = x[k];
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  // Final iteration, local input, global output, INC=1
  i = t << 2;
  reverse = ((dir & i) == 0); // asc/desc order
  for (int k=0;k<4;k++) x[k] = aux[(i+k) & wgBits];
  B4V(x,0);
  for (int k=0;k<4;k++) data[i+k] = x[k];
}
```

We now reach 131 Mkey/s for Key+Value, and 190 Mkey/s for Key only. Writing specific kernels for **inc=128,32,8** without the **inc** loop will probably lead to even more performance.

ParallelBitonic (updated with C* kernels)		
Kernels	**Key+Value**	**Key**
B2	49	73
B2+B4	78	115
B2+B4+B8	103	160
C2+B*	92	108
C4+B*	131	190

OpenCL Sorting

Eric Bainville - June 2011

Parallel merge, local

Here again, we sort in local memory first. We execute **N** threads in workgroups of **WG** threads, and each workgroup sorts a segment of **WG** input records. The output contains **N/WG** ordered sequences.

In merge sort, the main step of the algorithm is to merge together two sorted sequences. Suppose $a_0,a_1,\ldots,a_{p-1}$ and $b_0,b_1,\ldots,b_{p-1}$ are two sorted sequences. The position of a_i in the output is $i+f(b,a_i)$, where $f(seq,x)$ is the number of values in sequence **seq** that are less than **x**. Since **b** is sorted, this number can be obtained by dichotomic search in $\log_2(p)$ iterations. The same goes for the position of b_i in the output. This is implemented in the following kernel, each thread executes its own full dichotomic search in the sibling sequence:

```
__kernel void ParallelMerge_Local(__global const data_t * in,__global data_t * out,__local data_t * aux)
{
  int i = get_local_id(0); // index in workgroup
  int wg = get_local_size(0); // workgroup size = block size, power of 2

  // Move IN, OUT to block start
  int offset = get_group_id(0) * wg;
  in += offset; out += offset;

  // Load block in AUX[WG]
  aux[i] = in[i];
  barrier(CLK_LOCAL_MEM_FENCE); // make sure AUX is entirely up to date

  // Now we will merge sub-sequences of length 1,2,...,WG/2
  for (int length=1;length<wg;length<<=1)
  {
    data_t iData = aux[i];
    uint iKey = getKey(iData);
    int ii = i & (length-1);  // index in our sequence in 0..length-1
    int sibling = (i - ii) ^ length; // beginning of the sibling sequence
    int pos = 0;
    for (int inc=length;inc>0;inc>>=1) // increment for dichotomic search
    {
      int j = sibling+pos+inc-1;
      uint jKey = getKey(aux[j]);
      bool smaller = (jKey < iKey) || ( jKey == iKey && j < i );
      pos += (smaller)?inc:0;
      pos = min(pos,length);
    }
    int bits = 2*length-1; // mask for destination
    int dest = ((ii + pos) & bits) | (i & ~bits); // destination index in merged sequence
    barrier(CLK_LOCAL_MEM_FENCE);
    aux[dest] = iData;
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  // Write output
  out[i] = aux[i];
}
```

ParallelMerge_Local	
WG	**Top speed**
1	178
2	150
4	163
8	203
16	280
32	406
64	616
128	486
256	383

Performance of the ParallelMerge_Local kernel (Key+Value sort), Mkey/s.

The code is a little tricky to write correctly :-). Each thread performs **O(1)** global memory accesses, and $O(\log^2(WG))$ local memory accesses. We can observe this $O(\log^2(WG))$ behaviour for the largest values.