Efficient Computation of the Kleene Star in Max-Plus Algebra using a CUDA GPU

HIROYUKI GOTO and KAZUHIRO TOYODA
Department of Industrial and System Engineering
Hosei University
3-7-2 Kajino-cho, Koganei, Tokyo 184-8584 JAPAN
JAPAN
{goto-h@, 10x4058@stu.}hosei.ac.jp

Abstract: - This research aims to accelerate the computation of the Kleene star in max-plus algebra using CUDA technology on graphics processing units (GPUs). The target module is the Kleene star of a weighted adjacency matrix for directed acyclic graph (DAGs) which plays an essential role in calculating the earliest and/or latest schedule for a class of discrete event systems. In recent NVIDIA GPU cards, an environment for high performance computing is provided to general developers, for which we aim to exploit the benefit of using GPUs. Using an NVIDIA Tesla C2075 for our experiments, we obtained approximately a 30-fold speedup compared with an Intel Xeon E5645.

Key-Words: - Directed acyclic graph, adjacency matrix, Kleene star, max-plus algebra, GPU, CUDA

1 Introduction
The primary concern of this research is 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], [7].

In view of this, we aim to accelerate the computation of the Kleene star in max-plus algebra. We implement a code for CUDA GPUs, and measure the speedup effect.

2. Target Algorithm
We first introduce the specific notations and operation rules in max-plus algebra. Denoting the real field by \( \mathbb{R} \), we define a field \( \mathbb{R}_{\max} = \mathbb{R} \cup \{-\infty\} \). Then, for \( x, y \in \mathbb{R}_{\max} \), we define operators and unit elements:

\[
\begin{align*}
\max(x, y) & = \max(x, y), \quad x \oplus y = x + y + \varepsilon \quad (=-\infty), \quad \varepsilon = (0) \quad \text{if} \quad m \leq n,
\end{align*}
\]

For matrices \( X, Y \in \mathbb{R}_{\max}^{m \times n} \), and \( Z \in \mathbb{R}_{\max}^{n \times q} \),

\[
[X \oplus Y]_{ij} = [X]_{ij} \oplus [Y]_{ij}, \quad [X \odot Z]_{ij} = \oplus_{k=0}^{n} ([X]_{ik} \odot [Z]_{kj}).
\]

For the unit matrices, \( \varepsilon \) is a matrix whose elements are all \( \varepsilon \), while \( \varepsilon \) is a matrix with diagonal elements set to \( \varepsilon \) and off-diagonal elements to \( \varepsilon \). Operator \( \odot \) has higher precedence than \( \oplus \).
Let $X \in \mathbb{R}^{n \times n}$ be a DAG weighted adjacency matrix. Our target algorithm is the computation of:

$$X^* = \bigoplus_{r=0}^{r-1} X^{\otimes r} = e \oplus X \oplus \ldots \oplus X^{\otimes(r-1)},$$

where

$$[X]_{ij} = \{w_{ij} : \text{if there is an arc } j \to i, \text{ else } \varepsilon \},$$

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 $X^{\otimes(r-1)} \neq \varepsilon$ and $X^{\otimes r} = \varepsilon$ ($1 \leq r \leq n$). It is known that $[X^*]_{ij}$ 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 [8] by inspecting the elements of $X$.
- **Initialization**, $O(n^2)$: prepare and initialize a work matrix $W \in \mathbb{R}^{n \times n}$.
- **Update**, $O(m \cdot n)$: update the work matrix according to $[W]_{ij} \leftarrow [W]_{ij} \oplus [X]_{ij} \otimes [W]_{i\bar{l}}$ 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 $l$ ($1 \leq l \leq n-1$) in ascending order.

On completion of these procedures, the values in the resulting matrix are given by the elements in $W$. We note here 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. In CUDA terminology, the PC and GPU card are called the host and device, respectively. On the device side, there is either a single or multiple processing units, referred to as the streaming multiprocessor (SM). Each SM has 8, 32 or 48 CUDA cores (Core in Fig. 1), a 16KB shared memory, registers, and two types of caches. The 16KB shared memory is shared between the cores and has small latency. Computational programs for the CUDA cores 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 $[X]_{ij}$ in the original adjacency matrix $X$. This implies allocating sufficient memory to store two $n \times n$ matrices: $X$ and $W$. On the other hand, the number of non- $\varepsilon$ (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 $X$ 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 [9]. Let the integer field be denoted by \( z \), then the compression result yields the following three arrays.

- \( \text{Val} (\in \mathbb{R}^m_{\max}) \): stores the values of non-\( \varepsilon \) elements in \( X \) in column major order.
- \( \text{Idx} (\in \mathbb{Z}^m_{\max}) \): stores the corresponding row numbers of the elements in array ‘Val’.
- \( \text{Ptr} (\in \mathbb{Z}^{m+1}_{\max}) \): 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 \( X \) is not needed after the Kleene star computation, this space can be reused for the work matrix \( W \).

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 \( W \) is prepared in global memory. This matrix is initialized to \( e \), where we use (-FLT_MAX) to represent \( e \). 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, \( W \) is updated sequentially in topological order from upstream source nodes to downstream ones. Fig. 2 depicts the update process for source node \( \bar{i} \). The list of succeeding nodes, in other words destination nodes, is obtained from \( \text{Idx}(\bar{s}) \), where \( \bar{s} = \text{Ptr}(\bar{i}); (\text{Ptr}(\bar{i}+1)-1) \). Let an element from \( \bar{s} \) and the number of elements of \( \bar{s} \) be denoted as \( i_s \in \bar{s} \) and \( |\bar{s}| = s \), respectively. First, the values of \( [X]_{i_s \bar{i}} \) and \( i_k (1 \leq k \leq s) \), which are obtained from \( \text{Val}(\bar{s}) \) and \( \text{Idx}(\bar{s}) \), respectively, are transferred from host memory to texture memory. Next, the values of \( [W]_{ij} (1 \leq j \leq n) \) are transferred to texture memory. Then, we invoke a kernel to update \( [W]_{ik} (1 \leq k \leq s, 1 \leq j \leq n) \).

On the kernel side, each invoked thread retrieves the value of the target element \( [W]_{ik} \) from the global memory, and \( [X]_{ij} \) and \( [W]_{ij} \) from the texture memory. Then, the thread compares \( [W]_{ik} \) with \( [X]_{ij} \otimes [W]_{ij} \) and updates the former if the latter value is greater. Here it should be noted that the comparison and update must not be executed if \( [W]_{ij} = e \). 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 \( \bar{i} \) is continuous with respect to row order but scattered with respect to column order. After all updates for the source nodes \( \bar{i} \) (\( 1 \leq l \) \( n-1 \)) have been completed, the values of \( X^* \) are stored in \( W \), and the resulting array is transferred from device to host.

![Fig. 2. Update process for source node \( \bar{i} \).](image)

![Fig. 3. Hierarchy structure of blocks and threads in a kernel.](image)
For simplicity, we assumed that only a single GPU is available in the current explanation. If multiple GPUs are available, 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. We use a PC installed with an Intel® Xeon® E5645 2.40GHz running Linux CentOS 6.3 for x86-64, equipped with an NVIDIA Tesla® C2075. The specifications of Tesla C2075 are shown in Table 1. The compilation and execution environments are:

- CUDA driver: version 304.54,
- Software development kit: SDK version 5.0,
- Compiler for the CPU: gcc version 4.4.6 with ‘-march=native -O3’ options,
- Compiler for the GPU: nvcc version V0.2.1221 with ‘-O3’ option.

We prepare an adjacency matrix $X$ and compute the Kleene star $X^*$. For $X$, we first attach arcs $i \rightarrow i+1$ for all $i$ ($1 \leq i \leq n-1$), and then append arcs $j \rightarrow i$ ($1 \leq j \leq n$ and $j+1 \leq i \leq n$) 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. Regarding the block sizes $B$ and $C$ in Fig. 3 for updating $W$, $B=64$ and $C=8$ are adopted.

Each experiment is performed five times with the same random seed, and the average of the medium three computation times is adopted. First, we measure the performance using only the Xeon CPU. Table 2 shows the computation times in milliseconds with a varying number of nodes $n=500, 1000, 2000, 4000$. As the results clearly indicate, the procedure for updating $W$ is the bottleneck and this part requires extensive tuning. Recalling that the time complexity for the update is $O(m \cdot n)$ and noting $m \approx n^2/4$ 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 using the Tesla GPU card and the speedup effect compared with the CPU. The speedup is defined as (computation time using a CPU) / (computation time using a GPU). Since the first three procedures do not use a GPU, the corresponding computation times are almost the same as the ones in Table 2. The last row represents the speedup effects in the total computation times. Though the speedup is evident as $n$ increases, the effect may level off if we compute for larger number of nodes.

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

6. Conclusion

We have focused on accelerating the computation of the Kleene star in max-plus algebra using a CUDA GPU. 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 CUDA cores do not have a branch predictor, they are not naturally efficient for max operations. Nevertheless, we accomplished a speedup approximately 30-fold with a Tesla GPU, compared with an Intel Xeon CPU.
References:


