PARALLELIZATION OF THE DEMPSTER-SHAFER APPLICATION TO TALBOT'S METHOD FOR NUMERICAL INVERSE LAPLACE TRANSFOMS

By

JONATHAN AARON CAIN

---

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Mathematics

THE UNIVERSITY OF ARIZONA

M A Y 2 0 1 1

Approved by:

Dr. Moysey Brio
Department of Mathematics

# The University of Arizona Electronic Theses and Dissertations Reproduction and Distribution Rights Form

| | |
|---|---|
| Name (Last, First, Middle)<br>Cain, Jonathan Aaron | |
| Degree title (eg BA, BS, BSE, BSB, BFA): BS | |
| Honors area (eg Molecular and Cellular Biology, English, Studio Art): Math | |
| Date thesis submitted to Honors College: 5/2/2011 | |
| Title of Honors thesis:<br>Parallelization of the Dempster-Shafer Application to Talbot's Method for Numerical Inverse Laplace Transforms | |
| :The University of Arizona Library Release | I hereby grant to the University of Arizona Library the nonexclusive worldwide right to reproduce and distribute my dissertation or thesis and abstract (herein, the "licensed materials"), in whole or in part, in any and all media of distribution and in any format in existence now or developed in the future. I represent and warrant to the University of Arizona that the licensed materials are my original work, that I am the sole owner of all rights in and to the licensed materials, and that none of the licensed materials infringe or violate the rights of others. I further represent that I have obtained all necessary rights to permit the University of Arizona Library to reproduce and distribute any nonpublic third party software necessary to access, display, run or print my dissertation or thesis. I acknowledge that University of Arizona Library may elect not to distribute my dissertation or thesis in digital format if, in its reasonable judgment, it believes all such rights have not been secured.<br>Signed: Jonathan Cain<br>Date: 5/2/11 |

Last updated: Nov 15, 2009

# PARALLELIZATION OF THE DEMPSTER-SHAFER APPLICATION TO TALBOT'S METHOD FOR NUMERICAL INVERSE LAPLACE TRANSFORMS

JONATHAN CAIN

ABSTRACT. There are three major focuses for this thesis. First is an introduction to the Dempster-Shafer theory of evidential reasoning and its application to Tablot's method for the numerical inverse Laplace Transform (DSNLAP Method). Second is a brief introduction to parallel scientific computing and the utilization Grapics Processing Units (GPUs). Third is a parallelization and analysis of the DSNLAP method as implemented via Accelereyes' Jacket Library on GPUs.

CONTENTS



## 1. OVERVIEW OF RESULTS

The standard approach for numerical Laplace inversion via Bromwich contour integration is ill-posed and subject to catastrophic error due to the $e^{st}$ integrand with $\text{Im}(s) \to \pm\infty$. Talbot's method replaces the Bromwich contour with a contour $L$ that is only in the left half complex plane effected by $\text{Re}(s) \to -\infty$ at both ends of $L$, [2]. This replacement is legitimate

*Date*: May 2, 2011.

provided that $L$ incorporates all the singularities of $F(s)$, which is done by shifting and scaling $s$. Talbot proposed a contour of steepest decent to explicitly find how $s$ should be scaled and shifted given an $f(t)$, and subsequently there have been other methods (e.g., [1], [3]). By scaling and shifting $s$, one should produce an optimal contour for the given $F(s)$. However, Talbot's Method requires that the user have prior knowledge of the parameters that define $s$ with the purpose to render the correct the contour.

Recently, work has been made at finding such an optimal contour using the Dempster-Shaffer theory of evidential reasoning [10]. By applying this theory, the user need not have the prior knowledge of the singularities and parameters of the contour. Instead, this method judiciously chooses an optimal contour by making an "educated comparison" to known transform pairs in the form of the resolvent matrix $F(s) = (sI - A)^{-1}$ and the matrix exponential $f(t) = e^{At}$. Like most numerical inverse Laplacemethods, "the strategy to be described is based on an intensive study of a large number of experimental results using many different types of function F(s)" [2].

Because of the large requirements for memory and the network concatenation of many algorithms working both simultaneously and sequentially, the computation time for this method given non-mpi (Message Passing Interface) capabilities is disenchanting. However, with the now wide-scale availability of graphic processing units (GPUs), such tasks are reduced in computation time by parallelization. Thus, the final algorithm is presented in the Jacket format for computation on NVIDIAs GPUs.

## 2. Introduction to the DSNLAP Method

The Laplace transform is a two step method: One must first transform the time domain to the Laplacian space,

$$\mathcal{L}[f(t)] = F(s) = \int_0^\infty e^{-st} f(t)dt, \; \mathrm{Re}(s) > \gamma$$

$$f(t) : \mathbb{R} \to \mathbb{C}.$$

Then, one must transform the Laplacian back to the time domain,

$$f(t) = \mathcal{L}^{-1}[F(s)] = \frac{1}{2\pi i} \oint_{\text{Bromwich}} e^{st} F(s)ds, \; t > 0 \tag{1}$$

$$F(s) : \mathbb{C} \to \mathbb{C}.$$

The latter stage of the method is ill-posed due to an exponential amplification of error. In the last century, there have been many techniques developed to remedy this (e.g., [7], [6]).

2.1. **Talbot's Method.** Talbot's Method is one such method for the numerical inversion of the Laplace transform. Consider (1) and let the Bromwich contour be from $c - i\infty$ to $c + i\infty$ for any $c \in \mathbb{R}$ so that $c > \gamma$, $\gamma$ acting as the so-called *abscissa of convergence.* This requirement ensures that all of the singularities of $F(s)$ are to the left of the contour. However, simply evaluating the right side of (1) via numerical quadrature must take into consideration the oscillations of $e^{st}$ in the integrand as $\mathrm{Im}(s) \to \pm\infty$. Talbots Method is literally the consideration of this by the following two requirements:

(1) As $|s| \to \infty$, $|F(s)| \to 0$ uniformly in $\mathrm{Re}(s) < \gamma$.
(2) All singularities $s_j$ of $F(s)$ are known and $\forall j$, $|\mathrm{Im}(s_j)| < a$ for some $a \in \mathbb{R}$.

Talbot ensured the fulfillment of the second requirement by shifting ($\sigma$) and stretching ($\lambda$) $s$, so that if $F(s_j)$ is a singularity, then $F(\lambda s_j + \sigma)$ has the corresponding singularity. The contour,

$$\Gamma(\theta) = \sigma + \lambda\theta(iv + \cot\theta) \tag{2}$$

with geometric parameters

$$\begin{aligned} \sigma &\in (-\infty, \infty) \\ \lambda &\in [0, \infty) \\ v &\in [0, \infty) \\ \theta &\in (-\pi, \pi), \end{aligned}$$

is proposed by Talbot in [2] whose formulation is omitted here. The Laplace inversion (1) becomes:

$$f(t) = \frac{\lambda e^{\sigma t}}{2\pi i} \int_{-\pi}^{\pi} e^{t(\Gamma(\theta)-\sigma)} F(\Gamma(\theta)) h(\theta) d\theta, \tag{3}$$

where

$$h(\theta) = \frac{d}{d\theta}[\frac{1}{\lambda}\Gamma(\theta)]. \tag{4}$$

The difficulty in the method lies in the computation of accurate geometric parameters that are left to the user. It was the purpose of [10] to use a computer learning technique to provide these parameters for accurate integration.

2.2. **Dempster-Shafer Theory.** The Dempster-Shafer theory (DS) of evidence is a generalization of classic probability theory, most illustrated by the assignment of probabilities to sets of events, rather than events themselves. Information based on events that may not be to varying degrees, precise or accurate can be assimilated into the probabilistic model by way of these sets. Instead of acting on events of the sample space, DS acts on elements of the power set of the universal set of all possibilities, $P(\Omega)$. DS works primarily with three measures: A basic probability mass function defined by,

$$m : P(\Omega) \to [0, 1],$$

$$m(\emptyset) = 0, \text{ and } \sum_{A \in P(\Omega)} m(A) = 1, \tag{5}$$

where $A$ represents the proportion of evidence supporting the claim that a particular element in $X$ belongs to the set $A$, but to no particular subset of $A$. The other two DS measures, a belief function and a plausibility function, define the lower and upper bounds, respectively, for an interval of probabilities in the classic sense. Formally, for every $A \subset P(X)$,

$$Bel(A) = \sum_{B:B\subseteq A} m(B)$$

$$Pl(A) = \sum_{B:B\cap A=\emptyset} m(B).$$

These are two continuous measures that are non-additive, meaning that summing $Bel$ or $Pl$ over all $A \in P(X)$ is not necessarily 1. If $\mathcal{P}(A)$ represents the classic probability, then

$Bel(A) \leq \mathcal{P}(A) \leq Pl(A)$ [8]. One natural question that arises from the above basic description is how to transverse the elements of the power set of the universal set, combining the three aforementioned measures. The answer is that there are many of them and that they depend on the specific application. Thus, a unique combination for the DSNLAP Method is required. Recall that to satisfy the Talbot inversion integral (3) requires computation of the geometric parameters of the contour (2). We assume our input information of the Laplacian is in the form of the resolvent matrix, $F(sI - Q)$, for an input matrix $Q$; also, the matrix exponential is computed, $e^{Qt}$ for input times $t$ using MATLAB's *expm* file that is assumed to be accurate in double precision. We will see that information stored in the resolvent and the exponential will be key for determining what contour to choose.

2.3. **Test Classes and the Frame of Discernment.** To create the contours, we define six classes of input test matrices, $Q$, taken the Mathworks MATLAB Gallery. Each is representative of a specific spectral behavior that present challenges for accuracy and stability in computation of the matrix exponential [9]. Let $Q$ be a square matrix with dimension $n$. The six classes are:

(1) Pei: For some $\alpha \in \mathbb{C}$, $Q_1 = \alpha * I + J_n$, where $J_n$ is the square matrix of $n$ ones. This matrix is symmetric, singular if $\alpha = 0$ or $\alpha = -n$, and has all but one eigenvalue at $\lambda = 1$ with $n-1$ degeneracy and one eigenvalue at $\lambda = n+1$.
(2) Hanowa: For $n = 2m$, $m \in \mathbb{N}$, let $d \in \mathbb{C}$. Then, the following definition for $Q_2$ renders the eigenvalues $\lambda \in \{d \pm ki | 1 \leq k \leq m\} \subset \mathbb{C}$.
$$Q_2 = \begin{bmatrix} d * eye(m) & -diag(1:m) \\ diag(1:m) & d * eye(m) \end{bmatrix}.$$
(3) Min$_{ij}$: A positive symmetric definite matrix, $Q_3$ defined by $Q_3(i,j) = \min(i,j)$.
(4) Parter: A Cauchy and Toeplitz matrix, $Q_4$, defined by $Q_3(i,j) = \frac{1}{i-j+\frac{1}{2}}$. Most of the singular values are close to $\pi$.
(5) RIS: The Symmetric Hankel Matrix defined by $Q_5(i,j) = \frac{1}{2(n-i-j+\frac{3}{2})}$. This tends the eigenvalues to cluster around $\frac{\pi}{2}$ and $-\frac{\pi}{2}$.
(6) Redheff: Redheffer's matrix of zeros and ones defined by:
$$Q_6(i,j) = \begin{cases} 1 & \text{if } j = 1 \text{ or } \frac{i}{j} \in \mathbb{N} \\ 0 & \text{Otherwise} \end{cases}.$$

The test matrices $Q$ are for the determination of the accuracy of the final results; i.e., one could use any input matrix $Q$ needed for application, but to find how truly successful the method is, we present the above test matrices to give a worse-case scenario for computation accuracy.

2.4. **Feature Extraction and Contour Rules.** Accuracy is determined by comparison of the matrix exponential of the input matrix $Q$ to MATLAB's *expm* function. If we the Laplacian as the resolvent matrix, $F(s) = (sI - Q)^{-1}$, and the we consider only the case that $f(s) = e^{st}$, then the inverse Laplace formula, Equation (1) becomes,
$$e^{Qt} = f(Q) = \frac{1}{2\pi i} \oint_{\Gamma} e^{st}(sI - Q)^{-1} ds.$$
The contour $\Gamma$ from above is defined to be a Talbot's contour given by Equation (2). Given an input matrix $Q$ from one of the six test classes, we begin by extracting features of these

matrices. For our input matrix, $Q$, four features considered, and such are the basis for creating a basic probability mass function as well as our hypothesis definition from above. These four features are intended to provide information on the spectral behavior of $Q$; but there are others that are just as comprehensive. These were chosen because of their quick computation time.

(1) Trace of $Q$, denoted $\text{Re}(\text{tr}(Q)) = \text{Re}(\sum_i Q_{ii})$, provides the sum of the eigenvalues of $Q$.
(2) One Norm of $Q$, denoted $\|Q\|_1 = \max_i \left[\sum_j |Q_{ij}|\right]$ is the maximum absolute column sum.
(3) Frobenius Norm of $Q$, denoted $\|Q\|_F = \sqrt{Q^*Q}$, where $Q^*$ is the conjugate transpose. If $Q$ is Hermitian, then $\|Q\|_F$ is the spectral radius of $Q$.
(4) Infinity Norm of $Q$, denoted $\|Q\|_\infty = \max_j \left[\sum_i |Q_{ij}|\right]$ is the maximum absolute row sum.

TABLE 1. Classes of contour rules

| Class | $\sigma$ | $v$ | $\mu$ |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
| 2 | $\text{Re}(\text{tr}(Q)\ )$ | 1 | 1 |
| 3 | 0 | $\frac{1}{\|Q\|}$ | $\|Q\|$ |
| 4 | $\|Q\|$ | $\frac{1}{\|Q\|}$ | $\|Q\|$ |
| 5 | $\|Q\|$ | 1 | 1 |
| 6 | $\text{Re}(\text{tr}(Q)\ )$ | $\|Q\|$ | $\sqrt{v}$ |
| 7 | $\text{Re}(\text{tr}(Q)\ )$ | $\|Q\|$ | $v$ |
| 8 | $\text{Re}(\text{tr}(Q)\ )$ | $\|Q\|$ | $v^2$ |
| 9 | $\text{Re}(\text{tr}(Q)\ )$ | $\|Q\|$ | $v^3$ |

We use these feature extractions to fill in the entries of Table (1), creating our contour rules that satisfy the correct Talbot contour, $\Gamma$. These nine classes (Table (1))of rules form what is called the hypothesis definition, which provides the possible decision states in the DS Algorithm. Let $X$ be the set of features, and $Y$ the nine classes of contour rules. We form the universal set $\Omega = X \times Y$ of size $|\Omega| = 13$, whose power set ($|P(\Omega)| = 2^{13}$) is called the frame of discernment. The frame of discernment can be thought of as feature-contour pairs; it is the power set of the hypothesis definition for each of the four features. Computationally, we treat the frame of discernment as a binary permutation array with $2^{13}$ entries with each entry corresponding to an array of permutation of the classes.

2.5. **Mass Assignment and Fusion.** The basic mass assignment is given by $\mu : P(\Omega) \to [0, 1]$. Each is piecewise continuous and has the property that:

$$\mu(\emptyset) = 0 \text{ and } \sum_{A \in P(\Omega)} \mu(A) = 1$$

These basic mass assignments are determined by the training stage of the algorithm. While these do describe the basic probability mass functions outlined in Equation (5), we are more concerned with the fusion of masses between features to give only mass to the accuracy of the contours that itself satisfies Equation (5). Let $A, B \subset \Omega$ be decision states for the accuracy of some contour rule for all features. For example, $A$ could give information on the second class of contour rules in Table (1); so, $A = \{2\} \times Y$. Let $C \subset P(\Omega)$ denote the hypothesis power set, i.e., the contour specific information in $P(\Omega)$. For features 1 and 2, we get mass assigned to $C$ by

$$m_{1,2}(C) = \frac{\sum_{A\cap B=C} \mu_1(A)\mu_2(B)}{1 - \sum_{A\cap B=\emptyset} \mu_1(A)\mu_2(B)}. \tag{6}$$

This means that the mass assigned to $C$ is ratio of the sum of the mass from the decisions $A$ and $B$ that support $C$, to the sum of the mass in $A$ and $B$ that is in conflict to $C$. For a set of decision states $C$ and $D$ in $P(X)$, we define the belief that $C$ is true by $Bel(C) = \sum_{D:D\subseteq C} m(D)$. There are many ways define the basic mass assignment$\mu$ from above. This process is called the training stage of the DS algorithm. Let $Q$ be a test matrix. In our algorithm, we consider the relative error between MATLAB's *expm* function, $e^Q$, and our matrix exponential $\alpha$, computed from the feature-contour pair $A \in P(X)$. We have that

$$\mu(A) = \exp(-\max_Q \left(\frac{\|e^Q - \alpha\|_F}{\|e^Q\|_F}\right)) \tag{7}$$

Using the maximum gives a worst case bound on all the training test matrices $Q$. However, we wish to give information on the feature-contour rule pair for all the features. Let $\mathcal{F}$ be the set of features. For all $C \subset P(X)$ , we assign $m(C) = \sum_{\mathcal{F}} \mu(A)$ with $A \subset C$. This is really a generalization of Equation(6) as it sums up the relavent information over the features. However, we see that information in $A$ that is in contradiction with the hypothesis decision $C$ is constant due to the maximum in Equation (7).

The DSLNAP Method described above produces competitive results for the numerical inversion of the Laplace transform via Talbot's method. However, the algorithm consists training the algorithm to find the basic mass assignments, and also the execution of the program given an arbitrary input matrix for real life application. While the execution run time is low, the run time for the training is measured in days. Thus, we turn to graphics processing units to ameliorate this problem.

## 3. Introduction to Parallel Computing in Jacket

Graphics processing (GPUs) units offer exciting new capabilities for sometimes drastic reduced computation time in scientific computing applications. Originally engineered for the simultaneous computation of the many three-color pixels in televisions, the development of the GPU has allowed scientist to cheaply use time-reducing techniques in their programs, the process of which is called parallel computing.

### 3.1. Parallel Computing Architecture.

To understand programming on a GPU, we must first understand the what we are programming on. This is called the parallel computing architecture. On a central processing unit, lines of code, or commands, are physically executed sequentially. However, on a GPU, certain types of code are computed simultaneously. The so-called Flynn's taxonomy classifies the types of code that are compatible with GPUs [11], given in Table 2. The CPU is an example of SISD where a single (instruction) processor

TABLE 2. Flynn's Taxonomy

| | Single Instruction | Multiple Instruction |
|---|---|---|
| Single Data | SISD | MISD |
| Multiple Data | SIMD | MIMD |

executes a single data stream sequentially. However, the GPU is an example of SIMD, where a single instuction is executed on multiple data streams simultaneously. When multiple data streams are executed simultaneously, the process is called parallel computing.

On a GPU, we can run thousands of lines simultaneously. This works by first breaking down the instructors into what are called threads: Threads are the smallest unit of processing that can be scheduled by an operating system. These thousands of threads have access to hundreds or core processors. A core processor reads and executes a program by breaking up the data that is sent to the threads, and also concatenating the data from the threads after the threads run the instruction. The hundreds of cores have access to memory that stores the data before and after it is sent to the hundreds of cores. There are many ways for a GPU to physically setup any number of threads, cores, and memory.

What is important from this in terms of writing code is that the instruction sent to the threads must be a single mathematical operation. However, the code can be slightly more complicated because the hundreds of cores judiciously allocate the single instructions to the threads. Thus, at a fixed time during the execution of a code, different threads may be running different basic operations. The idea of optimizing the interaction of threads and cores is not limited to the hardware, but also the code that is run on them.

3.2. **Coding in CUDA.** CUDA is a computer language specifically designed for computation on GPUs [12]. It is a robust language, based in C, that allows the user to control almost any part of the thread/core interchange. CUDA is as much a language as it is an architecture. In order to gain control of the physical interplay between threads and cores, CUDA architecture includes what is called a shared memory that allows for quick access between the two. Because CUDA is quite different from C, or any other CPU-based, procedural software language, coding in CUDA requires new set of skills for proper use.

3.3. **Kernel, Blocks, and Memory.** CUDA does not only run on the GPU, but also the CPU. We call the CPU and the system's memory, the host. The GPU and its memory are called the device. A set of lines that is executed on the device is called a kernel. More specifically, the kernel sends a host code to one complier and the device code to another compiler, while maintaining that same line of code run in both places. In CUDA, one calls the kernel function by,

```
kernel<<<a,b>>>().
```

Placing an argument in the parentheses will pass it to the device. In the double angle brackets, we find two parameters to be given to the kernel. The first parameter, a, tells how many copies of the kernel to be run. These copies are called blocks and together they make what is called a grid. The second parameter, b, is how many threads per block are to run in parallel. There are restrictions as to how many blocks can be run due to the number of cores available. However, with both divisions many simultaneous copies of the kernel may be run. Another important feature of CUDA is the allocation of memory. The CUDA call is

`cudaMalloc(a,b),`

which distributes the memory on the device at runtime. The first argument is a pointer that holds the location of the allocated memory on the device. The second parameter is the size of the allocated memory. Once memory is allocated from the host to the device, this memory cannot be access and modified by the host. Another function,

`cudaMemcpy(a,b,c)`

complements the implementation of the memory allocation by allowing access to the device memory. The first parameter, a, is the destination for the memory stored at the second parameter, b. The last parameter is CUDA-specific: The parameter, c, can be anyone of the following, which tell the specified host-device transfer:

```
cudaMemcpyDeviceToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToDevice
```

The last importat function is the

`cudaFree(a)`

function that after the computation has been sent back to the CPU, we free up the memory space for the parameter a. The use of the kernel and memory allocation between host and device is much more subtle that the crass-course above; however, these tools are sufficient prerequisites for understanding more advanced topics, such as shared memory. The following is a code that adds two vectors:

```
#define N   20

int main( void ){
    int u[N], v[N];
    int *dev_u, *dev_v, *dev_sum;

    // allocation of memory to the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_u, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_v, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_sum, N * sizeof(int) ) );

    // make up u and v on the CPU
    for (int i=0; i<N; i++)
        u[i] = 2*i;
        v[i]=-0.5;
    end

    // put u and v on the GPU
    HANDLE_ERROR( cudaMemcpy( dev_u, u, N * sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_v, v, N * sizeof(int), cudaMemcpyHostToDevice ) );

    add<<<N,1>>>( dev_u, dev_v, dev_sum );
```

```
    // put the sum on the CPU from the GPU
    HANDLE_ERROR( cudaMemcpy( sum, dev_sum, N * sizeof(int), cudaMemcpyDeviceToHost )

    // free the memory on the GPU
    cudaFree( dev_u );
    cudaFree( dev_v );
    cudaFree( dev_sum );

    return 0;
}
```

The add function above, acting like our kernel function, has an input of $N = 10$ and 1, corresponding to the number of blocks and threads per block respectively. There are physical limitations for these numbers. The number fo blocks that can be implimented in a single run is 65,535 and the maximum number of threads per block is 512.

3.4. **Shared Memory.** The process of allocating memory, creating blocks, and splitting blocks into threads is not too complicated as the above code for summing two vectors indicates; however, when the limitations on the sizes of the blocks and threads begin to require a more complicated implimentation, we must begin to transfer data between not only the device and the host, but also between the threads, the blocks, the device, and the host. CUDA has installed a space of memory, called shared memory, that when a variable is stored to it using the typical C function, the complier will put a copy on each block. Thus, each block has the same data to work on, the process of which is called splitting parallel blocks. What is useful about shared blocks is that each thread in the block only sees the data stored to that specific block and no other block.

3.5. **Jacket.** Accelereyes' Jacket Library presents a more user friendly version than CUDA. The Jacket library is a set of GPU enabled functions, written in CUDA, but executable in MATLAB. Jacket is incredibly easy to use, but lacks the intuitive understanding and more flexible computing restraints that CUDA offers. Thus, the question of why and how Jacket works so well for certain functions and not others, is answered in CUDA–and why knowledge of CUDA is necessary for understanding Jacket.

Jacket is composed of two types of functions: (1) Those specific to programming on a GPU, like running a for loop on the GPU, and (2) those for general scientific computing, such as trapezoidal integration. For functions under type (1), we find large speed up times when the number of specific datum that needs to be computed simultaneously is high. For type (2), there is speed up even for small arrays of data. As we turn now to the parallelization of the DSLNAP Method, we will find that the majority of our speed up times and also the majority of the CPU run time are in functions of type (2).

The utilization of GPUs as General Purpose GPUs requires that we also use the CPU in coordination with our GPU in the process called dynamic programming. Our algorithm will use both the host and the device, with both the host memory and device memory. How these four things interact is very important. For example, say that we wish to compute a small pool of data simultaneously on the GPU. This is achieved by using the CPU to send the data to the device memory and the instruction to the device. The GPU then runs the instruction simultaneously and stores the result in the device memory. Next, the data is sent back to the host memory via the CPU. Thus, if the amount of time necessary for the transfer of the data between the device and the host is larger than the actual speed up time of the GPU computation, then there is the possibility and the entire process will take longer than simply computing on the CPU alone.

Normally on a CPU in MATLAB, we test computation time by surrounding the code with *tic* and *toc*. If we were to do this with code for the GPU, then we would not be testing the time it takes data to be transferred from the device to the host. The Jacket Library includes the functions *geval* and *gsync*. To illustrate their use, the following come from the Jacket media wiki page [13].

```
% ensure GPU is ready to begin
gsync
tic

for i = 1:n
    a = a .* a;
end

% ensure computation is evaluated on CPU
geval(a)

% wait until GPU finishes executing
gsync
toc
```

Now we are ready to parallelize parts of the DSNLAP method for reduction in run time.

## 4. Parallelization of DSNLAP Method

Because the Jacket Library is composed of special functions that drastically speed up the run time, this thesis will present parts of the DSNLAP method using functions of the Jacket Library. However, it is futile to discuss the speed up time of the entire method for a few reasons. First, the method consists of two distinct parts: Training and execution. Training the algorithm on tens of thousands of random matrices produces our basic probability masses by running the algorithm and comparing the error in the results. For each of these random matrices, the entire DSNLAP method is run, meaning that we complete the Talbot method for the inversion. This is where the majority of the run time lies, in training. Contrasting this with the execution, where we receive only one matrix and use the previous training to invert from the Laplacian back to the time domain, we find that the method is completed only once. This separation calls for quite different demands on the device and also the host memory, and thus is a reason why considering the speed up of the entire method is superfluous.

4.1. **Training.** Training the algorithm consists of the method described in the first section of this paper, where we create six classes of input matrices. However, each of these is generated using the MATLAB *rand* function for the real and imaginary parts of the Laplacian. Because

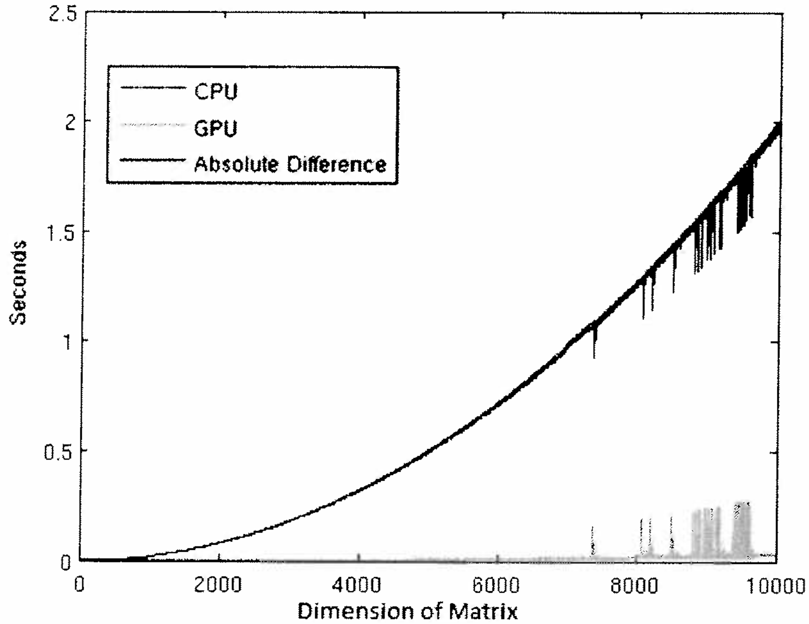


FIGURE 1. Comparison of run times for creating random numbers in a gfor loop.

the algorithm compares extracted feature values from all of the test matrices, we need to store all the matrices on the host server.

Figure 1 shows the drastic increase in run time for the following line:

```
gfor i = 1:10000
a=rand(i);
end
```

The training algorithm outfitted for performance on GPU follows:[1]

```
%NLAPDTRAIN The function is used to train the Dempster-Shafer theory for
%the selection Talbot's method contours when applied to the calculation
%of the matrix exponential. Due to rigorous bounds on the MATLAB expm
%functions estimate, it is taken as the 'truth' value. The matrices are
%limited to square matrices with complex elements.
%
%Output file format: The format of the output file for the training set is:
%Column: Description
%1 : Square matrix dimension
%2 : Maximum allowed real or imaginary element values
%3 : Trace
%4 : Determinant
%5 : TwoNorm
%6 : LogNorm
%7 : Contour 1 maximum absolute error
%8 : Contour 1 maximum relative error
%9 : Contour 2 maximum absolute error
%10 : Contour 2 maximum relative error
%11 : Contour 3 maximum absolute error
%12 : Contour 3 maximum relative error
```

[1](*) denotes a LaTeX formatting break

```
 %13 : Contour 4 maximum absolute error
 %14 : Contour 4 maximum relative error
 %15 : Contour 5 maximum absolute error
 %16 : Contour 5 maximum relative error
 %17 : Contour 6 maximum absolute error
 %18 : Contour 6 maximum relative error
 %
 %Talbot's method comment:
 %Talbot's method uses contours in the complex plane defined by
 % s(\theta) = \sigma + \lambda*\theta*(i\nu + \cot(\theta))
 %where:
 %\lambad, \nu, \sigma are real numbers
 %
 %\sigma = shift to the intercept position on the real axis of the contour
 % (\lambda + \sigma)
 %\nu = scaling factor for the intercept position on the imaginary axis of
 %the contour, \theta at the imaginary axis intercept is the solution to the
 %transcendental equation involving \sigma and \lambda
 %
 %\lambda = contour shape factor, the asymptotic value for the imaginary
 %part of the contour is \pm \lambda \pi \nu
 %
 %-\pi < \theta < \pi
 %
 %Authors: Patrick Kano and Jonathan Cain
 %
 %
 %
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 function NLapDStrain(NumTrainMats,MaxMatrixDim,MaxElementValue,RandomSeed,NumClasses,NumTheta)

 %%Parameters and the test matrix
 if(nargin==0)
  NumTrainMats    = 500;  %Number of random matrices to include in the training set
  MaxMatrixDim    = 25;    %Matrix dimension in the text set
  MaxElementValue = 25;     %Real and imaginary matrix elements have values from [-MaxElementValue,MaxElementValue]
  RandomSeed      = 1234;  %Random seed integer for the test matrices
  NumClasses      = 6;     %Number of contour classes
  NumTheta        = 2^10;  %Number of \theta points
 end

%%Random matrices selection
%Instead of generating and storing the test random matrices, the
%matrices are uniquely specified by setting the random seed.
rand('state',RandomSeed);

%%Open the output file
FileID = fopen('TrainPerformance.txt','w');

%%Preallocate GPU computation of random matrices
gfor DimIndex= 2:MaxMatrixDim
  M1(DimIndex)=rand(DimIndex,DimIndex);
  M2(DimIndex)=rand(DimIndex,DimIndex);
gend

%%Send back to CPU
M1=geval(M1)
M2=geval(M2)

%%Run the set of test matrices through the feature extraction and numerical
%inversion
for DimIndex = 2:MaxMatrixDim
```

```
for MatIndex=1:NumTrainMats
 disp(['Matrix: Dimension ',num2str(DimIndex),' Index = ',num2str(MatIndex)]);
 RawMatrix = 2*MaxElementValue*(rand(DimIndex,DimIndex) + i*rand(DimIndex,DimIndex))-MaxElementValue;

 %%Scale matrix as done by in MATLAB expmdemo1
 [fscale,escale] = log2(norm(RawMatrix,'inf'));
 sscale          = max(0,escale+1);
 InputMatrix     = RawMatrix/(2^sscale);

 ConjMatrix = InputMatrix';

 %%Feature extraction
 TraceTestFea = trace(InputMatrix); %det(exp(A)) = exp(tr(A))
 DetTestFea = det(InputMatrix);     %if zero, the one eigenvalue is zero
 TwoNormFea = norm(InputMatrix,2);  %spectral radius is bounded by the matrix 2-norm
 LogNormFea = norm((InputMatrix+ConjMatrix)/2); %the log norm is related to the condition number

 %%'Exact' matrix exponential from expm
 ExactExpm = expm(RawMatrix);

 %%Plot of the contour in the complex plane
 %figure(12345);
 %hold on;
 %clf;

 %%Scan through the classes of contours
 for ClassIdx=1:NumClasses
  switch ClassIdx
   case 1
    sigma  = 0;
    nu     = 1;
    lambda = TwoNormFea;
   case 2
    sigma  = 0;
    nu     = 1;
    lambda = real(TraceTestFea);
   case 3
    sigma  = real(TraceTestFea);
    nu     = real(TraceTestFea);
    lambda = 1;
   case 4
    sigma  = TwoNormFea;
    nu     = TwoNormFea;
    lambda = 1;
   case 5
    sigma  = real(TraceTestFea);
    nu     = 1;
    lambda = real(TraceTestFea);
   case 6
    sigma  = TwoNormFea;
    nu     = 1;
    lambda = TwoNormFea;
 end %switch statement

 %%Trapezoid rule for the integration on the selected contour
 ThetaVec = linspace(-pi,pi,NumTheta+2);
 ThetaVec = ThetaVec(2:end-1);

 %Vectorized
 %%Define the Talbot's method integrand
 %s-\sigma, L-\lambda, n-\nu, a-\theta
 %STalbot = inline('s + L*a*(i*n + cot(a))','s','n','L','a');
```

```
    %dsda = inline('i*L - L*(a + (a*cot(a) - 1)*cot(a)','L','a');

    STalbot      = sigma*ones(1,NumTheta) + lambda.*ThetaVec.*(i*nu*ones(1,NumTheta) + cot(ThetaVec));
    dsdtheta     = i*lambda*ones(1,NumTheta) - lambda*(ThetaVec + (ThetaVec.*cot(ThetaVec) - 1).*cot(ThetaVec));

    IntegrandVec = zeros(DimIndex,DimIndex,NumTheta);
    for ThetaIdx=1:NumTheta
     IntegrandVec(1:DimIndex,1:DimIndex,ThetaIdx) = exp(STalbot(ThetaIdx)).*inv(STalbot(ThetaIdx)*eye(DimIndex) -
 (*)InputMatrix).*dsdtheta(ThetaIdx); %e^{s}
    end %Theta vector

    %%Plot of the contour in the complex plane
    %figure(12345);
    %axis tight;
    %switch ClassIdx
    % case 1, plot(real(STalbot),imag(STalbot),'-bo');
    % case 2, plot(real(STalbot),imag(STalbot),'-rx');
    % case 3, plot(real(STalbot),imag(STalbot),'-gh');
    % case 4, plot(real(STalbot),imag(STalbot),'-c*');
    % case 5, plot(real(STalbot),imag(STalbot),'-kp');
    % case 6, plot(real(STalbot),imag(STalbot),'-m+');
    %end %switch statement

    %%Numerical integration on the complex contour
    ApproxExpm = trapz(ThetaVec,IntegrandVec,3)/(2*pi*i);

    %%Square back up to the exponential of the RawMatrix on GPU
    gfor SquareIdx = 1:sscale, ApproxExpm = ApproxExpm*ApproxExpm; geval(ApproxExpm); gend

    %%Relative and absolute error
    AbsErrorVec(ClassIdx) = norm(ApproxExpm-ExactExpm,'fro');
    RelErrorVec(ClassIdx) = AbsErrorVec(ClassIdx)/norm(ExactExpm,'fro');
   end %Classes for loop

   [RelErrorMinVal, RelErrorMinIdx] = min(RelErrorVec)

  end %Input matrix for loop
 end %Dimensions of the matrix index

%%Close the output file
fclose(FileID);
end %function definition
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Upon inspection, the majority of the run time will lie in the loop:

```
for ThetaIdx=1:NumTheta
   IntegrandVec(1:DimIndex,1:DimIndex,ThetaIdx) = exp(STalbot(ThetaIdx)).*
(*)inv(STalbot(ThetaIdx)*eye(DimIndex) -InputMatrix).*dsdtheta(ThetaIdx); %e^{s}
end %Theta vector
```

Although there is Jacket function for the /emphinv function, it is not gfor compatible. Our algorithm requires that we compute Talbot's method over the specified contour; however, this is done after two *for* loops that serve as the mechanism that allows one to transverse the frame of discernment. Because we must compute the inverse function in a *for* loop, there cannot by any GPU speed up because the *inv* function is not an example of a simple instruction. There is also merit in running the portion of our algorithm that integrates on the contour takes via the *trapz* function, which is gfor compatible. Figure 2 shows the increase in run time for trapezoidal integration:

```
gfor i = 1:10000
```

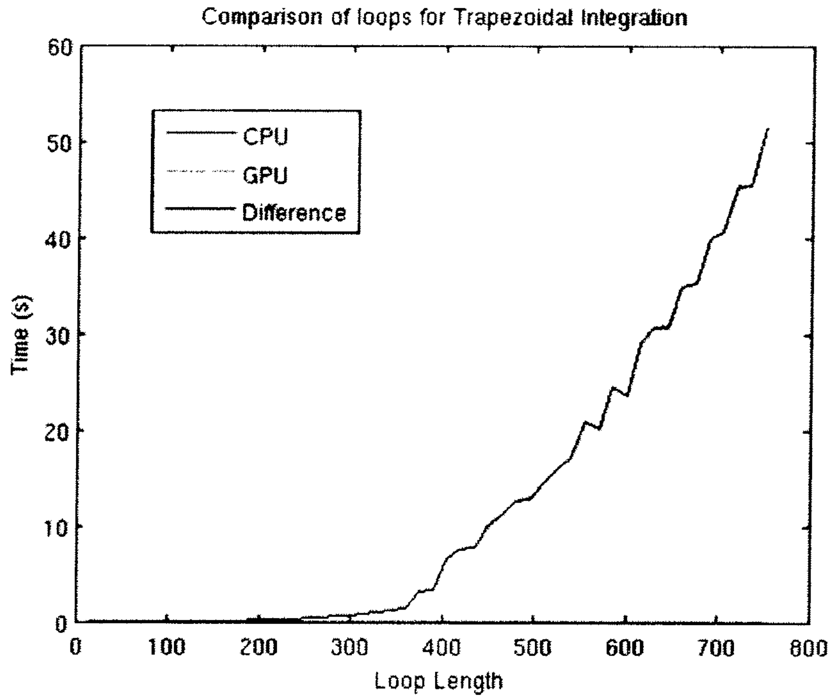


FIGURE 2. Comparison of Run Times for trapazodial integration in a gfor loop.

```
a=trapz(n);
end
```

We find that at a very high dimension, the speed up times will be drastic. Note, however, that the training algorithm does not use a *gfor* loop, which only shows the memory, not SIMD limitations and capabilities. There are also small *for* loops in the above code that are run on the GPU. However, the majority of the computation time for the training portion of the algorithm can only be improved by generating random matrices on the GPU.

4.2. **Execution.** Execution of the algorithm, given only one input matrix, is compatible for computation on the GPU, however, because there are not external loops that run through the frame of discernment while the algorithm is running the Talbot Method. Again, the majority of the computation time lies in the Talbot inversion. To parallelize this, we can run the *exp* before on the GPU, then compute the for loop. However, this yields relatively the same results as the majority of the run time lies in the *inv* function, even if the dimension of the input matrix is large.The source code for the execution is:

```
%NLAPDMAIN The function is used to perform Dempster-Shafer reasoning to
%select optimal contours for Talbot's method  when applied to the calculation
%of the matrix exponential.
%The allowed input matrices are limited to square matrices with complex elements.
%
%
%Talbot's method comment:
%Talbot's method uses contours in the complex plane defined by
% s(\theta) = \sigma + \lambda*\theta*(i\nu + \cot(\theta))
%where:
%\lambad, \nu, \sigma are real numbers
%
%\sigma = shift to the intercept position on the real axis of the contour
```

```
%          (\lambda + \sigma)
%\nu = scaling factor for the intercept position on the imaginary axis of
%the contour, \theta at the imaginary axis intercept is the solution to the
%transcendental equation involving \sigma and \lambda
%
%\lambda = contour shape factor, the asymptotic value for the imaginary
%part of the contour is \pm \lambda \pi \nu
%
%-\pi < \theta < \pi
%
%Authosr: Patrick O. Kano and Jonathan Cain
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function ApproxExpm = NLapDSmain(RawMatrix,NumTheta,ClassIdx,sigma,nu,lambda)

%%Parameters and the test matrix
if(nargin==1)
 NumTheta = 1024;
 DSswitch = 1;    %Use DS
 Classswitch = 1; %Use the user specified contour paramters
 sigma  = 0;
 nu     = 0;
 lambda = 0;
elseif(nargin==2) %Use DS
 DSswitch = 1;    %Use DS
 Classswitch = 1; %Use the user specified contour paramters
 sigma  = 0;
 nu     = 0;
 lambda = 0;
elseif(nargin==3)
 DSswitch = 0;    %Use the user specific contour class
 Classswitch = 1; %Use the user specified contour paramters
elseif(nargin==6)
 DSswitch = 0;    %Use the user specified contour parameters
 Classswitch = 0; %Use the user specified contour paramters
else
 error('NLapDSmain: Invalid number of inputs.');
end

%%Check that the input matrix is square.
[Nrows,Ncols] = size(RawMatrix);
if(Nrows~=Ncols), error('NLapDSmain: The input matrix must be square.'); end
MaxMatrixDim = Nrows;

%%Check that the number of theta points makes sense
if NumTheta<=0, error('NLapDSmain: The input number of theta poins must be positive.'); end

%%Set up the frame-of-discernment
NumClasses = 6;
NumSets = 2^6;

FrameDiscernBinary = zeros(1,6);

for nidx=1:NumClasses
 Tempbins = zeros(1,6);
 for sidx=1:nidx, Tempbins(sidx) = 1; end
 TempPerms = unique(perms(Tempbins),'rows');
 FrameDiscernBinary = [FrameDiscernBinary; TempPerms];
end

%%Scale matrix as done by in MATLAB expmdemo1
```

```
[fscale,escale] = log2(norm(RawMatrix,'inf'));
sscale          = max(0,escale+1);
InputMatrix     = RawMatrix/(2^sscale);

ConjMatrix = InputMatrix';

%%Feature extraction
TraceTestFea = trace(InputMatrix); %det(exp(A)) = exp(tr(A))
DetTestFea = det(InputMatrix);     %if zero, the one eigenvalue is zero
TwoNormFea = norm(InputMatrix,2);  %spectral radius is bounded by the matrix 2-norm
LogNormFea = norm((InputMatrix+ConjMatrix)/2); %the log norm is related to the condition number

if DSswitch==1

 %%Dempster-Shafer reasoning for the contours
 %Singleton-sets
 SingleMassA = MassFunA(TraceTestFea);
 SingleMassB = MassFunB(DetTestFea);
 SingleMassC = MassFunC(TwoNormFea);
 SingleMassD = MassFunD(LogNormFea);

 %%Assign masses to each set in the frame-of-discernment
 AssignMassA = zeros(1,NumSets);
 AssignMassB = zeros(1,NumSets);
 AssignMassC = zeros(1,NumSets);
 AssignMassD = zeros(1,NumSets);

 for sidx=2:NumSets
  NonzeroSum = sum(FrameDiscernBinary(sidx,1:NumClasses));

  AssignMassA(sidx) = sum(FrameDiscernBinary(sidx,1:NumClasses).*SingleMassA)/NonzeroSum;
  AssignMassB(sidx) = sum(FrameDiscernBinary(sidx,1:NumClasses).*SingleMassB)/NonzeroSum;
  AssignMassC(sidx) = sum(FrameDiscernBinary(sidx,1:NumClasses).*SingleMassC)/NonzeroSum;
  AssignMassD(sidx) = sum(FrameDiscernBinary(sidx,1:NumClasses).*SingleMassD)/NonzeroSum;
 end

 %Normalize the masses if nonzero (the total masses over all sets in frame <=1)
 TotalAssignMassA = sum(AssignMassA);
 TotalAssignMassB = sum(AssignMassB);
 TotalAssignMassC = sum(AssignMassC);
 TotalAssignMassD = sum(AssignMassD);

 if(0<TotalAssignMassA), AssignMassA = AssignMassA/TotalAssignMassA; end
 if(0<TotalAssignMassB), AssignMassB = AssignMassB/TotalAssignMassB; end
 if(0<TotalAssignMassC), AssignMassC = AssignMassC/TotalAssignMassC; end
 if(0<TotalAssignMassD), AssignMassD = AssignMassD/TotalAssignMassD; end

 %%Dempster-Shafer fucsion: Combine the massess for the different features
 %%for each set

 %Feature A
 AssignMassCombo = AssignMassA;

 %Feature A with Feature B mass fusion
 AssignMassCombo = FunDSfusion(AssignMassCombo,AssignMassB,FrameDiscernBinary,NumSets);

 %Feature C with Features AB combined mass fusion
 AssignMassCombo = FunDSfusion(AssignMassCombo,AssignMassC,FrameDiscernBinary,NumSets);

 %Feature D with Features ABC combined mass fusion
 AssignMassCombo = FunDSfusion(AssignMassCombo,AssignMassD,FrameDiscernBinary,NumSets);

 %%Belief calculation: A decision for the contour parameters based on the
```

```
 %%maxium belief
 BeliefVec = zeros(1,NumClasses);
 for cidx=1:NumClasses
  BeliefVec(cidx) = sum(FrameDiscernBinary(1:NumSets,cidx).*(AssignMassCombo(1:NumSets)'));
 end

 [BeliefMax,ClassIdx] = max(BeliefVec);

 disp(['Belief = ',num2str(BeliefVec)]);
 disp(['Selected Class = ',num2str(ClassIdx)]);

end %if DSswitch==1

if Classswitch==1 %Use ClassIdx to define contour parameters based on the contour class;
 %%Select the contour
 switch ClassIdx
  case 1
   sigma  = 0;
   nu     = 1;
   lambda = TwoNormFea;
  case 2
   sigma  = 0;
   nu     = 1;
   lambda = real(TraceTestFea);
  case 3
   sigma  = real(TraceTestFea);
   nu     = real(TraceTestFea);
   lambda = 1;
  case 4
   sigma  = TwoNormFea;
   nu     = TwoNormFea;
   lambda = 1;
  case 5
   sigma  = real(TraceTestFea);
   nu     = 1;
   lambda = real(TraceTestFea);
  case 6
   sigma  = TwoNormFea;
   nu     = 1;
   lambda = TwoNormFea;
 end %switch statement
end %if Classswitch == 1

%%Trapezoid rule for the integration on the selected contour
ThetaVec = linspace(-pi,pi,NumTheta+2);
ThetaVec = ThetaVec(2:end-1);

%Vectorized
%%Define the Talbot's method integrand
%s-\sigma, L-\lambda, n-\nu, a-\theta
%STalbot = inline('s + L*a*(i*n + cot(a))','s','n','L','a');
%dsda = inline('i*L - L*(a + (a*cot(a) - 1)*cot(a)','L','a');

STalbot      = sigma*ones(1,NumTheta) + lambda.*ThetaVec.*(i*nu*ones(1,NumTheta) + cot(ThetaVec));
dsdtheta     = i*lambda*ones(1,NumTheta) - lambda*(ThetaVec + (ThetaVec.*cot(ThetaVec) - 1).*cot(ThetaVec));

%%Plot of the contour in the complex plane
figure(12345);
plot(real(STalbot),imag(STalbot),'-ko');
axis tight;
xlabel('Real Axis');
ylabel('Imaginary Axis');
```

```
%%Numerical integration on the complex contour
IntegrandVec = zeros(MaxMatrixDim,MaxMatrixDim,NumTheta);
for ThetaIdx=1:NumTheta
 IntegrandVec(1:MaxMatrixDim,1:MaxMatrixDim,ThetaIdx) = exp(STalbot(ThetaIdx)).*inv(STalbot(ThetaIdx)*eye(MaxMatrixDim) -
(*)InputMatrix).*dsdtheta(ThetaIdx); %e^{s}
end %Theta vector

ApproxExpm = trapz(ThetaVec,IntegrandVec,3)/(2*pi*i);

%%Square back up to the exponential of the RawMatrix on the GPU
gfor SquareIdx = 1:sscale, ApproxExpm = ApproxExpm*ApproxExpm; geval(ApproxExpm); gend

end %function definition
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

4.3. **Conclusion.** There is merit in implementing algorithms on GPUs versus CPUs for complex numerical algorithms. However, there are occasions when the algorithm is not subject to the SIMD requirement for computation on a GPU. The Talbot inversion step is one such problem to the DSNLAP Method. What is required for this step is MIMD or MPI capabilities.

There are other places in the above listing of the DSNLAP Method that are parallelizable. One such area of more advanced study would be to create and traverse the frame of discernment using the fact that the frame of discernment is a binary matrix, which can be rendered on the GPU using the *glogical* function in Jacket. Other parts of the algorithm can also be put on the GPU that will receive good speed ups as the dimension gets large; however, the majority of the run time still lies on the *inv* function.

To achieve a speed up for practical implementation of this algorithm, the *inv* function would benefit greatly if rewritten into CUDA. Although the function itself is not an example of SIMD, there is still merit in trying to replace the *inv* function with the backslash operator. However, this will require significant changes to the algorithm itself since currently the input for the *inv* function is being indexed inside the *for* loop. The author is presently unaware of a way to implement such solution, but is hopeful that a GPU rendering can be found.

## References


[1] Cohen A. *Numerical Methods for Laplace Transform Inversion*, Springer Science+Business, LLC, pp, 121-139, 2007.

[2] Talbot, A. "The accurate numerical inversion of Laplace transform", *J. Inst. Maths. Applics.*, vol. 23, pp. 97-120, 1979.

[3] Murli, A. and M. Rizzardi. "Algorithm 682. Talbot's method for the Laplace inversion problem", *ACM Trans. Math. Soft.*, vol. 16, pp.158-168, 1990.

[4] Dempster, A. P. "Upper and lower probabilities induced by a multivalued mapping", *The Annals of Mathematical Statistics.* 38 (2): 325339, 1967.

[5] Shafer, G.*A Mathematical Theory of Evidence* Princeton University Press, 1976.

[6] Weeks, W.T. "Numerical inversion of the Laplace transform using Laguerre functions", *J. Assoc. Comput. Mach.*, vol. 13, pp. 419-429, 1966.

[7] Post, E. "Generalized Differentiation", *Trans. Amer. Math. Soc.*, vol. 32, p. 723-781, 1930.

[8] Klir, G. J. and M. J. Wierman. *Uncertainty-Based Information: Elements of Generalized Information Theory.* Heidelberg, Physica-Verlag, 1998.

[9] Kano, P. O., Brio, M., and Maloney, J. V. *Application of Weeks Method for the Numerical Inversion of the Laplace Transform to the Matrix Exponential.* Comm. Math. Sci., Vol. 3, No. 3, pp. 335-372, 2005.

[10] Kano, P. O., Brio, M., Dostert, P., and Cain, J. *Dempster-Shafer Evidential Theory for the Selection of Contours in Talbots Method for the Numerical Inversion of the Laplace Transform and Application to the Computation of the Matrix Exponential.* Submitted to Computer and Mathematics with Applications, February 2011.

[11] Flynn, M. (1972). "Some Computer Organizations and Their Effectiveness". IEEE Trans. Comput. C-21: 948.

[12] Sanders, J. and Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley, Boston, 2011.

[13] Accelereyes Jacket Documentation. www.accelereyes.com



*E-mail address*: `cain1@email.arizona.edu`