LETTER

Acceleration of Computing the Kleene Star in Max-Plus Algebra Using CUDA GPUs

Hiroyuki GOTO[†a)], ***Member***

SUMMARY This research aims to accelerate the computation module in max-plus algebra using CUDA technology on graphics processing units (GPUs) designed for high-performance computing. Our target is the Kleene star of a weighted adjacency matrix for directed acyclic graphs (DAGs). Using a inexpensive GPU card for our experiments, we obtained more than a 16-fold speedup compared with an Athlon 64 X2.

key words: *Kleene star, max-plus algebra, adjacency matrix, DAG, GPU, CUDA*

1. Introduction

This research aims to accelerate the computation of the Kleene star [1] of weighted adjacency matrices in max-plus algebra [1], [2], using computers equipped with high performance graphics processing units (GPUs). We implement a program using CUDA (compute unified device architecture) technology [3], which is available on recent NVIDIA GPUs.

The Kleene star plays an essential role in max-plus algebra approaches to scheduling problems for repetitive discrete event systems (DESs). To be precise, the governing equation in max-plus algebra, referred to as the state equation, includes the Kleene star in the transition matrix [4].

Hereafter, we focus on DESs whose behavior can be described by a directed acyclic graph (DAG). Let the number of nodes and arcs in the system be n and m, respectively. If we compute the Kleene star based on the most efficient algorithm known thus far, the time complexity is $O(n \cdot (n + m))$ [4], [5]. On the other hand, the state equation includes other addition and multiplication operations, the worst time complexity of which is $O(n^2)$. Thus, the bottleneck in computing the state equation lies in the Kleene star.

In the field of high performance computing, on the other hand, much attention has been paid to the concept of general-purpose computing on graphics processing units (GPGPU). In particular, recent GPU cards produced by NVIDIA Corporation provide substantial benefits for parallel computation, and the company itself supplies an easy-to-implement environment for developers and researchers. Recently, the effectiveness and advantages of using GPUs for technical computations have been widely reported [6]–[8].

In view of this, we aim to accelerate the computation of the Kleene star in max-plus algebra by implementing code for CUDA GPUs. Then, we measure the effective speedup using both single and multiple GPUs.

Manuscript received September 1, 2010.

[†]The author is with Nagaoka University of Technology, Nagaoka-shi, 940–2188 Japan.

a) E-mail: hgoto@kjs.nagaokaut.ac.jp

DOI: 10.1587/transinf.E94.D.371

2. Target Algorithm

First, we introduce the specific notations and operation rules in max-plus algebra. Denoting the real field by $\mathcal{R}$, we define a field $\mathcal{R}_{\max} = \mathcal{R} \cup \{-\infty\}$. Then, for $x, y \in \mathcal{R}_{\max}$, we define operators and unit elements: $x \oplus y = \max(x, y)$, $x \otimes y = x + y$, ε $(= -\infty)$, and e $(= 0)$. If $m \le n$, $\oplus_{k=m}^{n} x_k = \max(x_m, x_{m+1}, \cdots, x_n)$. For matrices $\boldsymbol{X}, \boldsymbol{Y} \in \mathcal{R}_{\max}^{m \times n}$, and $\boldsymbol{Z} \in \mathcal{R}_{\max}^{n \times q}$, $[\boldsymbol{X} \oplus \boldsymbol{Y}]_{ij} = [\boldsymbol{X}]_{ij} \oplus [\boldsymbol{Y}]_{ij}$ and $[\boldsymbol{X} \otimes \boldsymbol{Z}]_{ij} = \oplus_{l=1}^{n} ([\boldsymbol{X}]_{il} \otimes [\boldsymbol{Z}]_{lj})$. For the unit matrices, $\boldsymbol{\varepsilon}$ is a matrix whose elements are all ε, while $\boldsymbol{e}$ is a matrix with diagonal elements set to $\boldsymbol{e}$ and off-diagonal elements to ε. Operator $\otimes$ has higher precedence than $\oplus$.

Let $\boldsymbol{X} \in \mathcal{R}_{\max}^{n \times n}$ be a DAG weighted adjacency matrix. Our target algorithm is the computation of:

$$\boldsymbol{X}^* = \oplus_{l=0}^{r-1} \boldsymbol{X}^{\otimes l} = \boldsymbol{e} \oplus \boldsymbol{X} \oplus \cdots \oplus \boldsymbol{X}^{\otimes (r-1)},$$

where $[\boldsymbol{X}]_{ij} = \{w_{ij}$: if there is an arc $j \to i$, else $\varepsilon\}$, and w_{ij} is the weight of arc $j \to i$. If we denote the number of nodes by n, there is an instance r that satisfies $\boldsymbol{X}^{\otimes (r-1)} \neq \boldsymbol{\varepsilon}$ and $\boldsymbol{X}^{\otimes r} = \boldsymbol{\varepsilon}$ $(1 \le r \le n)$. It is known that $[\boldsymbol{X}^*]_{ij}$ gives the maximum value of the cumulative weights for paths from node j to node i.

Amongst the most efficient algorithms for computing the Kleene star in terms of time complexity, the method in [5] is attractive, since the work matrix can be partitioned into arbitrary column major blocks and each block can be processed independently. The essential procedures and time complexities are given below.

- Topological sort, $O(m+n)$: sort the nodes in topological order based on a depth first search (DFS) algorithm [9] by inspecting the elements of $\boldsymbol{X}$.
- Initialization, $O(n^2)$: prepare and initialize a work matrix $\boldsymbol{W} \in \mathcal{R}_{\max}^{n \times n}$.
- Update, $O(m \cdot n)$: update the work matrix according to $[\boldsymbol{W}]_{i:} \Leftarrow [\boldsymbol{W}]_{i:} \oplus [\boldsymbol{X}]_{i\bar{l}} \otimes [\boldsymbol{W}]_{\bar{l}:}$ for all succeeding nodes i of source node $\bar{l}$, where $\bar{l}$ represents the original node number of sequence l in the topologically sorted graph. Then, repeat this for all l $(1 \le l \le n-1)$ in ascending order.

On completion of these procedures, the values in the resulting matrix are given by the elements in $\boldsymbol{W}$. We note here

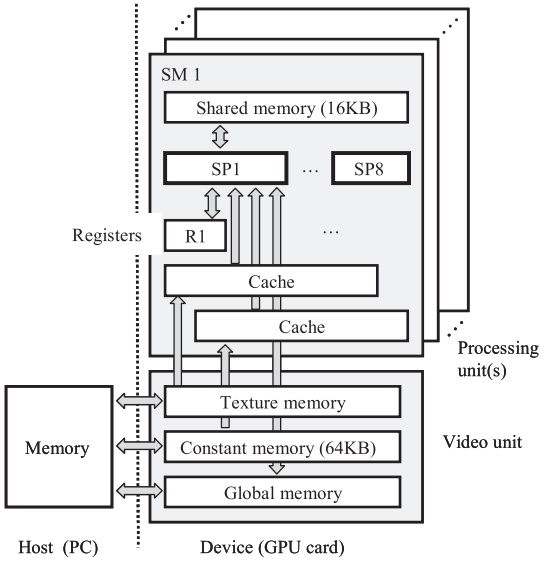

Fig. 1 CUDA hardware model.

that the third process corresponds to an elementary transformation in conventional algebra.

3. CUDA Architecture

The basic structure of a CUDA GPU is depicted in Fig. 1. We suppose here that only a single GPU card is installed on the target PC. In CUDA terminology, the PC and GPU card are called the host and device, respectively. On the device side, there can be either a single or multiple processing units, referred to as the streaming multi-processor (SM). Each SM has eight scalar processors (SPs), a 16 KB shared memory, registers, and two types of caches. The number of SPs in an SM is not always eight; in recent high-end GPUs there are 32 SPs. The 16 KB shared memory is shared between the SPs and has small latency. Computational programs for the SPs are referred to as kernels, with each SP in charge of a task identified by a thread.

In the video unit, depicted in the lower part of the figure, there are three types of memories: global, constant, and texture memories, which are shared between all SMs. To communicate data between the host and device, we must use the global memory, but its latency is quite large. On the other hand, the texture and constant memories are read-only for SMs and accessed data are cached. Thus, the latency can be significantly reduced by accessing the same or adjacent data multiple times. Owing to there being various types of memories in CUDA GPUs, we have to consider well in advance which memories to use, in order to exploit the benefits for computation speed.

4. Implementation

First, we improve the algorithm to reduce the required memory. In existing methods, the update process is performed using $[\boldsymbol{X}]_{i\bar{l}}$ in the original adjacency matrix $\boldsymbol{X}$. This implies allocating sufficient memory to store two $n \times n$ matrices: $\boldsymbol{X}$ and $\boldsymbol{W}$. On the other hand, the number of non-ε (non-zero, in conventional algebra) elements, denoted by m, follows $m \leq n \cdot (n-1)/2$ because we are focusing on DAGs. Thus, using a full matrix workspace is redundant for large scale systems.

In view of this, we first convert $\boldsymbol{X}$ to a compressed form and the remaining procedures are performed using the compressed data. It should be noted that the target algorithm occasionally needs the list of succeeding nodes for a given source node. As a format suited to this, we adopt the compressed column storage (CCS) format [10]. Let the integer field be denoted by $\mathcal{Z}$, then the compression result yields the following three arrays.

- Val ($\in \mathcal{R}^m_{\max}$): stores the values of non-ε elements in $\boldsymbol{X}$ in column major order.
- Idx ($\in \mathcal{Z}^m_{\max}$): stores the corresponding row numbers of the elements in array 'Val'.
- Ptr ($\in \mathcal{Z}^{n+1}_{\max}$): stores the start positions of each column in arrays 'Val' and 'Idx'.

Once the memory space for these arrays has been prepared, if the original matrix $\boldsymbol{X}$ is not needed after the Kleene star computation, this space can be reused for the work matrix $\boldsymbol{W}$.

We now implement the code for CUDA. As shown in the next section, the bottleneck in the Kleene star computation lies in the update process. Thus, we optimize this part extensively.

In preparation, floating point memory storage for the work matrix $\boldsymbol{W}$ is prepared in global memory. This matrix is initialized to $\boldsymbol{e}$, where we use (-FLT_MAX) to represent ε. Moreover, we prepare two arrays for storing 'Val' and 'Ptr' in texture memory. As pointed out in the previous section, several alternative memories are available. In fact, we experimented with code that used the shared and constant memories, but the performance thereof was not good. Thus, we opted to use texture memory.

Then, $\boldsymbol{W}$ is updated sequentially in topological order from upstream source nodes to downstream ones. Figure 2 depicts the update process for source node $\bar{l}$. The list of succeeding nodes, in other words destination nodes, is obtained from $\mathrm{Idx}(\mathcal{S})$, where $\mathcal{S} = \mathrm{Ptr}(\bar{l}) : (\mathrm{Ptr}(\bar{l}+1) - 1)$. Let an element from $\mathcal{S}$ and the number of elements of $\mathcal{S}$ be denoted as $i_k \in \mathcal{S}$ and $|\mathcal{S}| = s$, respectively. First, the values of $[\boldsymbol{X}]_{i_k\bar{l}}$ and i_k $(1 \leq k \leq s)$, which are obtained from $\mathrm{Val}(\mathcal{S})$ and $\mathrm{Idx}(\mathcal{S})$, respectively, are transferred from host memory to texture memory. Next, the values of $[\boldsymbol{W}]_{\bar{l}j}$ $(1 \leq j \leq n)$ are transferred to texture memory. Then, we invoke a kernel to update $[\boldsymbol{W}]_{i_k j}$ $(1 \leq k \leq s,\ 1 \leq j \leq n)$.

On the kernel side, each invoked thread retrieves the value of the target element $[\boldsymbol{W}]_{i_k j}$ from the global memory, and $[\boldsymbol{X}]_{i_k\bar{l}}$ and $[\boldsymbol{W}]_{\bar{l}j}$ from the texture memory. Then, the thread compares $[\boldsymbol{W}]_{i_k j}$ with $[\boldsymbol{X}]_{i_k\bar{l}} \otimes [\boldsymbol{W}]_{\bar{l}j}$ and updates the former if the latter value is greater. Here it should be noted that the comparison and update must not be executed

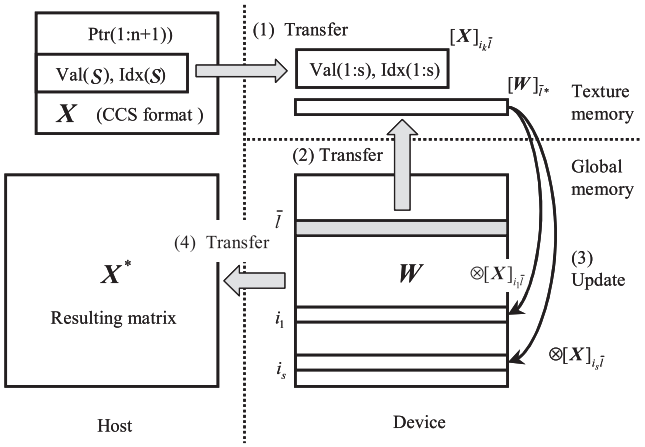

Fig. 2 Update process for source node $\bar{l}$.

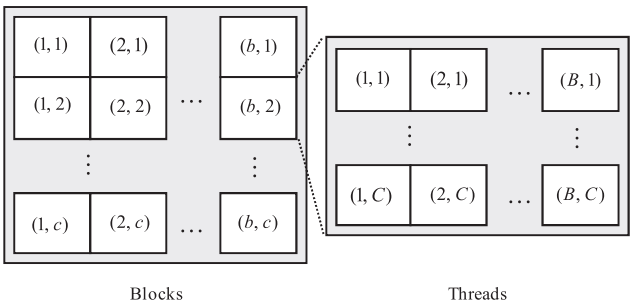

Fig. 3 Hierarchy structure of blocks and threads in a kernel.

if $[\boldsymbol{W}]_{\bar{l}j} = \varepsilon$. As implied by the above, a huge number of conditional branches occur in max-plus algebra operations. Since there is no branch predictor in GPU processors, this feature may be disadvantageous; dissimilar to floating point computations in conventional algebra.

The kernel is invoked for every source node with one or more succeeding nodes. We illustrate the allocation of blocks and threads in the kernel in Fig. 3. The kernel includes $c \times b$ two-dimensional blocks, with each block having $C \times B$ two-dimensional threads, where $b = \lceil n/B \rceil$ and $c = \lceil s/C \rceil$. In current NVIDIA GPUs, $B \cdot C \leq 512$ must be followed, and B should be a multiple of 16 for efficient access to global memory, known as coalescing [3]. Thus, B and C should be set with care. We should also note here that the update location for $\boldsymbol{W}$ is continuous with respect to row order, but scattered with respect to column order. After all updates for the source nodes $\bar{l}$ $(1 \leq l \leq n-1)$ have been completed, the values of $\boldsymbol{X}^*$ are stored in $\boldsymbol{W}$, and the resulting array is transferred from device to host.

For simplicity, we assume that only a single GPU is available in the current explanation. However, if multiple GPUs are available simultaneously, the work matrix can be partitioned column-wise into an arbitrary number of different sized blocks, and the update process can be executed independently in parallel.

5. Performance Evaluation

The performance of the proposed algorithm is measured using a PC equipped with CUDA GPUs. We use a PC installed with (a) an AMD Athlon™ 64 X2 Dual Core 5600+ 2.90 GHz running Linux Fedora 13 for x86-64, and two NVIDIA GPU cards, namely (b) a GeForce GTS 250 and (c) a Quadro NVS 420.

Table 1 shows the specifications of the two GPU cards, noting that cards (b) and (c) have one and two GPUs on a single card, respectively. The compilation and execution environments are: CUDA driver version: 195.36.31; software development kit (SDK) version: 3.0; gcc version 4.4.4 with '-march=athlon64-sse3 -O3' compilation options; and nvcc for CUDA version V0.2.1221 with the '-O3' option.

Now we prepare an adjacency matrix $\boldsymbol{X}$ and compute the Kleene star $\boldsymbol{X}^*$. For $\boldsymbol{X}$, we first attach arcs $i \to i+1$ for all i $(1 \leq i \leq n-1)$, and then append arcs $j \to i$ $(1 \leq j \leq n-1, j+1 \leq i \leq n)$ with 1/2 probability. The weights of these arcs obey a [0, 1] normal distribution. Then, we sort the indices of the nodes randomly, and swap the corresponding rows and columns. Each experiment is performed three times with the same random seed, and the median computation time is adopted.

First, we measure the performance using only the CPU (a). Table 2 shows the computation times in milliseconds with a varying number of nodes n = 500, 1000, 2000, and 4000. As the results clearly indicate, the procedure for updating $\boldsymbol{W}$ is the bottleneck and this part requires extensive fine-tuning. Recalling that the time complexity for the update is $O(m \cdot n)$ and noting $m \cong n^2/4$ holds in this experiment, the computation time would increase eight fold if n were doubled. This estimation actually holds true for larger n.

Table 3 shows the computation times for updating $\boldsymbol{W}$ with varying block sizes B and C for n = 4000 using GPU (b) or (c). It appears that setting B = 64 or B = 32 is acceptable, although the times are not remarkably different for each B.

Table 4 shows the computation times for computing $\boldsymbol{X}^*$ and the speedup effect compared with CPU (a). The speedup is defined as (computation time using a CPU) / (computation time using GPU(s)). Cases 1 and 2 use a single GPU, whereas case 3 uses two GPUs simultaneously. In using multiple GPUs simultaneously, we must invoke multiple threads on the CPU side, known as the POSIX thread, and this requires some overhead for invocation and termination.

In case 1, the speedup is evident as n increases, but it seems to level off between 17 and 18. In cases 2 and 3, the speedup appears to level off around 1.2 and 2.3, respectively. From Table 1, the theoretical computation performance can be estimated by multiplying the number of cores by the SP clock speed. In this context, the ratio of the computations speed for case 1 to case 2 should be approximately 17.1, but the actual ratio was approximately 13.7 for n = 4000. This

Table 1 Specifications of the two GPU cards.

	(b) GeForce GTS 250	(c) Quadro NVS 420
Number of cores	128	8 (x 2)
SP clock (MHz)	1,500	1,400
Memory size	512 MB	256 MB (x 2)
- interface	256-bit	64-bit (x 2)
- bandwidth (GB/s)	115.2	11.2 per GPU
- clock (MHz)	1,800	700

Table 2 Computation times using only CPU (a) (ms).

Nodes n =	500	1,000	2,000	4,000
Arcs m =	62,496	250,014	1,001,134	4,001,435
Convert CCS	2.0	16.0	83.2	653.3
Topological sort	0.4	1.6	5.8	23.2
Initialize $\boldsymbol{W}$	0.6	1.6	6.6	26.4
Update $\boldsymbol{W}$	123.0	1,117.4	9,136.3	72,934.2
Total	126.0	1,136.6	9,231.9	73,637.1

Table 3 Computation times for updating $\boldsymbol{W}$ with varying B and C sizes (ms).

B	16	32	64	128	256	512
C	32	16	8	4	2	1
(b)	3,981	3,788	3,771	3,796	3,795	3,793
(c) ×1	62,440	60,678	61,027	61,740	61,831	62,186

Table 4 Computation times (ms) and speedup effects using GPUs.

	GPU(s)	n = 500	1,000	2,000	4,000
1	(b)	32.3	124.8	649.8	4,509.3
	Effect	3.90	9.11	14.21	16.33
2	(c) × 1	156.1	1,023.5	8,068.8	61,763.3
	Effect	0.81	1.11	1.14	1.19
3	(c) × 2	168.7	630.4	3,978.7	32,286.8
	Effect	0.75	1.80	2.32	2.28

difference may be due to the relatively higher time to convert X to the CCS format in case 1 compared with case 2.

We note here that the results using GPUs matched those using the CPU exactly, which indicates that the computation precision of GPUs conforms to standard single precision.

6. Conclusion

In this research, we focused on accelerating the computation of the Kleene star in max-plus algebra using CUDA GPUs. The primary target was updating the work matrix, the process of which is similar to elementary transformations of a matrix in conventional algebra. Since the SPs do not have a branch predictor, they are not naturally efficient for max operations. Nevertheless, we accomplished a speedup greater than 16-fold with a reasonable GPU, compared with an Athlon 64 X2 including the SSE3 optimization option. Accordingly, it is expected that the speedup would be much more significant using high-end GPUs.

References

[1] B. Heidergott, G.J. Olsder, and L. Woude, Max Plus at Work: Modeling and Analysis of Synchronized Systems, Princeton University Press, New Jersey, 2006.
[2] F. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat, Synchronization and Linearity, John Wiley & Sons, New York, 1992. http://maxplus.org
[3] NVIDIA Corporation, "CUDA programming guide version 3.0," Aug. 2010. http://developer.nvidia.com/cuda/
[4] H. Goto, "Efficient calculation of the transition matrix in a max-plus linear state-space representation," IEICE Trans. Fundamentals, vol.E91-A, no.5, pp.1278–1282, May 2008.
[5] H. Goto and H. Takahashi, "Fast computation methods for the Kleene star in max-plus linear systems with a DAG structure," IEICE Trans. Fundamentals, vol.E92-A, no.11, pp.2794–2799, Nov. 2009.
[6] P. Harish and P.J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," Proc. 14th Int. Conf. High Performance Comput. (HiPC '07), pp.197–208, 2007.
[7] Y. Munekawa, F. Ino, and K. Hagiwara, "Accelerating Smith-Waterman algorithm for biological database search on CUDA-compatible GPUs," IEICE Trans. Inf. & Syst., vol.E93-D, no.6, pp.1479–1488, June 2010.
[8] A. Buluç, J.R. Gilbert, and C. Budak, "Solving path problems on the GPU," Parallel Comput., vol.36, no.5–6, pp.241–253, 2010.
[9] T. Cormen and C. Leiserson, Introduction to Algorithms, MIT Press, Massachusetts, 2001.
[10] Y. Saad, "SPARSKIT: A basic tool kit for sparse matrix computations," Tech. Rep. 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.