A PORTABLE IMPLEMENTATION OF THE RADIX SORT ALGORITHM IN OPENCL

P. HELLUY

Abstract. We present a portable OpenCL implementation of the radix sort algorithm. We test it on several GPUs or CPUs in order to assess its good performances on different hardware. We also apply our implementation to the Particle-In-Cell (PIC) sorting, which is useful in plasma physics simulations.

1. Introduction

Particle-In-Cell (PIC) methods are very popular for numerical simulations in plasma physics. The principle is to move a big number of particles, representing the charged particles, on a grid where the electromagnetic field is computed. The particles are accelerated by the electric and magnetic forces. The electric field and magnetic fields are modified by the charges and the currents created by the particles.

For computing the charges and the currents, it is necessary to add the contribution of each particle at the grid points. In order to be efficient, or if using parallel algorithms, it is important that the particles are sorted according to their cell indices [TS07]. The number of grid cells is often small compared to the size of the particles list.

The radix sort algorithm is an efficient algorithm for sorting a list of integer keys. It can be rather easily adapted to parallel multicore architectures [ZB91]. An interesting point of this algorithm is that it is made of several passes. If the keys in the list are small, i.e. if they can be represented by a small number b of bits, then the number of passes is also small and thus the algorithm is faster. It is thus a good candidate for sorting particles in PIC methods.

Several implementations of the radix sort algorithm exist for recent Graphic Processing Units (GPU) devices. Generally, they assume that $b = 32$. For instance, NVIDIA proposes a highly optimized radix sort implementation in its CUDA/OpenCL environment [nvocl]. But it is difficult to adapt to other integer sizes and/or architectures.

In this paper we thus propose an optimized implementation of the radix sort algorithm in the OpenCL language, which is both adapted to multicore CPUs and GPUs. Our implementation allows to choose the size of the integers b and the size of the radix r in order to find the most efficient compromise. It is tested on several hardware and in a realistic PIC two-dimensional configuration. The code is available at `http://code.google.com/p/ocl-radix-sort/`.

2. Radix sort algorithm

The radix sort algorithm is a classical stable sort algorithm adapted to integer lists. A parallel version is well described in [ZB91]. We first recall its philosophy.

We consider a list of integers $K(j)$ for $j = 0 \cdots N-1$, associated to a permutation $\sigma(j)$, which initially satisfies $\sigma(j) = j$. Each integer $K(j)$ is between 0 and $2^b - 1$ and will be called a key. The integer b is the total number of bits used for representing the keys. The algorithm returns the sorted list $K'(j)$ and the associated permutation

$$\begin{aligned} K'(j) &\leq K'(j+1), \quad j = 0 \cdots N-2, \\ K'(j) &= K(\sigma(j)), \quad j = 0 \cdots N-1. \end{aligned}$$

For sorting the list, we first write the keys in the representation of radix $R = 2^r$. The integer r is the number of bits necessary for representing the radix R. We suppose that b is divisible by r and we note $p = b/r$. The integer p will be the number of passes in the algorithm. Each key k can be represented in a unique manner

$$k = \sum_{m=0}^{p-1} k_m R^m, \quad 0 \leq k_m < R.$$

Key words and phrases. GPU, OpenCL, radix sorting, Particle-In-Cell.

It is also classical to use the notation (think about a number written with the usual decimal radix representation)

$$k = k_{p-1}k_{p-2}\cdots k_0.$$

Each pass $q = 0 \cdots p-1$ of the algorithm consists in sorting the list according to the q^{th} digit (in base R) $K(j)_q$. It is important that in each pass, the sort is stable: two already ordered keys must stay ordered. In this way the work done by the previous passes on the lower weight digits is not spoiled by the current pass.

2.1. **Histograming.** The first step of each pass consists in computing an histogram. We perform the parallel construction of the histogram in the following way.

We first consider a set of (virtual) processors. It will be natural to call them items, because in the OpenCL implementation they will be naturally associated to work-items. More precisely, we consider G groups of I items, which gives a total number of GI processors.

For simplicity, we suppose that N is divisible by GI, but if it is not the case, it is always possible to extend the list up to a multiple of GI. If it is the case, the list has to be extended by keys equal to the biggest possible integer $2^b - 1$, so that these keys stay at the end of the list.

Each item will be in charge of a part of the list. More precisely, each group $g = 0 \cdots G-1$ is in charge of the sub-list of elements with indices

$$j = g\frac{N}{G} \cdots (g+1)\frac{N}{G} - 1,$$

and in each group, the item $i = 0 \cdots I-1$ of the group $g = 0 \cdots G-1$ is in charge of the sub-list of elements with indices

$$j = g\frac{N}{G} + i\frac{N}{IG} \cdots g\frac{N}{G} + (i+1)\frac{N}{IG} - 1. \tag{2.1}$$

Each item $i = 0 \cdots I-1$ in a group $g = 0 \cdots G-1$ has to compute a part of the histogram, noted $H(c, g, i)$. The quantity $H(c, g, i)$ is the number of keys in the list $K(j)$ (j satisfies (2.1)) that are examined by the item i in the group g and that are equal to c. In practice it is stored in a one-dimensional array $H(cGI + gI + i) = H(c, g, i)$.

The histogram is simply computed by inspecting the list, computing the q^{th} digit c of the key $K(j)$ and then incrementing $H(c, g, i)$. This part can be done completely in parallel. No atomic operations are needed because each item computes its own histogram.

2.2. **Scanning.** In the second part of the pass q, a parallel prefix sum is performed on the array H, in such away that it is replaced by an array H' containing

$$H'(0) = 0, \quad H'(m) = \sum_{l=0}^{m-1} H(l).$$

The parallel prefix sum is a standard operation in parallel computing. It is also often called a "scan". A rather recent, efficient and ingenious algorithm for computing the scan in parallel is due to Blelloch [Ble91]. It is well described in [Gem3], with examples in the NVidia CUDA language. Other implementations of the scan algorithm can also be found in [Appscan], [nvocl], [amdocl].

2.3. **Reordering.** The main point is then the following: *after the prefix sum, the histogram $H(c, g, i)$ contains the position, in the ordered list, of the first key with radix c that is examined by the item i in the group g.*

The histogram can thus be used for reordering the keys. For this, each item i in a group g examines its part of the list (see (2.1)). If in the pass q the key $K(j)$ corresponds to the radix c, i.e. if

$$K(j)_q = c,$$

then the key is moved at the good place in the new list

$$K'(H(c, g, i)) = K(j).$$

At this point it is also possible to update the permutation

$$\sigma'(H(c, g, i)) = \sigma(j).$$

The quantity $H(c, g, i)$ is then incremented.

The reordering part of the algorithm can also be done completely in parallel.

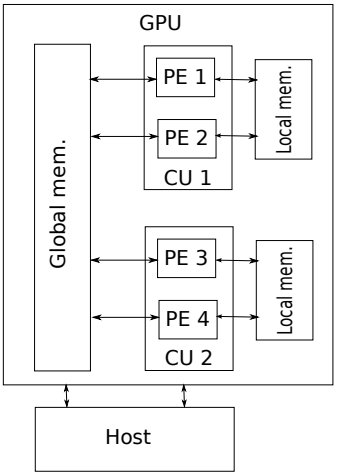

FIGURE 3.1. A (virtual) GPU with 2 Compute Units and 4 Processing Elements

3. OpenCL implementation

3.1. **OpenCL.** Today Graphic Processing Units (GPU) have more and more computation power. The Open Computing Language (OpenCL) is a software environment for simplifying the use of the GPUs for general computing. It is also possible to use OpenCL for driving a heterogeneous set of general multicore processors.

3.1.1. *Terminology.* We use the following terminology:

- Host: the computer into which the GPU is plugged.
- Kernels: the (generally small) programs that are executed on several of the computing cores of the GPU. For instance, the NVidia GPU GTX470 has 448 computing cores. Thanks to the OpenCL command queue management, it is possible to launch several millions kernel instances, which are dispatched on the hundred of cores of the GPU.

OpenCL means "Open Computing Language". It includes:

- A library of C functions (the OpenCL runtime), called from the host, in order to drive the GPU.
- A C-like language (the OpenCL language) for writing the kernels that will be executed on the computing cores.

It is practically available since september 2009. The specification is managed by the Khronos Group (that also drives the OpenGL specification) [Khro10].

3.1.2. *GPU.* Very schematically we can consider that a GPU is made of

- Global memory (typically 1.2 Gb for the recent NVIDIA GTX470);
- Compute units (typically 14).

Each compute unit is made of:

- Processing elements (typically 32), also called processors;
- Local memory (typically 48 kb) shared by the processors of the unit.

A schematic picture of a simple GPU is given in Figure 3.1

3.1.3. *Programation rules.* The same program can be executed on all the processing elements at the same time with the following rules:

- All the processing elements have access to the global memory;
- The processing elements have only access to the local memory of their compute unit;
- The access to the global memory is relatively slow while the access to the local memory is very fast.
- If possible, it is advised that the processing elements of the same compute unit access neighbour global memory locations, in order to improve "coalescence" (faster read/write access).
- The memory transfers between the host memory and the GPU are slow (as of may 2011) and should be avoided;

name	description	C++ name
N	size of the list	_N
I	number of items in a group	_ITEMS
G	number of groups	_GROUPS
b	size in bits of the integers	_TOTALBITS
r	size in bits of the radix	_BITS
R	radix	_RADIX
p	number of passes of a full sorting	_PASS
s	number of local scans before the global scan	_HISTOSPLIT

TABLE 1. Notation correspondance between this paper and the OpenCL implementation

- If several processing elements try a read-access at the same memory location (global or local) at the same time, all the reads will be successful;
- If several processing elements try a write-access at the same memory location (global or local) at the same time, ***only one write will be successful***. For some hardware, atomic operations maybe available, but should be avoided for performance reasons.

In order to perform a complex task, a kernel has to be executed many times.

- Each execution of a kernel is called a work-item.
- A work-group is a collection of work-items running on the processing elements of a given compute unit. They can access the local memory of their compute unit.
- Each work-item is identified by its local ID i and its work-group ID g.

For more details on OpenCL, we refer for instance to [Khro10].

3.2. **OpenCL implementation of the radix sort algorithm.** We have implemented the radix sort algorithm in a C++ class using OpenCL. The package can be downloaded at `http://code.google.com/p/ocl-radix-sort/`. It is made of a C++ class CLRadixSort and an example of use. We give in Table 1 the correspondence between the notations of this paper and the variable names in the implementation.

We give some indications of the implementation for the several part of the algorithm

3.2.1. *Histograming.* Because this part of the algorithm is fully parallel, it is rather easy to implement. However, it is important to organize the data in a special order in order to help coalesced access to global memory and to use local memory for better performance.

Transposition. The list K can also be considered as a matrix with IG rows and $\frac{N}{IG}$ columns stored in row-major order[1]. In this matrix, each row is examined by one work-item. Thus, two work-items in the same group access different rows of the matrix and the memory access are not optimal. Therefore, before sorting, we first transpose the matrix, which is equivalent to store it in the column-major order. We perform another transposition when all the passes are finished in order to recover the initial shape of the list. The transposition has to be organized carefully with the help of local memory, in order to help coalescence. See [RM09].

Local memory. The access to the histogram array can be arbitrary (because the list is probably initially not ordered !). For improving memory bandwidth in the histogram kernel we thus decide to store it partly in the local memory (cache) of the compute units. However it has to be done carefully: the size S in bytes of the local histogram constructed inside a group g on a single compute unit is equal to the number of radices R, times the number of items I inside a group, times the size in bytes of an integer (generally 4 for 32 bits integers)

$$S = 2^{r+2} I.$$

If C is the cache size of the compute units, it is necessary that

$$S < C.$$

For instance if the cache size $C = 16384$ bytes (NVidia GTX280) and if $b = 32$, it is possible to take $r = 8$ and $I = 8$ or $r = 4$ and $I = 128$. With a bigger cache $C = 49152$ bytes (NVidia GTX470) it is possible to take $r = 8$ and $I = 32$.

Smaller radix sizes r imply a higher number of passes (because $p = b/r$) while higher numbers of items I lead to a better occupation of the GPU. Thus, depending on the hardware, several combinations of r and I have to be tested in order to achieve the good compromise and the best performance.

[1] `http://en.wikipedia.org/wiki/Row-major_order`

name	system	memory	cache	compute units	processors
GeForce 320M	Mac OS X	256 MB	16 kB	6	48
GeForce GTX 470	Ubuntu 10.04	1.3 GB	48 kB	14	448
Radeon HD 5850	Ubuntu 10.04	512 MB	32 kB	18	1440
Phenom II X4 945	Ubuntu 10.04	4 GB		4	4
Phenom II X4 945 (one core)	Ubuntu 10.04	4 GB		1	1

TABLE 2. OpenCL devices for the comparisons

hardware	b	r	I	G	s
NV 320M	30	6	16	16	512
GTX 470	30	5	128	128	512
HD5850	30	6	32	128	512
Phenom II x4 (4)	30	6	1	4	8
Phenom II x4 (1)	30	6	1	1	8

TABLE 3. Quasi-optimal sorting parameters for several CPU and GPU

After that the local histogram has been computed in the cache of a group, it is copied to the global histogram H in the global memory. This copy is optimal, because coalesced.

3.2.2. *Scanning.* The scan algorithm is classical and practically described in [Gem3] or [Appscan] for instance. The histogram is first split into s parts that are scanned separately. The sums of each scanned part are then stored in an auxiliary array of size s. This array is also scanned. Finally, all the s separately scanned parts are updated, using the auxiliary array, in order to obtain the global scan of the histogram.

3.2.3. *Reordering.* This part is very similar to the histograming algorithm: first, the scanned histogram for group g is loaded into the local memory of the group. Then each item inspects its part of the list and, using its histogram, puts the corresponding keys at the right places in the new list. The same is done for the reordering permutation σ. The inspection of the list is bandwitdth-optimal because of the preliminary transposition (see Section 3.2.1). The access to the histogram is fast because it is loaded in the cache. The memory writes of the keys in the new list in global memory is the most expensive part of the algorithm because the access are randomly distributed.

3.2.4. *Possible improvements.* Here are some improvements that are not yet implemented:

- the index computation that allows to find the keys in the transposed array could be simplified;
- when the permutation is needed, it would be maybe more efficient to attach the keys and their permutation indices inside a pair of integers (OpenCL int2 type) for improving coalescence.

Here are some improvments that we do not plan to implement:

- modify the transpose and scan implementation in order to avoid memory bank conflicts. Indeed, these two parts of the algorithm represents a negligible part of the sorting time for big lists. The needed trick is also hardware dependent and makes the code less easy to read.
- modify the transpose implementation in order to avoid partition camping effects[RM09] because it is no more an issue on the most recent GPUs.

4. EXPERIMENTS

4.1. **Random lists sorting.** The first test consists in sorting a simple random list generated by the standard rand() function of <stdlib.h>. We consider the sizes $N = 2^{20} = 1\,048\,576 \simeq 1M$, $N = 2^{23} = 8\,388\,608 \simeq 8M$ and $N = 2^{25} = 33\,554\,432 \simeq 33M$. In this test, we do not compute the permutation, but only return the sorted list.

The different NVIDIA and AMD hardware are listed in Table 2. The current AMD implementation of OpenCL allows to select either the CPU or the GPU for the computations. When the CPU is selected, it is also possible to choose the number of activated CPU cores through the UNIX environment variable CPU_MAX_COMPUTE_UNITS. In the current AMD OpenCL implementation it is possible to access only half the memory of the Radeon HD 5850 (512 MB instead of 1 GB).

We have tried to find the optimal parameters for each hardware. They are summarized in Table 3.

name	transpose	histogram	scan	reorder	total	"speedup"
NV 320M	-	0.0610395	0.00307309	0.103567	0.167679	1.6
GTX 470	0.00123331	0.00162918	0.0026759	0.00497818	0.0105166	26
HD 5850	0.00247529	0.00672714	0.00445624	0.0206589	0.0343175	8
Phenom II x4 (4)	-	0.0136815	0.000936872	0.0227586	0.037377	7
Phenom II x4 (1)	-	0.0290106	0.00107121	0.0375346	0.0676164	4
STL C++ sort (optim.)	-	-	-	-	0.119109	2.3
STL C++ sort	-	-	-	-	0.27	1

TABLE 4. Sorting times (in seconds) for the 1M sequence

name	transpose	histogram	scan	reorder	total	"speedup"
NV 320M	-	0.265326	0.0352116	0.915375	1.21591	2
GTX 470	0.00882298	0.0113786	0.00267386	0.0459702	0.0688457	35
HD 5850	0.0216998	0.0291938	0.00266404	0.0734005	0.126958	19
Phenom II x4 (4)	-	0.0517777	0.000738896	0.098009	0.150526	16
Phenom II x4 (1)	-	0.15805	0.000490538	0.311058	0.469599	5
STL C++ sort (optim.)					1.08584	2.3
STL C++ sort					2.45	1

TABLE 5. Sorting times (in seconds) for the 8M sequence

name	transpose	histogram	scan	reorder	total	"speedup"
NV 320M	-	-	-	-	-	-
GTX 470	0.0350382	0.0442984	0.00267251	0.190379	0.272388	38
HD 5850	-	0.115329	0.00263945	0.287846	0.512948	21
Phenom II x4 (4)	-	0.180629	0.00080837	0.365266	0.546703	19
Phenom II x4 (1)	-	0.609637	0.000410401	1.31367	1.92372	5
STL C++ sort (optim.)	-	-	-	-	4.67	2.3
STL C++ sort	-	-	-	-	10.54	1

TABLE 6. Sorting times (in seconds) for the 33M sequence

We give the time spent in each part of the algorithm and the total sorting time. For comparison, we also indicate the sorting time obtained with the sort algorithm of the Standard Template Library (STL g++ version 4.2.1) on an Intel Core 2 Duo CPU at 2.13 GHz. The implemented sorting algorithm of the STL is the so-called introsort algorithm [Mu97]. We test the STL with or without compiler optimizations (-O option).

The results for the 1M, 8M and 33M sequences are given in Tables 4, 5, 6. The memory of the NV 320M GPU is not sufficient for the 33M list sorting. For the NV 320M GPU and the AMD CPU, we have also deactivated the transposition step because it has a bad effect on the efficiency.

As it is the tradition now in GPU publications, we have added a column that gives the "speedup" of the OpenCL implementation, i.e. the ratio of the not optimized STL sorting time over the OpenCL sorting time. Of course this number has to be considered with caution, because it highly depends on the CPU characteristics, compiler optimizations, size of the integers, shape of the sorted list, etc. It is only an indication of the performance of the parallel sorting algorithm. In addition, the STL sorting algorithm is a general template algorithm that can be applied to any type of data, which is not the case of our radix sort implementation, limited to list of integers $K(j)$ such that $0 \leq K(j) < 2^r$.

We observe that our implementation is rather efficient. Here we observe a speedup of magnitude 10, which is reasonably good.

GPU are not the best at sorting, and much higher speedup, of magnitude 100, are observed in other fields of scientific computing (for instance in signal processing or discontinuous Galerkin finite element methods). This drop of one order of magnitude is mainly due to the random memory access in the reordering step.

We also observe the very good performance of the OpenCL implementation on the AMD quad-core CPU. This indicates that OpenCL is also valuable for computations on hosts without GPU.

device	time $p = 6$	time $p = 2$	"speedup"
NV 320M	6.7084	1.42335	0.4
GTX 470	0.0679951	0.04502	11
HD5850	0.18143	0.109055	5
Phenom II x4 (4)	0.175059	0.0838514	6
Phenom II x4 (1)	0.49917	0.169612	3

TABLE 7. PIC sorting. Comparison of the algorithm for $p = 2$ or 6 passes for a list of 8M particles

4.2. **Particle-In-Cell (PIC) sorting.** In this section, we present another sorting problem that arises in the Particle-In-Cell (PIC) approximations. The PIC methods are classical in the numerical simulations of particle beams or in plasma physics. They are based on a big numbers of particles moving over a grid. Each particle belongs to a grid cell and is thus associated to a grid cell index. For efficiency reasons, it is important that the list is sorted according to the grid cell indices (see [TS07] and included references).

We consider in the periodic unit square $[0, 1[\times[0, 1[$ a grid of step $h = 1/32$. The cell with index $k = 32i + j$, $i = 0 \cdots 31$, $j = 0 \cdots 31$ corresponds to the little square $[ih, (i + 1)h[\times[jh, (j + 1)h[$. We observe that $0 \leq k < 32^2 = 2^{2r}$ with $r = 5$. Thus, it is possible to sort the particle list with only $p = 2$ passes of the radix sort algorithm with $R = 2^r = 32$. With integers of length $b = 30$ bits, 6 passes would be necessary. We wish to measure the gain of the reduced number of passes in a realistic simulation configuration.

For this, we construct a pseudo-random list of particles on the grid, with positions (x_j, y_j) and velocities (u_j, v_j) in the unit square. The pseudo-random numbers are generated thanks to four independent van der Corput sequences. We then perform a first sorting of the list. Then we move the particles according to the formula

$$\begin{aligned} x'_j &= x_j + u_j/32 \bmod 1, \\ y'_j &= y_j + v_j/32 \bmod 1. \end{aligned}$$

This choice ensures that the particles can either remain in the same cell or move to a neighboring cell. Thus the list is only "weakly disordered". We then perform another radix sort and measure the sorting time. The list is made of 8M particles. The results of two sorts are given, respectively for $p = 2$ or $p = 6$ passes, for several hardware, in Table 7. In this test, the computation of the reordering permutation is activated. Here, the "speedup" computation is based on the $p = 2$ device sorting time. The reference is the sorting time obtained with one core of the AMD CPU and $p = 6$ passes.

In this test again, we observe the good efficiency of our implementation. We also observe the good results obtained with the quad-core CPU. Our sorting algorithm could be used easily to improve standard CPU PIC solver.

5. Conclusion

We have developed and presented an efficient OpenCL implementation of the radix sort algorithm. Over other implementations that we have encountered on the web (as of may 2011), it has several advantages:

- it has been tested under several operating systems, hardware (CPU or GPU) and OpenCL libraries;
- it is possible to choose freely the sorting parameters: size of the integers, radix, etc.

It can still be improved on some minor points (bank conflicts, multiply and add operations but the performance are already very satisfying: we gain one order of magnitude over a classical single core introsort algorithm.

An interesting point is the very good behavior of the parallel radix sort algorithm on multicore standard CPUs. It indicates that it could very easily be applied to classical CPU PIC codes. Indeed, OpenCL allows the possibility to share a memory buffer between the host and the CPU device. This sharing is not efficient for a GPU because the data have to be transferred between the device and the host through the PCI express bus. This constraint will probably soon be released thanks to newly developed hardware (maybe the next generation of AMD Fusion processors that would share a CPU and a GPU with common memory).

References

[Khro10] Munshi, A. et al. The OpenCL specification. Version 1.1. Khronos OpenCL Working group. 2010.

[Ble91] Blelloch, G.E., Scans as Primitive Parallel Operations. IEEE Transactions on Computers, C-38(11):1526–1538, November 1989.

[Gem3] Nguyen, H. & al. GPU Gems 3. Addison-Wesley Professional (2007). See also http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html.

[Appscan] Apple developpers web site. OpenCL Parallel Prefix Sum Examplehttp://developer.apple.com/library/mac/#samplecode/OpenCL_Parallel_Prefix_Sum_Example/Introduction/Intro.html

[amdocl] AMD OpenCL SDK. http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx

[nvocl] NVidia CUDA/OpenCL SDK. http://developer.nvidia.com/

[Mu97] Musser, D. Introspective Sorting and Selection Algorithms, Software Practice and Experience vol 27, number 8, pages 983-993, 1997

[ZB91] Zagha, M. and Blelloch, G.. Radix sort for vector multiprocessors. Proc. of SuperComputing, 1991.

[TS07] D. Tskhakaya, R. Schneider, Optimization of PIC codes by improved memory management, Journal of Computational Physics, Volume 225, Issue 1, 1 July 2007, Pages 829-839

[RM09] Ruetsch, G. and Micikevicius P. Optimizing matrix transpose in CUDA. NVIDIA technical report. 2009. http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf

Université de Strasbourg, IRMA, France
E-mail address: helluy@math.unistra.fr
URL: http://www-irma.u-strasbg.fr/~helluy/