Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 465–470
ISBN 978-83-60810-22-4

Solving Linear Recurrences on Hybrid GPU Accelerated Manycore Systems

Przemysław Stpiczyński
Institute of Mathematics, Maria Curie-Skłodowska University, Lublin, Poland
Institute of Theoretical and Applied Informatics of the Polish Academy of Sciences, Gliwice, Poland
Email: przem@hektor.umcs.lublin.pl

***Abstract*—The aim of this paper is to show that linear recurrence systems with constant coefficients can be efficiently solved on hybrid GPU accelerated manycore systems with modern Fermi GPU cards. The main idea is to use the recently developed *divide-and-conquer* algorithm which can be expressed in terms of Level 2 and 3 BLAS operations. The results of experiments performed on hybrid system with Intel Core i7 and NVIDIA Tesla C2050 are also presented and discussed.**

I. Introduction

GRAPHICAL processing units (GPUs [10]) have recently been widely used for scientific computing due to their large number of parallel processors which can be exploited using the Compute Unified Device Architecture (CUDA) programming language [9]. GPUs offer very high performance at low costs for data-parallel computational tasks. Thus it is a good idea to develop algorithms for hybrid (heterogeneous) computer architectures where large parallelizable tasks are scheduled for execution on GPUs, while small non-parallelizable tasks should be run on CPUs [25]. In 2010 NVIDIA introduced a new GPU (CUDA) architecture with an internal name of Fermi [14]. The new architecture offers new features and much better performance than older CUDA devices when computations are carried out in double precision.

Linear recurrence systems with constant coefficients are central parts of many numerical algorithms for solving important problems like evaluation of orthogonal polynomials [2], symmetric tridiagonal eigenvalue problem [17], trigonometric interpolation [17], [18], numerical inverse of the Laplace Transform [8], [22] or general polynomial evaluation [19]. Also, they are commonly used in signal processing [16], [24]. There are several algorithms for solving linear recurrences on parallel computers [2], [3], [5], [6], [15], [26]. They are usually devoted to the problem of solving first or second order recurrence systems. In [21] we have proposed a new *divide-and-conquer* approach for solving m-th order linear recurrences with constant coefficients which can fully utilize the underlying hardware of modern computer systems (i.e. memory hierarchies and multiple processors). Our algorithm based on this approach achieves excellent speedup on various parallel computers [20], [23], [24].

It is clear that there is a real need to implement efficient solvers for linear recurrences and related problems on modern GPU accelerated manycore systems. The first attempt has been made by Nistor et al. [11]. They present a compile-time optimization technique for minimizing memory usage in the parallel computation of linear recurrences on GPUs. The aim of this paper is to show that the use of recently developed *divide-and-conquer* algorithm which can be expressed in terms of Level 2 and 3 BLAS operations [1] is crucial for achieving reasonable performance for linear recurrence computations on GPU-based systems.

II. The *Divide & Conquer* Method

Let us consider the following problem. For given real numbers f_k, $1 \le k \le n$, and a_j, $1 \le j \le m$, where $n \gg m$, find n numbers x_k, $1 \le k \le n$, such that

$$x_k = \begin{cases} 0 & \text{for } k \le 0 \\ f_k + \sum_{j=1}^{m} a_j x_{k-j} & \text{for } 1 \le k \le n. \end{cases} \tag{1}$$

It is clear that a simple algorithm based on (1), which requires $2mn$ flops, cannot utilize the underlying hardware, i.e. memory hierarchies, vector extensions and multiple processors, what is essential in case of modern manycore computer architectures. Thus, let us consider the following *divide-and-conquer* approach which allows to describe the process of solving (1) in terms of matrix operations [21]. First, let us choose positive integers r and s such that $rs \le n$ and $s > m$. It is clear that the numbers $x_1, \ldots, x_{rs}$ satisfy the following block system of linear equations

$$\begin{bmatrix} L & & & \\ U & L & & \\ & \ddots & \ddots & \\ & & U & L \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_r \end{bmatrix}, \tag{2}$$

where the vectors $\mathbf{x}_j$ and $\mathbf{z}_j$ are given by

$$\mathbf{x}_j = [x_{(j-1)s+1}, \ldots, x_{js}]^T \in \mathbb{R}^s, \tag{3}$$

and

$$\mathbf{f}_j = [f_{(j-1)s+1}, \ldots, f_{js}]^T \in \mathbb{R}^s. \tag{4}$$

The matrices L and U are defined as follows

$$L = \begin{bmatrix} 1 & & & & & \\ -a_1 & \ddots & & & & \\ \vdots & \ddots & \ddots & & & \\ -a_m & & \ddots & \ddots & & \\ & \ddots & & \ddots & \ddots & \\ & & -a_m & \cdots & -a_1 & 1 \end{bmatrix} \in \mathbb{R}^{s\times s}$$

and

$$U = \begin{bmatrix} & -a_m & \cdots & -a_1 \\ & & \ddots & \vdots \\ & & & -a_m \\ 0 & & & \end{bmatrix} \in \mathbb{R}^{s\times s}.$$

When we consider the special structure of the matrix U, we can rewrite the solution to (2) as

$$\begin{cases} \mathbf{x}_1 = L^{-1}\mathbf{f}_1 \\ \mathbf{x}_j = L^{-1}\mathbf{f}_j + \sum_{k=1}^{m} \alpha_j^k \mathbf{y}_k \quad \text{for } j = 2,\ldots,r, \end{cases} \tag{5}$$

where $\mathbf{e}_k$ denotes k-th unit vector of $\mathbb{R}^s$ satisfying $L\mathbf{y}_k = \mathbf{e}_k$, and coefficients α_j^k are as follows $\alpha_j^k = \sum_{l=k}^{m} a_{m+k-l} x_{(j-1)s-m+l}$. Note that to compute vectors $\mathbf{y}_k$, we need to find the solution of the system $L\mathbf{y}_1 = \mathbf{e}_1$, namely $\mathbf{y}_1 = [1, y_2, \ldots, y_s]^T$. Then we can form vectors $\mathbf{y}_k$ using

$$\mathbf{y}_k = [\underbrace{0,\ldots,0}_{k-1}, 1, y_2, \ldots, y_{s-k+1}]^T. \tag{6}$$

To develop the BLAS-based method, let us observe that instead of finding vectors $\mathbf{x}_j$ separately, we can find the matrix

$$X = [\mathbf{x}_1, \ldots, \mathbf{x}_r] \in \mathbb{R}^{s\times r}. \tag{7}$$

Let us define the following upper triangular matrix

$$C = \begin{bmatrix} a_m & a_{m-1} & \cdots & a_1 \\ 0 & a_m & \cdots & a_2 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & a_m \end{bmatrix} \in \mathbb{R}^{m\times m}. \tag{8}$$

During the first step of the algorithm we find $\mathbf{y}_1$ and all vectors $\mathbf{z}_j$. Obviously, it is equivalent to the problem of solving the block system of linear equations $LZ = F$, where $Z = [\mathbf{z}_1, \ldots, \mathbf{z}_r, \mathbf{y}_1]$ and $F = [\mathbf{f}_1, \ldots, \mathbf{f}_r, \mathbf{e}_1]$ are $s \times (r+1)$ matrices. We can find Z row by row using

$$Z_{k,*} = \begin{cases} \mathbf{0} & \text{for } k \le 0 \\ F_{k,*} + \sum_{j=1}^{m} a_j Z_{k-j,*} & \text{for } 1 \le k \le s. \end{cases} \tag{9}$$

Thus, after the sequence of operations

$$Z_{k,*} \longleftarrow F_{k,*} + C_{1,\max\{1,m-k+2\}:m} Z_{\max\{1,k-m\}:k-1,*} \tag{10}$$

performed for $k = 2,\ldots,s$, we have $Z = [\mathbf{z}_1, \ldots, \mathbf{z}_r, \mathbf{y}_1]$. Note that $Z_{*,1} = X_{*,1} = [x_1, \ldots x_s]^T$. During the second step of the algorithm we compute m last entries of all vectors $\mathbf{x}_2, \ldots, \mathbf{x}_r$ (i.e. $X_{s-m+1:s,2:r}$), and form the matrix

$$A = \begin{bmatrix} \alpha_2^1 & \cdots & \alpha_r^1 \\ \vdots & & \vdots \\ \alpha_2^m & \cdots & \alpha_r^m \end{bmatrix} \in \mathbb{R}^{m\times(r-1)}. \tag{11}$$

When we define $Y = [\mathbf{y}_1, \ldots, \mathbf{y}_m] \in \mathbb{R}^{s\times m}$, and then we can complete this step after the following sequence of operations

$$\begin{cases} A_{*,j-1} \longleftarrow CX_{s-m+1:s,j-1} \\ X_{s-m+1:s,j} \longleftarrow Z_{s-m+1:s,j} + Y_{s-m+1:s,*} A_{*,j-1} \end{cases} \tag{12}$$

performed sequentially for $j = 2,\ldots,r$. During the third step of the algorithm, we have to calculate $s-m$ first entries of all vectors $\mathbf{x}_2, \ldots, \mathbf{x}_r$ (i.e. $X_{1:s-m,2:r}$). It can be done using a single call to the Level 3 BLAS routine xGEMM (matrix-matrix multiplication):

$$X_{1:s-m,2:r} \longleftarrow Z_{1:s-m,2:r} + Y_{1:s-m,*} A. \tag{13}$$

The *divide-and-conquer* method can be summarized as Algorithm 1. It comprises three main steps: **Step1** (lines 3–5), **Step2** (lines 7–10) and **Step3** (line 11). There is no need to store the matrix Z. If we assume that the solution X can be stored in place of the matrix F, then **Step1** can be easily formulated in terms of the matrix-vector multiplications (the operation xGEMV from Level 2 BLAS), **Step2** is a sequence of two calls to the routines from Level 2 BLAS, namely xTRMV ("triangular matrix"-vector multiplication) and xGEMV. Finally, **Step3** is a single call to the routine xGEMM. Note that **Step1** and **Step3** can be easily parallelized (see Algorithm 2) using OpenMP [4].

Algorithm 1 BLAS-based algorithm for (1)

Require: given $a_1, \ldots, a_m$ and $\mathbf{f}_1, \ldots, \mathbf{f}_r$ given by (4), where $r > 1$, $s > 1$ and $rs = n$
Ensure: $X_{1:s,1:r} = [\mathbf{x}_1, \ldots \mathbf{x}_r]$ given by (7)
1: $X \leftarrow [\mathbf{f}_1, \ldots, \mathbf{f}_r, \mathbf{e}_1]$
2: $C \leftarrow \begin{bmatrix} a_m & \cdots & a_1 \\ & \ddots & \vdots \\ 0 & & a_m \end{bmatrix}$
3: **for** $k = 2$ to s **do**
4: $\quad X_{k,*} \leftarrow X_{k,*} + C_{1,\max\{1,m-k+2\}:m} X_{\max\{1,k-m\}:k-1,*}$ {xGEMV}
5: **end for**{ $X_{*,r+1} = \mathbf{y}_1$ }
6: $Y \leftarrow [\mathbf{y}_1, \ldots, \mathbf{y}_m]$
7: **for** $j = 2$ to r **do**
8: $\quad A_{*,j-1} \leftarrow CX_{s-m+1:s,j-1}$ {xTRMV}
9: $\quad X_{s-m+1:s,j} \leftarrow X_{s-m+1:s,j} + Y_{s-m+1:s,*} A_{*,j-1}$ {xGEMV}
10: **end for**
11: $X_{1:s-m,2:r} \leftarrow X_{1:s-m,2:r} + Y_{1:s-m,*} A$ {xGEMM}

The algorithm requires twice as many flops as the simple algorithm based on (1), namely $4mn$ [21]. It has been efficiently implemented on shared and distributed memory parallel

computers and clusters of workstations [20], [23] achieving reasonable performance. The performance of the algorithm (in Gflops) grows when the problem sizes (n, m) and the number of processors grow.

Algorithm 2 Parallel BLAS-based algorithm for (1)

Require: given $a_1, \ldots, a_m$ and $\mathbf{f}_1, \ldots, \mathbf{f}_r$ given by (4), where $r > 1$, $s > 1$, $rs = n$ and p is the number of parallel processors

Ensure: $X_{1:s,1:r} = (\mathbf{x}_1, \ldots \mathbf{x}_r)$.

1: $X \leftarrow (\mathbf{f}_1, \ldots, \mathbf{f}_r, \mathbf{e}_1)$

2: $C \leftarrow \begin{pmatrix} a_m & \cdots & a_1 \\ & \ddots & \vdots \\ 0 & & a_m \end{pmatrix}$

3: $t \leftarrow \lfloor (r+1)/p \rfloor$

4: **parallel for** $i = 0$ to $p-1$ **do**

5: $t_1 \leftarrow it + 1$; $t_2 \leftarrow (i+1)t$

6: **if** $i = p-1$ **then**

7: $t_2 \leftarrow r + 1$

8: **end if**

9: **for** $k = 2$ to s **do**

10: $X_{k,t_1:t_2} \leftarrow X_{k,t_1:t_2} + C_{1,\max\{1,m-k+2\}:m} X_{\max\{1,k-m\}:k-1,t_1:t_2}$ {xGEMV}

11: **end for**

12: **end parallel for**{ $X_{*,r+1} = \mathbf{y}_1$}

13: $Y \leftarrow (\mathbf{y}_1, \ldots, \mathbf{y}_m)$

14: **for** $j = 2$ to r **do**

15: $A_{*,j-1} \leftarrow C X_{s-m+1:s,j-1}$ {xTRMV}

16: $X_{s-m+1:s,j} \leftarrow X_{s-m+1:s,j} + Y_{s-m+1:s,*} A_{*,j-1}$ {xGEMV}

17: **end for**

18: $t \leftarrow \lfloor r/p \rfloor$

19: **parallel for** $i = 0$ to $p-1$ **do**

20: $t_1 \leftarrow it + 1$; $t_2 \leftarrow (i+1)t$

21: **if** $i = 0$ **then**

22: $t_1 \leftarrow t_1 + 1$

23: **end if**

24: **if** $i = p-1$ **then**

25: $t_2 \leftarrow r$

26: **end if**

27: $X_{1:s-m,t_1:t_2} \leftarrow X_{1:s-m,t_1:t_2} + Y_{1:s-m,*} A$ {xGEMM}

28: **end parallel for**

III. GPU Architecture and Implementation

The detailed description of NVIDIA CUDA and Fermi architectures can be found in [14] and [13]. A *computing device* (usually GPU) comprises a number of streaming multiprocessors (SM). Each SM consists of 32 (8 for older CUDA devices) scalar cores (streaming processors, SP). SMs are responsible for executing blocks of threads. Threads within a block are grouped into so-called *warps*, each consisting of 32 threads managed and executed together.

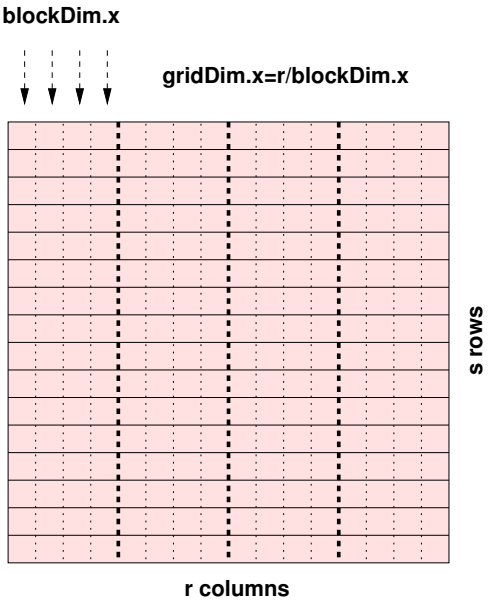

Fig. 1. Storage for the *divide and conquer* algorithm on GPUs

A device has its own memory system including the *global memory* (large but low latency), *constant* and *texture* read-only memories providing reduction of memory latency. Each SM has also a 48 kB (16 kB for older CUDA devices) of fast *shared memory* that can be used for sharing data among threads within a block. The global memory access can be improved by coalesced access of all threads in a half-warp. Threads must access either 4-byte words in one 64-byte memory transaction, or 8-byte words in one 128-byte memory transaction. All 16 words must lie in the same memory segment [13].

Fermi (CUDA) programs consist of a number of C functions called *kernels* that are to be executed on devices as threads. Kernels are called from programs executed on CPUs. Host programs are also responsible for allocation of variables in the device global memory. There are also some CUDA API-functions used to copy data between host and device global memories.

The basic idea of the implementation of the *divide-and-conquer* algorithm on a hybrid CPU + GPU system is to run large parallel tasks (**Step1** and **Step3**) on GPU, while small sequential tasks (namely finding the vector $\mathbf{y}_1$ and **Step2**) are executed on CPU. To allow coalesced memory access, elements of the $s \times r$ array X should be stored row-wise in the global memory of a device. Thus we cannot use CUBLAS (i.e. the implementation of the BLAS for CUDA [12]) because routines from that library assume column-wise storage of matrices. Each thread is responsible for computing one column of the array. The ID of its own column can be computed using `idx=blockIdx.x*blockDim.x+threadIdx.x`. Each block of threads is responsible for computing one *panel* – a group of adjacent columns (Figure 1). For simplicity, we assume that $r = (\#blocks) \times (\#threads\ in\ block)$.

The numbers a_j, $1 \le j \le m$, are stored in the constant

```
__global__ void step1(int m,int s,int r,
                              double *x){
   __shared__ double xc[16][MAXBSIZE];
   int idx=blockDim.x*blockIdx.x+
                              threadIdx.x;
   int myIdx=threadIdx.x;

   xc[0][myIdx]=x[idx];
   double t;

   // rows 0..m-1
   for(int i=1;i<m;i++){
     t=x[i*r+idx]; // read from global mem.
     for(int j=1;j<=i;j++)
        t+=xc[i-j][myIdx]*a_d[j-1];
     x[i*r+idx]=t; //write from global mem.
     xc[i][myIdx]=t;
   }

   // rows m..s-1
   for(int i=m;i<s;i++){
     t=x[i*r+idx];
     for(int k=1;k<=m;k++)
        t+=xc[m-k][myIdx]*a_d[k-1];
     for(int k=0;k<m-1;k++)
       xc[k][myIdx]=xc[k+1][myIdx];
     xc[m-1][myIdx]=t;
     x[i*r+idx]=t; //write to global mem.
   }
}
```

Fig. 2. Kernel for **Step1**

```
__global__ void step3(int m,int s,int r,
                  double *x, double *y){

   __shared__ double xc[16][MAXBSIZE];
   int idx=blockDim.x*blockIdx.x+
                           threadIdx.x;
   int myIdx=threadIdx.x;

   for(int k=0;k<m;k++)
     xc[k][myIdx]=x[(s-m+k)*r+idx];

   if(idx==0){
      for(int k=0;k<m;k++)
        xc[k][0]=0;
   }
   y=y+m-1;

   for(int i=0;i<s-m;i++){
      double t=0.0;
      for(int k=m-1;k>=0;k--)
        t+=xc[k][myIdx]*y[i-k];
      x[i*r+idx]+=t;
   }
}
```

Fig. 3. Kernel for **Step3**

memory of GPU. The algorithm starts with the kernel `step1` (Figure 2) on GPU. To reduce the number of references to the global memory, we use shared memory to store m recently computed entries of the thread's column. At the same time CPU solves the system $L\mathbf{y}_1 = \mathbf{e}_1$. Next, m last rows of the array X are copied from the device memory and CPU computes (12). Then the vector $\mathbf{y}_1$ and m last rows of X are copied to the device memory. Finally, the kernel `step3` (Figure 3) is scheduled for execution on GPU.

IV. Results of Experiments

The performance results in this section use the hybrid computer system with Intel Core i7 950 Extreme Edition(3.06 GHz, 24GB RAM) and NVIDIA Tesla C2050 (448 cores, 3GB GDDR5 RAM), running under Linux with Intel `icc` (version 12.0) and NVIDIA `nvcc` (release 3.2) compilers and Intel MKL Library (version 10.3). We have tested the following algorithms:

- the simple sequential algorithm based on (1) executed on CPU,
- Algorithm 1 executed on CPU,
- the BLAS-based *divide-and-conquer* algorithm executed on GPU+CPU.

The performance of the simple sequential algorithm is very poor (up to 660 Mflops), thus the exemplary results shown in Figure 5 and 6 describe the performance of two considered implementations of the BLAS-based algorithm. They have been tested for various values of n from $n = 1024^2 = 1048576$ to $n = 16384^2 = 268435456$, $m = 1, 2, 4, 6, 8, 16$ and various values of r and s, where $rs = n$. It is clear that greater values of the parameter r let us keep a GPU busy, but in such a case the sequential part of the algorithm (namely **Step2**) is longer, thus the GPU+CPU implementation achieves the best performance for $r = s$. The CPU implementation achieves the best performance for $s = 4r$. The performance of the GPU+CPU implementation grows when both m and n grow, while the performance of the CPU implementation decreases when n grows. The use of GPU+CPU is profitable for $n \geq 4194304 = 2048^2$. The GPU+CPU implementation is up to 58 (single precision) and 47 (double precision) times faster than CPU for $m = 1$. For greater values of m, GPU+CPU also outperforms CPU. One can observed that the speedup decreases with growing of m, because on CPU, f or greater values of m, the BLAS routines xGEMV and xGEMM are much more efficient. Finally it should be noticed that our GPU+CPU implementation achieves much better speedup than the algorithm presented in [11].

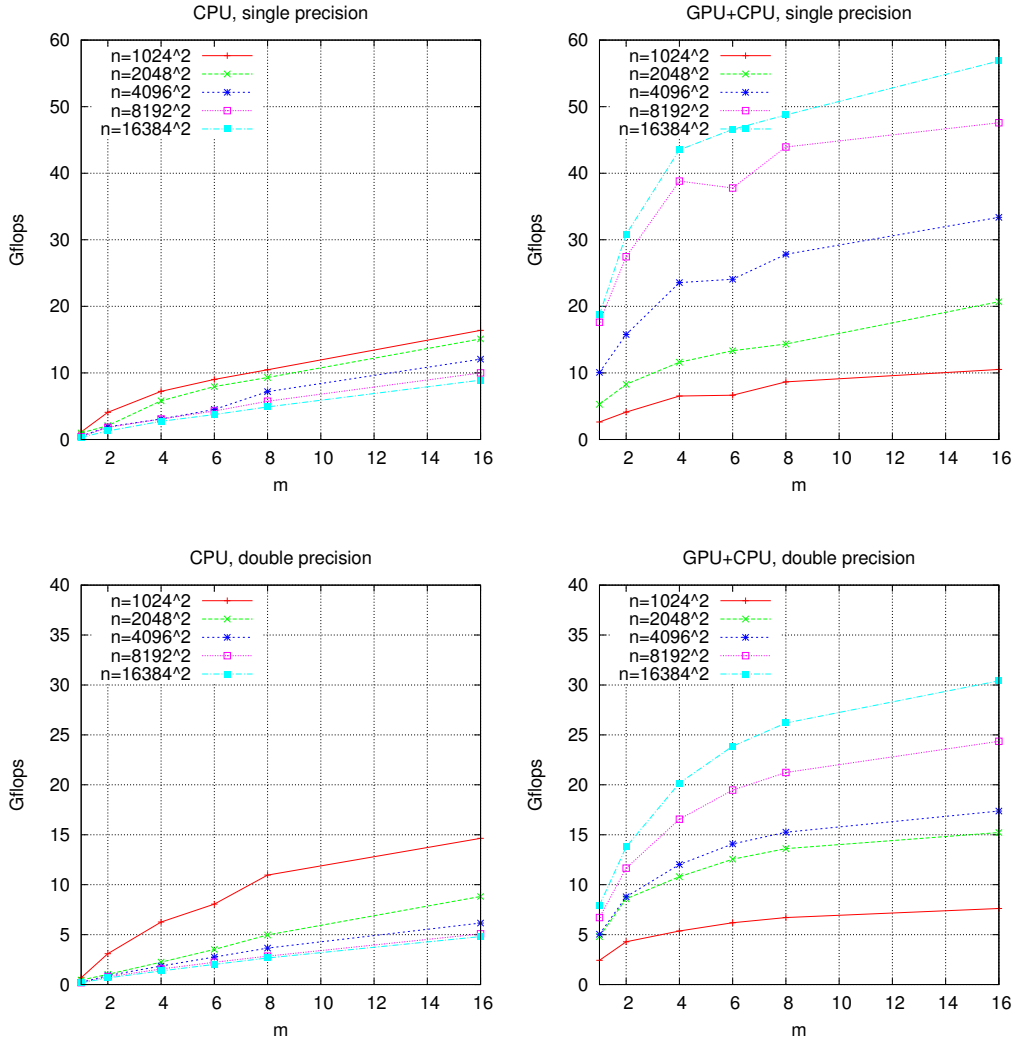

Fig. 5. Performance of the BLAS-based CPU implementation and the hybrid GPU+CPU implementation for single and double precision

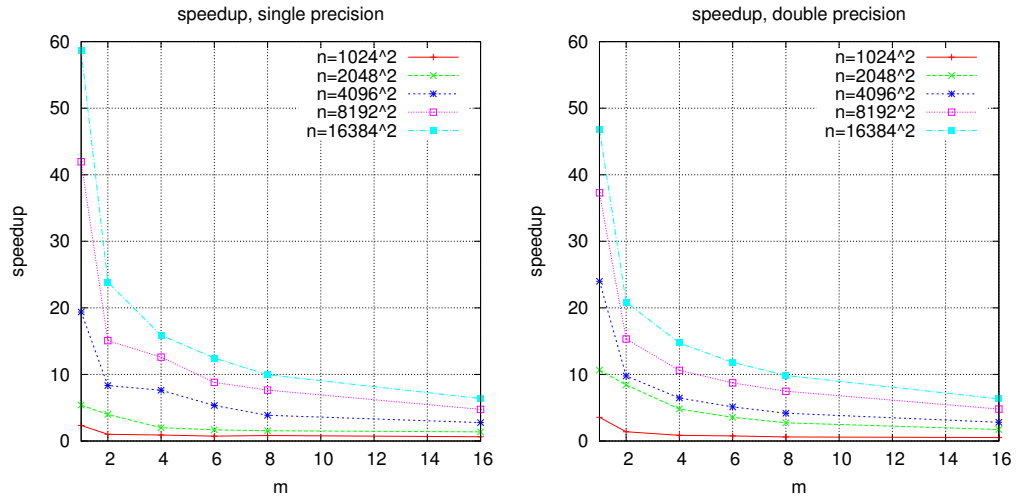

Fig. 6. Speedup of the hybrid GPU+CPU implementation over the BLAS-based CPU implementation

```
int bsize=256;

step1<<<r/bsize,bsize>>>(m,s,r,x_d);

rec1y(m,s,a_h,y_h+m-1);
for(int k=0;k<m-1;k++){
  y_h[k]=0;
}

cudaMemcpy(t_h, &x_d[(s-m)*r],
           r*m*sizeof(*t_h),
       cudaMemcpyDeviceToHost);

for(int j=1;j<r;j++){   // Step 2
  for(int i=0;i<m;i++){
    int ig=s-m+i;
    for(int k=0;k<m-i;k++)
      x_h[ig*r+j]+=
      a_h[m-k]*x_h[(s-m+k)*r+j-1];
  }
}

cudaMemcpy(&x_d[(s-m)*r],
           t_h,s*sizeof(*t_h),
       cudaMemcpyHostToDevice);
cudaMemcpy(y_d, y_h,
           (m-1+s)*sizeof(*y_d),
       cudaMemcpyHostToDevice);
step3<<<r/bsize,bsize>>>(m,s,r,x_d,y_d);

cudaThreadSynchronize();
```

Fig. 4. Host program

V. CONCLUSIONS

We have shown that linear recurrence systems with constant coefficients can be efficiently solved on hybrid GPU accelerated manycore systems with modern Fermi GPU cards using the *divide and conquer* BLAS-based algorithm. The reasonable performance can be achieved by using some optimization techniques for GPUs like coalesced global memory access and the use of shared memory for caching elements of global memory arrays during computations. It should be noticed the algorithm can be easily implemented for CPU + multiple GPU systems using OpenCL [7].

ACKNOWLEDGEMENTS

The author would like to thank the anonymous referees for valuable discussions and suggestions.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostruchov, and D. Sorensen, *LAPACK User's Guide*. Philadelphia: SIAM, 1992.
[2] R. Bario, B. Melendo, and S. Serrano, "On the numerical evaluation of linear recurrences," *J. Comput. Appl. Math.*, vol. 150, pp. 71–86, 2003.
[3] G. Blelloch, S. Chatterjee, and M. Zagha, "Solving linear recurrences with loop raking," *Journal of Parallel and Distributed Computing*, vol. 25, pp. 91–97, 1995.
[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Francisco: Morgan Kaufmann Publishers, 2001.
[5] H. Hafner and W. Shonauer, "Invastigation of different algorithms for the first order recurrence," *Supercomputer*, vol. 40, pp. 34–41, 1990.
[6] J.-L. Larriba-Pey, J. J. Navarro, A. Jorba, and O. Roig, "Review of general and Toeplitz vector bidiagonal solvers." *Parallel Comput.*, vol. 22, no. 8, pp. 1091–1125, 1996.
[7] A. Munshi, *The OpenCL Specification v. 1.0*. Khronos OpenCL Working Group, 2009.
[8] A. Murli and M. Rizzardi, "Algorithm 682: Talbot's method for the Laplace inversion problem," *ACM Trans. Math. Soft.*, vol. 16, pp. 158–168, 1990.
[9] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, pp. 40–53, 2008.
[10] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, pp. 56–69, 2010.
[11] A. Nistor, W.-N. Chin, T.-S. Tan, and N. Tapus, "Optimizing the parallel computation of linear recurrences using compact matrix representations," *J. Parallel Distrib. Comput.*, vol. 69, pp. 373–381, 2009.
[12] NVIDIA Corporation, *CUBLAS Library*. NVIDIA Corporation, 2009, available at http://www.nvidia.com/.
[13] ——, *CUDA Programming Guide*. NVIDIA Corporation, 2009, available at http://www.nvidia.com/.
[14] ——, "NVIDIA next generation CUDA compute architecture: Fermi," http://www.nvidia.com/, 2009.
[15] K. Shimizu and Y. Kanada, "Solving linear recurrence problems on supercomputers," *Supercomputer*, vol. 8, no. 1, pp. 30–37, Jan. 1991.
[16] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. San Diego, CA: California Technical Publishing, 1997.
[17] J. Stoer and R. Bulirsh, *Introduction to Numerical Analysis*, 2nd ed. New York: Springer, 1993.
[18] P. Stpiczyński, "Fast parallel algorithms for computing trigonometric sums," in *Proceedings of PARELEC 2002, International Conference on Parallel Computing in Electrical Engineering*, M. Tudruj and A. Jordan, Eds. IEEE Computer Society Press, 2002, pp. 299–304. [Online]. Available: http://computer.org/proceedings/parelec/1730/17300299abs.htm
[19] ——, "Fast parallel algorithm for polynomial evaluation," *Parallel Algorithms and Applications*, vol. 18, no. 4, pp. 209–216, 2003.
[20] ——, "Numerical evaluation of linear recurrences on various parallel computers," in *Proceedings of Aplimat 2004, 3rd International Conference, Bratislava, Slovakia, February 4–6, 2004*, M. Kovacova, Ed. Technical Univerity of Bratislava, 2004, pp. 889–894.
[21] ——, "Solving linear recurrence systems using level 2 and 3 BLAS routines," *Lecture Notes in Computer Science*, vol. 3019, pp. 1059–1066, 2004.
[22] ——, "A note on the numerical inversion of the laplace transform," *Lecture Notes in Computer Science*, vol. 3911, pp. 551–558, 2006.
[23] ——, "Numerical evaluation of linear recurrences on high performance computers and clusters of workstations," in *Proceedings of PARELEC 2004, International Conference on Parallel Computing in Electrical Engineering*. IEEE Computer Society Press, 2004, pp. 200–205.
[24] ——, "Evaluating recursive filters on distributed memory parallel computers," *Comm. Numer. Meth. Engng.*, vol. 22, pp. 1087–1095, 2006.
[25] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, pp. 232–240, 2010.
[26] H. van der Vorst and K. Dekker, "Vectorization of linear recurrence relations." *SIAM J. Sci. Stat. Comput.*, vol. 10, no. 1, pp. 27–35, 1989.