ISSN 1342-2812

CUDA Implementation of Iterative Updating: the Radix-2 Algorithm and Discrete Fourier Transforms

Mikael Onsjö, Kenta Kasai, Osamu Watanabe

Feb. 2010, C–268

CUDA Implementation of Iterative Updating: the Radix-2 Algorithm and Discrete Fourier Transforms

Mikael Onsjö*, Kenta Kasai†, and Osamu Watanabe*
*Dept. of Mathematical and Computing Sciences
†Dept. of Communications and Integrated Systems
Tokyo Institute of Technology, Tokyo, Japan

Research Report C-268, Dept. of Math. Comp. Sci.

Abstract

We consider the problem of computing $\boldsymbol{f}_m(\boldsymbol{f}_{m-1}(\cdots \boldsymbol{f}_1(\boldsymbol{x})\cdots))$ where each function $\boldsymbol{f}_i \colon R^n \to R^n$ can be broken up in pairs so that the computation at, e.g., indices k and l involve only the vales of the argument at positions k and l. That is, $\boldsymbol{f}_j(\boldsymbol{u}))_k \overset{\text{def}}{=} f_j^+(\boldsymbol{u}_k, \boldsymbol{u}_l)$ and so on. This generalizes "butterfly" algorithms, such as Radix-2 for computing Fourier transforms.

We demonstrate how to use a graphics Processing Unit (GPU), such as the Tesla C1060 with 240 cores, to perform a large number of executions of these algorithms efficiently in parallel. This has a general application among other things in the decoding of non-binary linear codes where the vectors are probability distributions over $GF(256)$. Interestingly, in this case it appears bank conflicts with shared memory cannot be completely avoided, yet may be reduced drastically by a nontrivial reordering of operations.

1 Introduction and Notation

The problem we consider can shortly be formulated as to compute

$$\boldsymbol{X} \leftarrow \boldsymbol{f}_m(\boldsymbol{f}_{m-1}(\cdots \boldsymbol{f}_1(\boldsymbol{x})\cdots))$$

where $\boldsymbol{f}_j \colon R^n \to R^n$, n even, and the functions restricted to the following form: Each $\boldsymbol{f}_j$ is associated with a list L_j of ordered pairs of numbers, $((k_{ij}, l_{ij}))_{i=0\ldots n/2}$, such that each number in $[0, 1, \ldots, n-1]$ occurs exactly once. For any (k_{ij}, l_{ij}) in the list, the k_{ij}:th elemnt of $\boldsymbol{f}_j(\boldsymbol{u})$ is defined as

$$(\boldsymbol{f}_j(\boldsymbol{u}))_{k_{ij}} \overset{\text{def}}{=} f_j^+(k_{ij}, l_{ij}) \overset{\text{def}}{=} f_j^+(\boldsymbol{u}_{k_{ij}}, \boldsymbol{u}_{l_{ij}})$$

and similarily the l_{ij}:th element as

$$(\boldsymbol{f}_j(\boldsymbol{u}))_{l_{ij}} \overset{\text{def}}{=} f_j^-(k_{ij}, l_{ij}) \overset{\text{def}}{=} f_j^-(\boldsymbol{u}_{k_{ij}}, \boldsymbol{u}_{l_{ij}}).$$

The functions f_j^+, f_j^- are generally considered simple in the sence that they can be implemented with a fixed, unbranching, list of such instructions as are available on relevant devices. Note that as each element is computed by the same function operating on different inputs, the system fits into the SIMD (Single Instruction Multiple Data) framework, which is an important point. From a mathematical viewpoint it is not neccessary that the functions be exactly identical, however we prefer to omit the index i nevertheless in order to emphasise the SIMD aspect. The tables of pairs is predetermined and given in advance.

If n and m are sufficiently large, e.g. 256 and 8 respectively, the problem may be well suited for GPU implementation, e.g. on the Tesla C1060 device. As is normal, we assume that several instances of the problem are solved independently and concurrently by different blocks of CUDA code. Hence we may focus the discussion on a single block solving only a small portion of the instances. A straightforward implementation may be organized as in Algorithm 1, where each thread is given responsibility for one position in the list of pairs, (k_{ij}, l_{ij}).

Our implementation of the radix-2 algorithm [JO06] follows this organization and is the driving motivation along with its application in the decoding of non-binary linear codes. We will explain this further in a later section. However, the main issue in all cases is the so called *bank conflicts* that may be incurred when reading X from shared memory.

The threads of a CUDA block of code are executed in half-warps of W at a time. With branching free code such as in Algorithm 1, all the instructions of a single half-warp may be carried out in parallel in the same time it would take to carry out a single instruction. Some issues can, however, arrise when data is read from the shared memory:

If the words in the shared memory are numbered $0, 1, \ldots,$ then memory position i is said to be connected to memory bank $(i \bmod W)$. The threads of a half-warp may read one word on each memory bank in parallel; this is the prefereable case. If two or more positions accessed by a half warp are on the same bank, we incur a so called bank conflict and the words must be read sequentially. If the largest number of words on a single bank is $c > 1$, we say that the half-warp incurs a c-way bank conflict and the time needed to execute the instruction is essentially multiplied by c.

The threads of a block may be organized into half-warp groups; thread t belongs to group $h = \lfloor t/W \rfloor$ and we write $H_h = \{Wh, Wh+1, \ldots, W(h+1)-1\}$. Threads of the same group are therefore always advanced in the same half-warps or conversely, threads of different half-warp groups are never advanced in parallel but piecewise sequentially. This suggests that we may be able to minimize bankconflicts by reordering the lists, L_j, of indices (k_{ij}, l_{ij}).

A different issue that should be addressed is that of memory latency. When moving data, such as (k_{ij}, l_{ij}), from global to shared memory there is a comparatively large delay, e.g. about 500 ticks per 128 bits [NV09]. Fortunately the GPU can mask this latency to some degree by switching the execution to another block when one is waiting for data. A CUDA kernel is said to be *memory bandwidth limited* if the time it takes to transfer data from global memory exceeds the time it takes to perform arithmetic

Algorithm 1 CUDA-style pseudo code for the implementation of an iterative updating algorithm as a GPU kernel. fp and fm represent the implementations of the functions f^+ and f^- respectively. The number of threads should be exactly $n/2$. The vectors x and out are input and output respectively, kept in global memory space. k, l are considered constant matrices also kept in global memory space.

```
  shared float[] X[n]
  shared int[][] K[n][m]
  shared int[][] L[n][m]
  const int i = threadIdx.x
  float x1, x2

  for j = 1 to m do
    K[i][j] = k[i][j]
    L[i][j] = l[i][j]
  end for

  for job = 1 to MAXJOBS do
    X[i] = x[job][i]
    X[i + n/2] = x[job][i + n/2]
    for int j = 1 to m do
barrier
      x1 = X[K[i][j]]
      x2 = X[L[i][j]]
      X[K[i][j]] = fp( i, j, x1, x2 )
      X[L[i][j]] = fm( i, j, x1, x2 )
    end for
barrier
    out[job][i] = X[i]
    out[job][i + n/2] = X[i + n/2]
  end for
```

instructions; this is usually undesirable. As will be explained near the end of the next section, Algorithm 1 is organized to avoid memory bandwidth limitations and hence its performance may be described adequately by estimating the time for arithmetic instructions and bank conflicts with the shared memory.

2 Analysis and Optimization of General Algorithm

We first consider the method outlined in Algorithm 1. Consider the threads of a single half-warp group H_h. Note that every instruction carried out by this group is done in constant time wrt. W, with the possible exception of the lines of the innermost loop. In relation to these lines, bank conflicts may occur during the reading and writing of

X[k[i][j]] and X[l[i][j]] from/to shared memory. Let T_{hj} denote the actual time it takes the half warp to carry out the instructions at iteration j of the loop; we may write

$$\begin{aligned} T_{hj} = \Theta(& \max_{r=0\ldots W-1} |\{i \in H_h \colon k_{ij} \bmod W = r\}| \quad + \\ & \max_{r=0\ldots W-1} |\{i \in H_h \colon l_{ij} \bmod W = r\}|). \end{aligned}$$

Proposition 1 follows.

Proposition 1. *The time it takes to perform Algorithm 1 is bounded by:*

$$c \sum_{h=0}^{\lceil n/W \rceil} \sum_{j=0}^{m} T_{hj}$$

where c is some constant independent of n, m and W.

Note that in the best case, without bank conflicts, this time is $\Theta(nm/W)$ while in the worst case it could be as much as $\Theta(nm)$. The latter case would of course completely negate most advantages of using GPU parallelism. If on the other hand we may assume that the pairings of indices in each list and the list's order, are choosen uniformly at random, the situation resembles the well studied problem refered to as *balls-into-bins.* Proposition 2 follows:

Proposition 2. *Suppose the indices k_{ij}, l_{ij} are choosen uniformely at random and $n \gg W$. Then there is a constant c and an exponentially decreasing function ϵ such that Algorithm 1 terminates in time less than*

$$c \frac{nm \log W}{W \log \log W}$$

with probability $> 1 - \epsilon(W)$.

Proof. Choose a half-warp group, H_h, uniformly at random and the indices k_{ij} for some fixed j and all $i \in H_h$. If $n \gg W$, the indices, k_{ij} may be considered as drawn independently at random from R^n. The probability of k_{ij} being on any specific memory bank is exactly $1/W$. The largest number of indices on a single bank is then known [MU05] to be $(\log W)/\log \log W$ on average and sharply concentrated as in the statement of the proposition. The same is true for indices l_{ij}. □

If for instance $W = 16$, as on the Tesla C1060 GPU, then $(\log W)/\log \log W \approx 2.72$. So while the worst case takes 16 times longer than the best case, the average only takes about 2.7 times longer than the same.

We note that there is some arbitrariness in the ordering of each list, L_j; reordering the list may affect the time of the algorithm but not the final result. We therefore propose the greedy reorganization Algorithm 2. Note that this algorithm is assumed to be executed only once prior to a large number of executions of Algorithm 1.

Algorithm 2 Greedy algorithm for reducing bank conflicts by rearranging the list of index pairs. The function SumBankConflicts simply returns $\sum_{h=0}^{\lceil n/W \rceil} T_{hj}$ for the rearranged list it is given. swap swaps two elements at given indices in a list.

```
Input: List of pairs: $L(i)$, $i = 0 \dots n/2 - 1$
Output: List of pairs: $LL(i)$, $i = 0 \dots n/2 - 1$
  $LL \leftarrow L$
  for $it = 0$ to MAXIT do
    $it \leftarrow it + 1$
    $oscore \leftarrow$ SumBankConflicts($LL$)
    for $ti = 0$ to $n/2 - 1$, $tj = 0$ to $n/2 - 1$ do
      $score \leftarrow \infty$
      $TLL \leftarrow$ swap($LL, ti, tj$)
      $tscore \leftarrow$ SumBankConflicts($TLL$)
      if $tscore < score$ then
        $(score, i, j) \leftarrow (tscore, ti, tj)$
      end if
    end for
    if $score < oscore$ then
      $LL \leftarrow$ swap($LL, i, j$)
    else
      return $LL$
    end if
  end for
  return $LL$
```

It should be noted that Algorithm 1 can be considered limited by arithmetic instructions rather than memory bandwidth, provided that the number of jobs assigned to each block, MAXJOBS, is sufficiently large. This is indeed the purpose of the loop on line 11 and the pre-fetching of k and l on lines 7 to 8. If the implementations of f^+ and f^- require a large number of instructions or there are many (unavoidable) bank conflicts on line 17 to 21, the pre-fetching and job loop may be altogether unneccessary.

3 The Discrete Fourier Transform and the Radix-2 Algorithm

The discrete fourier transform of a sequence $x(n)$, $n = [0, \dots, (N-1)]$, is commonly defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}$$

where $W_N = \mathrm{e}^{-2\pi j/N}$ and j stands for the imaginary unit. If for some integer v, $N = 2^v$, a power of two, then the summation can conveniently and recursively be split into even and odd numbered cases by the observation that $W_N^{2nk} = W_{\frac{N}{2}}^{nk}$ and $W_N^{(2n+1)k} = W_N^k W_{\frac{N}{2}}^{nk}$:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_{\frac{N}{2}}^{kn} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_{\frac{N}{2}}^{kn}.$$

The two new sums are again fourier transforms of the decimated series over even and odd numbered indices respectively. This divide-and-conquer approach leads to what is called the *Decimation-in-time Radix-2* algorithm [JO06], which is a special case of the Cooley Tukey algorithm [CT65].

Implementation can be done *in-place* with sequences of so called *butterfly operations.* Define

$$\begin{aligned} f^+(x_0, x_1; W) &= x_0 + W x_1 \\ f^-(x_0, x_1; W) &= x_0 - W x_1 \end{aligned}$$

and proceed to carry out the computation in $\log_2 N$ iterations. The first ($j = 1$) iteration calculates the $N/2$ two-point fft's as:

$$\begin{aligned} x(k) &\leftarrow f^+(x_k, x_{k+N/2}; 1) \\ x(k + N/2) &\leftarrow f^-(x_k, x_{k+N/2}; 1). \end{aligned}$$

for $k = [0, 1, \ldots, (N/2 - 1)]$. The next iteration will start by combining the two-point fft's on index pairs $(0, N/2)$ and $(N/4, N/4 + N/2)$ respectively into a four-point fft. This operation can again be expressed in terms of butterfly operations, f^+ and f^- carried out in pairs:

$$\begin{aligned} x(0) &\leftarrow f^+(x_0, x_{N/4}; 1) \\ x(N/4) &\leftarrow f^-(x_0, x_{N/4}; 1) \\ x(N/2) &\leftarrow f^+(x_{N/2}, x_{N/2+N/4}; W_{2^2}^1) \\ x(N/2 + N/4) &\leftarrow f^-(x_{N/2}, x_{N/2+N/4}; W_{2^2}^1). \end{aligned}$$

The ordering here is well defined, and it is not difficult to program the calculation of remaining index pairs or the constant W, in the second or subsequent iterations. However, as normal mathematical notation becomes somewhat cluttered, we will omit to specify this explicitly.

It is important to note that if the original vector is stored in natural order, $(0, 1, \ldots, N-1)$, the tranformed result will be stored (in the same place) in *bit-reversed* order, $(0, N/2, N/4, N/2 + N/4, \ldots)$. Precisely speaking, this form is obtained by reversing the order of the bits of an index written in binary form with exactly $\log N$ digits (having preceding zeros if neccessary). Conversely of course, if the original vector is already stored in bit-reversed order, the final result will be in natural order.

The approach we have outlined now essentially follows the recipe specified by Algorithm 1. The only thing that needs to be addressed, briefly, is the introduction of the parameter W into the functions f^+ and f^-. This parameter can, however, be computed a priori and provided along with the index pairs, k_{ij}, l_{ij}, and hence does not change the situation signifficantly.

There are two points that motivate this implementation on the GPU with precomputed indices and parameter:

1. If a large number of vectors of the same size need to be fourier transformed, precomputation can save time even on sequential computing devices. This may be slightly more important on the GPU as the support for integer operations (e.g. modulos) is usually comparatively worse relative to the support for floating point operations.

2. The order that elements of x are stored in may differ between applications and over time. It may be convenient to maintain only one kernel implementation that is provided different precomputed lists for different cases.

On the other hand it should be noted that the fast fourier transform is an intensly studied method. There are very fast and more general implementations for GPU and the implementation suggested here is not particularly novel per ce. We believe, however, that it is one of the fastest and simplest for cases $N = 2^v$; the purpose of this section is to show that it fits in the general framework of Algorithm 1 and that it has the virtue of being flexible under reorganization of the input vector.

As was mentioned early, bank-conflicts may occur in Algorithm 1; this depends on which order the butterflies of each iteration are listed in. However, we may note that a butterfly operation specified by index pair (k, l) and constant W is equivalent to the reversed operation (l, k) with constant $-W$. This opens up another level of possibilities which is not captured by the greedy optimization of Algorithm 2, but can of course be added easily. However, when the input, $\boldsymbol{x}$, is in natural order it is not difficullt to see that there is an ordering of the butterflies that eliminates bank conflicts altogether, if we allow the reversal of some butterfly operations.

4 The Fourier Transform on GF(256) and its Application to the Decoding of Non-binary LDPC Codes

As is explained in [DK98], the decoding of non-binary LDPC codes involve probability distributions over GF(2^v), for some positive integer v, where for the purpose of this text we will assume that $2^v = 256$. It is convenient to work with these probability distributions in the frequency domain as decoding algorithms commonly require certain convolutions to be computed, and convolutions in the frequency domain become mere elementwise multiplications.

Commonly the decoding is done with iterative schemes; in each iteration a large number of different 256-element vectors representing different probability distributions on GF(256) need to be fourier transformed. As is described in [RU08], the Fourier transform may for this purpose be defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}$$

where $N = 256$ and $W_N = \mathrm{e}^{-\pi j}$.

Since the Twiddle factors, W_N^k, are now always 1 or -1, they may essentially be ignored in the implementation of the Radix-2 Decimation-in-time algorithm following the structure of Algorithm 1 by taking W to be 1 in every butterfly. Hence we may simply define

$$f_j^+(k_{ij}, l_{ij}) \overset{\text{def}}{=} \boldsymbol{x}_{k_{ij}} + \boldsymbol{x}_{l_{ij}}$$
$$f_j^-(k_{ij}, l_{ij}) \overset{\text{def}}{=} \boldsymbol{x}_{k_{ij}} - \boldsymbol{x}_{l_{ij}}$$

and assume that each precomputed pair (k_{ij}, l_{ij}) is ordered appropriately. Unfortunately this means that the order of each butterfly becomes fixed (as opposed to the previous section where they could be reversed), and we may not be able to avoid bank conflicts completely. Fortunately the benefit of avoiding multiplications outweight this concern by far in practice.

It is sometimes convenient to arrange the elements of a field with characteristic 2 according to binary representation. If for example we choose the irreducible polynomial $x^3 + x + 1$ as basis for polynomial representation, we may write

$$(0, 1, x, x^2, x^3, x^4, x^5, x^6) = (0, 1, x, x^2, x + 1, x^2 + x, x^2 + x + 1, x^2 + 1)$$

which in binomial form can be represented as

$$(000, 001, 010, 100, 011, 110, 111, 101)_2 = (0, 1, 2, 4, 3, 6, 7, 5).$$

If the elements of a vector $\boldsymbol{x}$ is changed to this ordering, the index pairs, (k_{ij}, l_{ij}), for the fft must be changed accordingly. It is no longer obvious how the lists, L_j, should be ordered to minimize the number of bank conflicts, and we find it convenient to apply the greedy Algorithm 2.

Notes: As $\boldsymbol{f}$ is in this case a relatively simple function and the number of bank conflicts quite limited, it seems a number of eg. five or more jobs per block (MAXJOBS) is preferable. Further, $n = 256$ means $m = 8$; if these parameters are fixed at compile time, the loop on line 15 should be *unrolled* for a small but not insignificant time improvement.

As Table 1 demonstrates there is a preferable blocksize around 256 for the numbers detailed. In this case, the random version takes about 20% more time than the optimized. Similarily, the "nounroll" version takes about 5% longer than the optimized.

Acknowledgement

This research was supported in part by the JSPS Global COE program "Computationism as a Foundation for the Sciences".

blocksize	random	optimized	nounroll
64	1.227	1.017	1.029
128	1.044	0.868	0.892
256	0.968	0.807	0.846
512	0.989	0.833	0.915
1024	1.034	0.883	1.043

Table 1: Experiment results for Fourier transforming 4096 vectors of size 256 using a C1060 Tesla GPU. "Blocksize" refers to the CUDA term and is constrained by MAXJOBS*blocksize=4096. The other columns contain run times in seconds. "Random" means the order of L_j is random contrary to "optimized", in which case Algorithm 2 has been applied. "nounroll" is the optimized version but the inner loop in the CUDA kernel was not unrolled at compile time. Each time entry is the average of 100 tries.

References

[CT65] Cooley, J. W., Tukey, J. W.: An algorithm for the machine calculation of complex fourier series. Mathematics of Computation 19 (90), pp. 297-301, (1965).

[DK98] Davey M.C., MacKay, D.: Low-Density Parity-Check Codes over GF(q). IEEE Commun. Letters, vol. 2, pp. 165-167, June, (1998).

[JO06] Jones, D.: Decimation-in-time (DIT) Radix-2 FFT. Connexions Web site, http://cnx.org/content/m12016/1.7/, Sep 15, (2006).

[MU05] Mitzenmacher, M., Upfal, E.: Probability and Computing, Randomized Algorithms and Probabilistic Analysis. Cambridge Univ. Press, (2005).

[NV09] Nvidia: CUDA Programming Guide, ver. 2.3.1. http://www.nvidia.com/object/cuda_develop.html, Aug 26 (2009).

[RU08] Richardson, T., Urbanke, R.: Modern Coding Theory. Cambridge University Press, (2008).