ISSN 1342-2812

Research Reports on Mathematical and Computing Sciences

Faster Evaluationof
ZBDD Compressed Multi-Linear Functions
with GPU Parallelism

Shin-ichi Minato, Mikael Onsjö, Osamu Watanabe

Jan. 2011, C–274

Department of
Mathematical and
Computing Sciences
Tokyo Institute of Technology

SERIES C: Computer Science

Faster Evaluation of ZBDD Compressed Multi-Linear Functions with GPU Parallelism

Shin-ichi Minato[†], Mikael Onsjö[*] and Osamu Watanabe[*]
[†]Graduate School of Information Science and Technology, Hokkaido University
[*]Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology
(watanabe(at)is.titech.ac.jp)

(Research Reort C-274, Dept. of Math. & Comp. Sci., January, 2011)

1 Background

This report describes a method to quickly evaluate multi linear functions (MLFs for short) using GPUs. The goal is to evaluate MLFs expressing marginal probabilities in large Bayesian Networks that are represented as compressed to zero-suppressed binary decision diagrams (ZBDDs for short) [Min93]. These functions have highly optimized CPU implementations; we compare them with their corresponding GPU implementations.

The method for creating the GPU implementations is not limited to the case of MLFs but could theoretically be applied to virtually any computation that can be factored into a (large) set of fairly homogeneous operations. Bayesian Networks, however, do due to their importance in a vast number of different applications provide a good setting for comparison.

In recent years, researchers have for various algorithms reported speedups in the order of 100, 200 or occasionally even several thousand times when moving from CPU computation to the GPU. In many cases it is, however, somewhat questionable whether the sequential, CPU, implementation used for comparison was fully optimized. In the case of MLFs represented by ZBDDs, however, significant effort has been made [MSS07] to provide a highly optimized single thread implementation.

An approach taken by Minato et. al. recently [TM10] is to generate a C program for the function and then have the GNU compiler, gcc, optimize the computation. At the present we do not know of any software that can provide the same service for compiling general code to the GPU. Instead we try to organize the data in parallel levels of computation and make an implementation in CUDA. Additionally since gcc (cc) requires huge amounts of time and memory, we were compelled to make a faster but efficient compiler for the CPU.

Our results indicate that for some large Bayesian Networks, publicly available for benchmarking purposes (see, e.g., [BNR]), an implementation on the Tesla C1060 GPU

runs at least 6 times faster than what we may expect from an optimized single thread implementation on a 2.5 GHz AMD Phenom processor.

2 Transforming a ZBDD Compressed MLF into a GPU program

In a ZBDD compressed MLF [MSS07], the computation is simply represented as a set of operations of one of the forms

$$f[a] \leftarrow f[b]x[c] + f[d] \qquad \textbf{typeA}$$
$$f[a] \leftarrow f[b]x[c] \qquad \textbf{typeB}$$

where f is the values kept in the nodes of the diagram, x the input values given as an argument to the MLF and a, b, c, d are varying indices to f and x respectively. Most of the values in f are originally uninitialized (if there is an operation like $f[a] = x[b]$ or $f[a] = 1$, we simply call $f[a]$ initialized).

For the C-program, each variable $f[i]$ is managed by the compiler which will try and optimize cache and memory usage (i.e. in the code $f[123]$ is written as $f123$) and the numbering is inconsequential. By contrast, in the GPU program the variables must by necessity be kept as an array in global memory and should be numbered compactly. Furthermore to enable coalesced writing, the variables are named so that the left side is in sequence, and this ordering is maintained whenever the operations are reordered. For example, a sub-sequence of the operations may be:

$$\begin{aligned} f[4] &\leftarrow f[1]x[2] + f[2] \\ f[5] &\leftarrow f[3]x[3] \\ f[6] &\leftarrow f[4]x[4] + f[5] \end{aligned}$$

which would mean that $f[6] = f[1]x[2]x[4] + f[2]x[4] + f[3]x[3]$ at the end.

Note that in this example, the ordering of computation for $f[4]$ and $f[5]$ is arbitrary - they may be computed in either order or in parallel as long as both are complete before the $f[6]$ calculation is begun. The three operations may be carried out in two respectively parallel levels. The second step in creating the GPU program is exactly this: Split the computation into a number of levels so that the operations of each can be computed in parallel. The necessary number of levels correspond to the longest path in the ZBDD. To begin with we choose the levels "greedily", that is, starting with the initialized variables as level 0, level 1 will be all variables that can be computed from the initialized ones and so on. In the worst case one might need as many levels as there are operations, but in practice and for the Bayesian Networks we are dealing with the number is significantly lower. Figure 1 show examples of how the levels may be chosen for some networks.

Each scalar multiprocessor (SM) on the GPU performs SIMD (single instruction multiple data) operations in warps of $W = 32$ threads at a time. The group of threads executed by one SM is called a block. To avoid many conditional jumps that take time

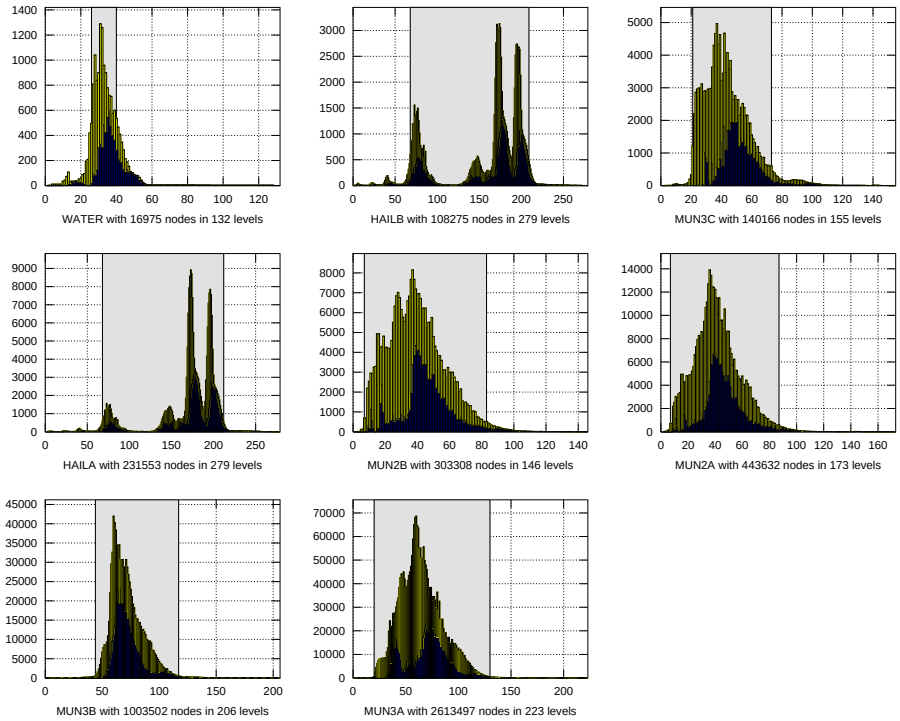

Figure 1: Histogram over the size of the levels obtained after analyzing the ZBDDs for various Bayesian Network MLFs. The gray area indicate levels that are performed by the GPU, e.g. the thresholds where the level size passes 480 operations. The dark area corresponds to typeA operations while the lighter colored parts of each bar represent typeB operations.

and may make the threads of a warp diverge, we organize the operations of each level so that typeA are first and typeB second. Then we divide the level of operations into a fixed number of blocks that we call $B = B_A + B_B$, each block is assigned a roughly equal number of consecutive operations from either the typeA (B_A blocks) or the typeB group (B_B blocks). No block performs operation of both types. Each block is set to execute a number, T, of threads, each of which performs a roughly equal number of operations that is now determined by the parameters and the size of the level.

A level consisting of few operations is not well suited for computation on the GPU. Certainly the number of operations should be more than the number of cores available. We therefore choose a threshold, T_h, relative to the number of cores (240 on the Tesla C1060): smaller levels are performed by the CPU and larger by the GPU. This works well in practice in many cases, but unfortunately when switching between CPU levels and GPU levels, data must be transferred between the two devices (e.g. over a PCIe

interface). It is therefore better to choose somewhat softer threshold criteria. Many networks may be divided into three phases; CPU preliminary, GPU and CPU posterior; as the level diagrams are roughly uni-modal. Networks with highly multimodal level diagrams (such as HAILA and HAILB in figure 1) on the other hand, are more complicated.

For performing the computation of GPU levels, the best performance will be had e.g. if both the the number of typeA and the number of typeB operations is fairly large and divisible by $B \cdot W$. For one level of operations it may be possible to postpone some operations to the next level, provided that the result of those operations is not needed in that level. The last step in organizing the operations is therefore to try and justify the sizes of the levels in this fashion. The task of optimizing in this way appears to be computationally hard in general, but a simple greedy approach performs relatively well in practice.

So far we have not made use of the software controlled shared cache that is available on each SM on the GPU. Doing so could possibly speed up the computation whenever one variable is used several times within the same level. In practice this seems to happen for the input variables, x, but less often for f. We therefore proceed as follows: For each type of variables in each level, sort the operations w.r.t. the index of the input operand x that is used. Take note of the largest and smallest index. Let M denote the number of words that can be contained in the shared cache; now split the input variables naturally into sub-groups of sizes M and the groups of operations correspondingly into sub-groups of operations. Whenever a scalar multiprocessor starts working on such a sub-group, it first preloads all the pertinent input variables into shared cache. Note that the indices of the input variables must be recomputed to indices in the cache, this can be done once during the compilation of the ZBDD into the GPU program.

3 Custom CPU ZBDD Compiler

Since the GCC C compiler requires too much time and memory for use with large networks, we had to make a custom compiler. This generates assembler code in linear time, then compiles and links it using GCC (GAS). As can be seen in the next section, this approach compiles significantly faster and yields results that run almost as fast as the optimized gcc versions. Furthermore the memory requirement in the latter case is linear and quite manageable for the networks used so far.

In this approach we select the order of the operations according to a depth first search where, if there is more than one child, we prefer to start with the child that has the largest number of parents. This ensures that almost all variables that have only one parent (is used only once after it is computed) never have to be stored in memory. As can be seen in figure 2, this is one of the more significant of optimizations.

Note: This indicates that the optimization technique applied by gcc that has most bearing on the result, is the reordering of computation so that some variables are not stored in main memory at all.

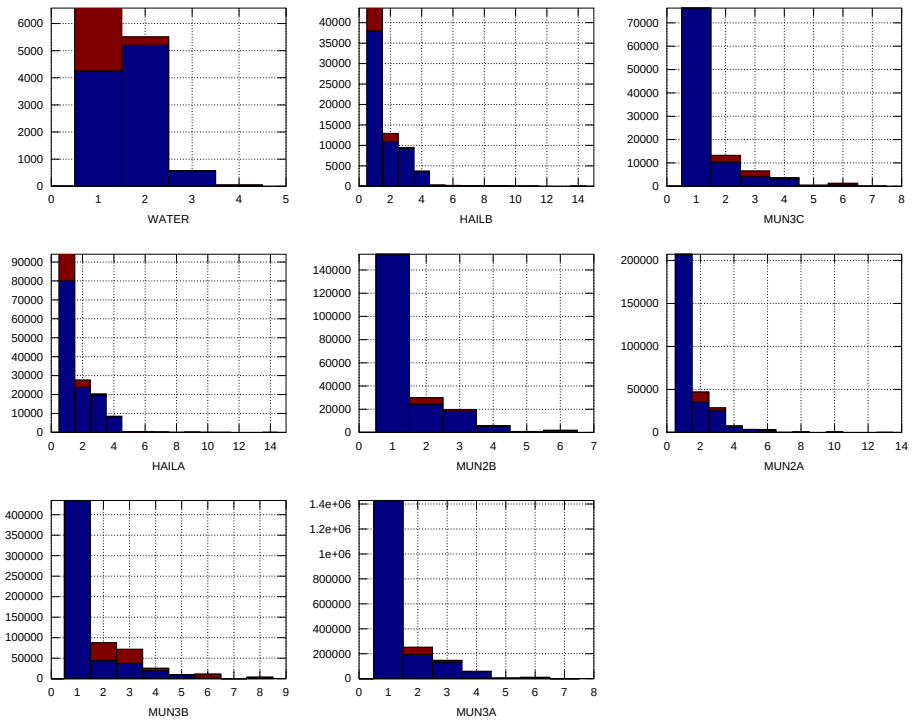

Figure 2: This shows for each network how many nodes there are that have a certain number of parents. In any of the larger networks, only about 20% of all nodes have more than one parent.

4 Presentation of Results

The results indicate that for some large networks a Tesla C1060 GPU implementation may be able to evaluate the multi-linear functions at least 6 times faster than an optimized single thread implementation with a 2.5 GHz AMD Phenom CPU, or 16 times faster than the unoptimized gcc version. The data is presented in figure 3.

When gcc is used without optimization options, the times increase regularly in a near linear fashion. For the GPU, the times appear to level out somewhat. Comparing the expected derivative[1] of the curves for large networks yield a factor of 16 or higher. When gcc is used with optimization O1, it becomes impossible to compile large networks in reasonable time (see table 1). Instead we provide a custom version that appears to be nearly as efficient as the gcc optimized one but with significantly less overhead to

[1] We compare the slant of the last line segments for the two curves.

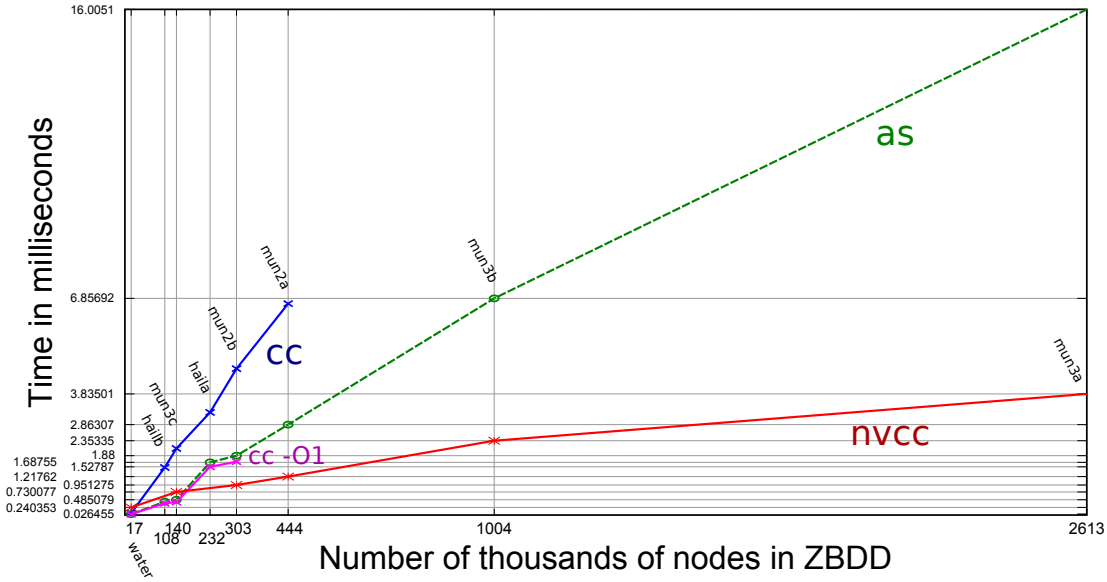

Figure 3: The time needed to evaluate some MLFs for various Bayesian Networks, for the Tesla C1060 GPU and the 2.5 GHz AMD Phenom CPU respectively. Note that a suitable GPU implementation for highly multi-modal case such as HAILA and HAILB (see figure 1) has not yet been made.

compile. When again the expected asymptotic behaviours[2] are compared, a factor of at least 6 is suggested for the GPU speedup.

5 Further Improvements on the GPU Side

While the single thread CPU implementation would appear to be nearly (relatively speaking) as optimized as it could be, there are a number of roads open for improving on the GPU implementation. In this section we will outline a preliminary study of one class of such possibilities.

The idea is based on the observation from figure 2 that most (i.e. 80-90%) of all operation have only one parent (the result is used only once). Furthermore it is apparent that there is a lot of overhead in the GPU kernel when a thread after performing a simple typeA or typeB operation must immediately perform several loop instructions (increment, condition and jump). This flaw could be ameliorated to a certain degree by performing more complicated operations than the simple typeA and typeB.

Such operations can be formed by merging subsets of the original operations. Any operation with only one parent can obviously be removed from whatever level it was originally in and be computed instead as a direct part of the parent operation. Unfortu-

[2] Again we compare the last line segments.

Network	ZBDD Nodes	nvcc	gcc	gcc -O1	custom (as)
WATER	17,000	5	3	78	0.16
HAILB	108,275	*	31	283	0.85
MUN3C	140,000	15	45	454	1.2
HAILA	231,553	*	99	1557	1.8
MUN2B	303,000	32	171	2146	2.2
MUN2A	444,000	46	366	*	3.3
MUN3B	1,004,000	108	*	*	8.4
MUN3A	2,613,000	314	*	*	18.6

*no data obtained

Table 1: Build (compile) times in seconds for GPU and CPU versions. The CPU versions were built using the GNU C compiler version 4.5 and the GNU assembler (GAS) version 2.20. The GPU version was built using the NVIDIA CUDA compiler version 3.00. "GCC -O1" indicates that the optimization flag O1 was passed to gcc.

nately this gives rise to a combinatorial explosion in the number of possible "compound" operation. At the moment we will therefore restrict our attention on the case where typeB operations are merged with parents into one of the following six types:

$$f[a] \leftarrow f[b]x[c] \qquad \textbf{type1}$$
$$f[a] \leftarrow f[b]x[c]x[d] \qquad \textbf{type2}$$
$$f[a] \leftarrow f[b]x[c] + f[d] \qquad \textbf{type3}$$
$$f[a] \leftarrow f[b]x[c]x[d] + f[e] \qquad \textbf{type4}$$
$$f[a] \leftarrow f[b]x[c] + f[d]x[e] \qquad \textbf{type5}$$
$$f[a] \leftarrow f[b]x[c]x[d] + f[e]x[g] \qquad \textbf{type6}$$

(mergings that would produce more complicated types are not allowed).

Looking again at figure 2 it is apparent that typeB operations with single parents far outnumber all other operations and so we can expect to be able to reduce the total number of operations by roughly 50% by organizing things carefully. With the type1-type6 operations the GPU still has a large overhead per operation. But while this is in a sense inefficient, it also means that reducing the total number of (simple) operations (in favor of the slightly more complicated operations) can be expected to boost performance almost proportionally. While the total amount of essential computation is unchanged, the total amount of overhead is reduced by 50%.

Extending the idea further we ask weather it is possible to combine operations even further. As mentioned before, the problem is a combinatorial explosion of the number of possible types. Each kernel block of e.g. 32 or 64 threads should execute operations of the same type, so when the number of represented types surpass, say, 1/32:th the size of the level we waste resources because of idling threads. Although the situation is highly problem dependent, it seems reasonable to hypothesize that the total number

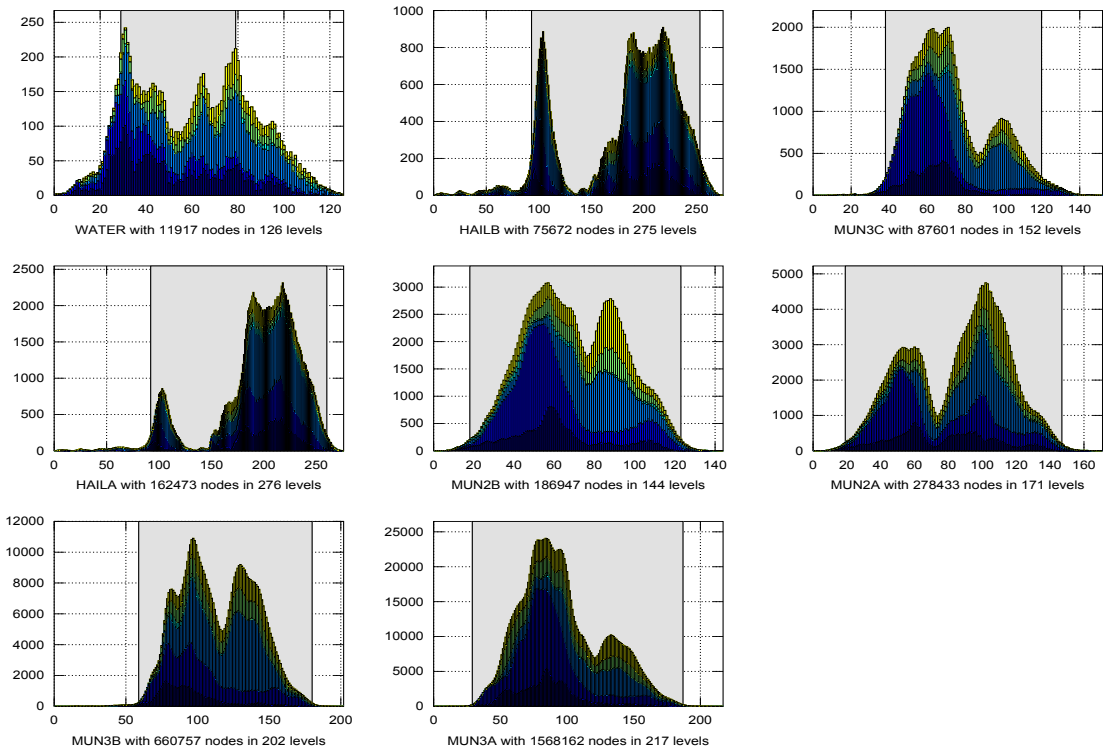

Figure 4: This shows for each network how many operations of each of the compound type1-6 operations there are organized into levels (in one possible way).

of operations can be reduced by a factor proportional to $\log \bar{L}$, where $\bar{L}$ is the average number of original typeA and typeB operations in a level, by recursively combining a number of single parent operations together to that depth.

At the same time there are many different ways of combining operations and it may very well be possible to avoid using several "types" altogether, or alternate types between different levels. Other approaches exist: It may be possible to reduce the total number of levels by, say, targeting one and merging all it's operations with parents. It may be possible to reduce the number of types recognized by the GPU kernel, by executing some simpler types as if they were more complicated, using dummy variables of 1 or 0.

In short, organizing the GPU computation appears to be a highly non-trivial optimization problem in its own right. Although this problem certainly will depend on the shape of the underlying ZBDD/network, it seems likely that some interesting, generally valid, statements may also be withing reach.

Acknowledgement

This research has been conducted under the collaboration of the JSPS Global COE program "Computationism as a Foundation for the Sciences" and the ERATO Minato Discrete Structure Manipulation Project.

References

[BNR] Bayesian Network Repository, `http://www.cs.huji.ac.il/ galel/Repository/`

[Min93] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems, in *Proc. of 30th ACM/IEEE Design Automation Conference* (DAC'93), pp. 272-277, Jun. 1993.

[MSS07] S. Minato, K. Satoh, and T. Sato, Compiling Bayesian networks by symbolic probability calculation based on Zero-suppressed BDDs, in *Proc. of 20th International Joint Conference of Artificial Intelligence* (IJCAI-2007), pp. 2550-2555. Jan. 2007.

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

[TM10] W. Takahashi and S. Minato, Synthesis of fast calculation programs from ZDDs for representing Bayesian networks and its evaluation, in *Proc. IEICE/IPSJ Forum on Information technology* 2010, D-009, Vol. 2, pp. 411-414, Sep. 2010 (in Japanese).