GPU based Sparse Grid Technique for Solving Multidimensional Options Pricing PDEs

Abhijeet Gaikwad
École Centrale Paris
Laboratoire MAS
Châtenay-Malabry, France
abhijeet.gaikwad@ecp.fr

Ioane Muni Toke
École Centrale Paris
Laboratoire MAS
Châtenay-Malabry, France
ioane.muni-toke@ecp.fr

ABSTRACT

It has been shown that the sparse grid combination technique can be a practical tool to solve high dimensional PDEs arising in multidimensional option pricing problems in finance. Hierarchical approximation of these problems leads to linear systems that are smaller in size compared to those arising from standard finite element or finite difference discretizations. However, these systems are still excessively demanding in terms of memory for direct methods and challenging to solve by iterative methods. In this paper we address iterative solutions via preconditioned Krylov subspace based methods, such as Stabilized BiConjugate Gradient (BiCGStab) and CG Squared (CGS), with the main focus on the design of such iterative solvers to harness massive parallelism of general purpose Graphics Processing Units (GPGPU)s. We discuss data structures and efficient implementation of iterative solvers. We also present a number of performance results to demonstrate the scalability of these solvers on the NVIDIA's CUDA platform.

Categories and Subject Descriptors

G.1 [**Numerical Analysis**]: Numerical Linear Algebra; I.3.1 [**Hardware Architecture**]: Graphics Processors

General Terms

Sparse Linear Iterative Solvers, Stabilized Biconjugate Gradient, Conjugate Gradient Squared

Keywords

NVIDIA CUDA, Iterative solvers, multidimensional option pricing

1. INTRODUCTION

Various problems in computational finance are formulated as high dimensional integrals stemming from large number of state variables and complex time structures. One major challenge is the pricing of high dimensional options, called basket or index options, with a large number of underlying risk factors. In the simplest multidimensional Black-Scholes model, the number of assets determines the dimensionality of underlying partial differential equations (PDEs).

Research on efficient pricing methods of these financial derivatives is one of the important areas of computational finance. On the one hand, the probabilistic formulation of the pricing problem easily translates into a Monte Carlo algorithm. Monte Carlo methods are flexible and therefore widely used for multidimensional pricing but they suffer from several drawbacks such as a relatively slow convergence and difficulty to compute accurate sensitivities of the solution (known as "Greeks" in finance). On the other hand, the direct solving of the underlying PDE offers fast convergence and easy computation of the sensitivities, but the method is often prohibitively computationally demanding and suffers from the *curse of dimensionality*: standard discretization of the PDE leads to systems that grow exponentially with the dimension of the problem.

The sparse grid combination technique can be used to control the exponential growth of unknowns in time dependent solutions [3]. In a nutshell, the method discretizes the problem on several sparse grids, then solves these sub-problems which have the same spatial dimensionality as the original problem but coarser discretization, and finally properly combines the partial solutions to get the final one. In the context of this work, we use the technique proposed in [3], which is briefly described in Section 2.

Computational efficiency of the sparse grid combination technique depends on the efficient solution of the resulting sub-problems. The linear system solvers can account for a large part of the overall computation time. In this paper we investigate through numerical experiments performance of Krylov subspace based iterative solvers, with which time and memory required per iteration do not increase and no restarting is needed as is the case with GMRes solvers[15]. Iterative methods have additional advantage that they do not change the structure of the problem. In this work, we focus on two solvers, BiCGStab and CGS. For regular or irregular sparse linear systems, the efficient implementation of solvers on parallel architectures becomes harder. Over the past three decades, a number of research efforts have resulted in parallel sparse linear solvers optimized for latest computational architectures. Our goal is to complement these efforts by developing a comprehensive sparse linear solver package for GPUs. Recently, GPGPUs have been used in various numerically intensive scientific applications

WHPCF ,November 15, 2009 Portland, Oregon, USA

due to their superiority over conventional CPUs with respect to achievable computational power and memory bandwidth. For instance, an Intel QuadCore Xeron 5140 CPU has a peak performance of 29 GFlops, whereas NVIDIA GPU Tesla C1060, has a peak performance of 933 GFlops[25]. Hardware vendors have also provided computational scientists with high level programming tools, like Common Unified Device Architecture (CUDA) from NVIDIA and Stream SDK, a precursor of Close-To-Metal from AMD-ATI, for hiding low level or direct access to GPUs, exposing as massively parallel data parallel processors. Compared to GPUs, CPUs are more flexible and can support a wider range of applications at the cost of greatly increased chip complexity. Specifically, programs that require complicated control flows and large data caches to achieve optimal performance are better suited for CPU-based implementations. On the other hand, contemporary GPUs have a significantly larger number of cores and devote a higher percentage of their transistors to floating point operations. Therefore, GPU provides massive parallelism and delivers better performance than CPU for certain applications. The sparse grid computation is an example of this kind, as the computation kernels are local and linear. Demonstrating the effectiveness of linear solvers on the CUDA platform will help us to establish the usefulness of this platform for various financial problems such as large scale portfolio optimization, asset management, etc.

The structure of the remainder of this paper is as follows. We first briefly discuss the combination technique in Section 2. In Section 3, we describe the NVIDIA GPU architecture. We provide a review of works that are of interest with regard to this paper in Section 4. The efficient parallel implementation details, including sparse matrix storage formats and the CUDA based matrix-vector multiplication (SpMV) libraries, are discussed in Section 5. We present numerical and performance results in Section 6. Section 7 contains a summery of our findings, and future directions for investigation.

2. COMBINATION TECHNIQUE FOR SOLVING THE BLACK-SCHOLES PDE

In this section, we briefly introduce the test case chosen for our GPU implementation of the combination technique. We choose to price European derivatives on a d-dimensional basket of risky assets in the multidimensional Black-Scholes framework.

Let us denote $S_i(t)$ the price of the i-th asset, $\sigma_i(t)$ its volatility, $\rho_{ij}(t)$ the linear correlation between assets i and j, and r the risk-free interest rate. We consider a European option with maturity T and payoff $h(\mathbf{S})$. The value $V(\mathbf{S},t), \mathbf{S} \in \mathbb{R}_+^d, t \in [0,T]$ of this option is solution of the following Black-Scholes partial differential equation:

$$\begin{aligned} \frac{\partial V}{\partial t} &= -\tfrac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}\rho_{ij}\sigma_i\sigma_j S_i S_j \tfrac{\partial^2 V}{\partial S_i \partial S_j} \\ &\quad -r\sum_{i=1}^{d} S_i \frac{\partial V}{\partial S_i} + rV, \end{aligned} \tag{1}$$

with terminal condition $V(\mathbf{S},T) = h(\mathbf{S})$. With constant parameters, this equation can be reduced to a multidimensional heat equation. Although very academic, this test case is similar to the PDEs of more realistic models. See [8] for a large panel of PDE in finance.

As mentioned earlier, solving PDE (1) in a high-dimensional case (say $d > 3$) is limited by memory restriction since the problem is affected by the so-called "curse of dimensionality": on a regular full grid of mesh size 2^{-n}, the discretized problem grows exponentially with the dimension. The sparse grid combination technique allows to solve a higher dimensional problem by reducing the size of the problem. The framework in our case is as follows. For simplicity, we assume our derivative contract is a basket up-and-out barrier option. In this case, the value of the option is zero as soon as one of the assets crosses a given value B, so that the PDE (1) can be solved with homogeneous Dirichlet boundary condition. Let us denote $\Omega = [0,B]^d$ as our domain. For each multi-index $\mathbf{i} \in \mathbb{N}^d$, we denote $\Omega_\mathbf{i}$ the Cartesian sparse grid on that domain with mesh size 2^{-i_k} in the k-th dimension. The standard finite differences solution of PDE (1) on the anisotropic grid $\Omega_\mathbf{i}$ is denoted $\tilde{V}_\mathbf{i}$. Then the solution V_n^{CT} obtained at level n with the sparse grid combination technique is:

$$V_n^{CT} = \sum_{l=n}^{n+d-1} (-1)^{d-1-l+n} \binom{d-1}{l-n} \sum_{i_1+\ldots+i_d=l} \tilde{V}_\mathbf{i} \tag{2}$$

Therefore, the sparse grid combination technique allows to build a discretization of the problem of size $\mathcal{O}(2^n n^{d-1})$ (on level n) instead of size $\mathcal{O}(2^{nd})$ in the full grid case. It also offers a natural coarse parallelization, since all solutions $V_\mathbf{i}$ summed in equation (2) can be computed independently from one another. All sub-grids $\Omega_\mathbf{i}$ are solved sequentially a *single* GPU using BiCGStab and CGS solver. Therefore, the speedups we present in Section 6 could be enhanced with a multiple GPU implementation using this "natural" parallelization.

3. NVIDIA GPU ARCHITECTURE AND THE CUDA PROGRAMMING MODEL

In this section, we discuss the GPU parallel computing architecture followed by the CUDA programming model which facilitates the developing data parallel applications on NVIDIA GPUs.

3.1 Processor Architecture

Traditionally designed to excel in visualization tasks like compute intensive rendering, the architecture of a GPU makes it an ideal candidate for massively parallel data processing. In general, unlike CPU, a GPU has more transistors dedicated to data processing than to data caching and flow control. A basic building block of NVIDIA GPUs is a multiprocessor with 8 cores, up to 16384 32-bit registers, 16KB memory shared between 1024 co-resident threads (a multiprocessor executes a block of up to 16 warps, comprising of up to 32 threads, simultaneously). With up to 240 cores (30 multiprocessors) and memory bandwidth up to 102 GBps, the latest generation of GPUs offers extremely cost-effective computational power not only for visualization but also for general purpose scientific computations [25].

3.2 Memory Architecture

NVIDIA GPU memory model is highly hierarchical and there exist per-thread local memory, per-thread-block shared memory and device memory which aggregates global, constant and texture memory allocated to a grid, an array of thread blocks. A thread executes a kernel, GPU program, and communicates with threads in the same thread block

via high-bandwidth low-latency shared memory. Generally, optimizing the performance of CUDA applications could involve optimizing data access patterns to these various memory spaces. Each of the memory space has certain performance characteristics and constraints. Efficient implementation of solver kernels must consider carefully CUDA memory spaces, specifically, local and global memories which are not cached and have high access latencies.

3.3 CUDA programming model

NVIDIA's Compute Unified Device Architecture (CUDA) is a general purpose parallel computing architecture with a novel parallel programming model and instruction set architecture. C for CUDA exposes the CUDA programming model as an abstraction of GPU parallel architecture using a minimal set of extensions to the C language by allowing programmers to define C functions, called kernels. When called, these kernels are executed N times in parallel by N different CUDA threads in a hierarchical fashion, as opposed to only once as is the case in regular C functions. NVIDIA also provides CUBLAS, a BLAS (Basic Linear Algebra Subprograms) library ported to CUDA, which enables the use of GPUs without direct operation of the CUDA drivers. Our solver implementations use this library mainly for vector-vector operations. CUBLAS does not provide sparse matrix storage structures.

4. RELATED WORK

Sparse Grid methods for option pricing: Combination technique was first introduced in [3]. Some results on convergence and error analysis of the method (which are not under investigation here) can be found in [1, 5, 6, 9]. The sparse grid combination technique has been successfully applied in fluid mechanics [4], in data mining [7] or in finance [2, 10]. The combination technique for solving the Black-Scholes PDE used in this work is in line with [4].

GPU based linear solvers: GPU memory can be efficiently utilized for solvers where the matrix has a regular structure. In this work, our target is to solve systems with irregular sparsity. A few other algorithms have been studied to solve sparse and dense linear systems. Dense linear algebra routines are provided by NVIDIA, and their careful optimizations are studied in [20]. It is well known that due to their regular access patterns, dense linear algebra algorithms are well suited to GPU architecture. However, due to the amount of data contained in a dense matrix, in most cases the computations are bandwidth limited. Direct factorization-based solvers have been ported to GPUs [21], [22]. Most of these works rely on blocking strategies to parallelize the operations.

Sparse linear algebra is somewhat more difficult to adapt to GPUs, at least for unstructured problems. Several techniques have been proposed in the literature [18, 11, 16]. The major issues involve how the sparse matrix is stored (compressed storage formats), and whether blocking is used. To exploit massive parallelism offered by GPU, the optimizations for reducing memory footprint and hence hiding the memory access latency is very important. Bell et al [16] compare GPU SpMV results with SpMV results on various multi-core platforms obtained by [17] and illustrate that GPUs offer best performance.

The first GPU-based Conjugate Gradient solver for unstructured matrices is proposed in [18]. To utilize memory bandwidth, blocked compressed sparse row (BCSR) format matrix storage is used in [11] instead of CSR. BCSR decreases number of memory fetches from the device memory to some extent, however, number of elements to be multiplied increases. Both these works solve systems in single precision floating point. A mixed precision, multi-grid solver for a GPU cluster is proposed in [24]. The multi-gpu based general purpose symmetric linear systems solver with double precision solution accuracy is presented in [23].

In [18], authors use textures to store non-zero coefficients of a matrix and its associated two level look tables for CSR format to implement conjugate gradient solver. The lookup table is used to address the data and to sort the rows of the matrix according to the number of non-zero coefficients in each row. Then an iteration is performed on the GPU simultaneously over all rows of the same size to complete, for instance, a matrix-vector operation. Another approach to implement sparse matrices based on CSR format was proposed in [19] and it utilizes vertex buffers where each vertex buffer is used for each non-zero element.

Kernels		Methods		
Kernel	FLOPs	Method	Initial	Iterations
SpMV	2α	CGS	$10n + 2\alpha - 4$	$24n + 4\alpha - 3$
dot	$2n - 1$	BiCGStab	$10n + 2\alpha - 4$	$30n + 4\alpha - 5$
axpy	$2n$	BiCG	$5n + 2\alpha - 1$	$15n + 2\alpha - 4$
scal	n			

Table 1: FLOPs for Basic Kernels and Iterative Solvers. n is dimension of a matrix, α is number of nonzeros.

5. ITERATIVE SOLVERS ON GPUS

We are interested in methods for solving nonsymmetric systems of linear equations, that arise from discretization of partial differential equations. One of the leading families for linear system solvers is iterative solvers known as Krylov subspace methods [15]. We selected BiCGStab and CGS methods for their suitability for solving nonsymmetric linear systems. Iterative methods use four basic computational kernels: matrix vector products, preconditioner, inner product (dot), and vector update (axpy). The choice of the preconditioner is very important for the efficient solution of a linear system, but we will not discuss preconditioning here because it is often problem dependent. The constituent kernels along with their computational costs are presented in Table 5. The efficiency of any iterative method is determined primarily by the performance of the matrix-vector products and therefore on the storage scheme used for the matrix. We elect some sparse matrix representations which could be suitable for matrices resulting in the combination technique and evaluate the performance of linear solvers with respect to various implementations of GPU based matrix-vector products.

5.1 Sparse Matrix Formats

There exist several sparse storage formats with the aim of representing sparse matrices economically. A survey of various sparse storage formats can be found in [15]. These differ in terms of amount of storage required, the accessing methods such as the amount of indirect addressing required

for fundamental operations like matrix-vector products, and their adaptability for parallel architectures of GPUs. Due to matrix sparsity, memory access patterns tend to be highly irregular and utilization of global uncached memory can suffer from low spatial or temporal locality. Each format takes advantage of specific properties of the sparse matrix and may achieve different degree of efficiency of space and computational efficiency. We prefer to consider general storage formats which are suitable for matrices with arbitrary sparse structure. Hence, we consider following formats in our study:

- **Coordinate (COO)**: is a general sparse matrix format that comprises of arrays *row*, *col* and *data* to store row indices, column indices and values of nonzero matrix entries, respectively. This format is very space inefficient and computationally intensive among the formats we considered.

- **Compressed Sparse Row (CSR)**: is a general-purpose sparse matrix format. It does not consider any ordering among nonzero values within each row. Subsequent nonzeros of rows are stored in contiguous memory, and additional integer arrays specify column index for each nonzero and beginning of offset of each row.

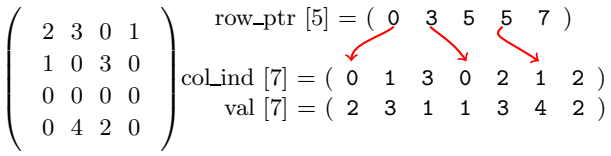
- **Block Compressed Sparse Row (BCSR)**: is particularly useful when the sparse matrix has square dense blocks of nonzeros in some regular pattern. It enables register blocking strategies, and vector processing significantly reduces the required memory bandwidth and computational time for matrices with large block sizes [15].

- **Hybrid (HYB)**: is a combination of the **Ellpack-Itpack (ELL)** (or **Diagonal (DIA)**) and COO format, by coupling the speed of ELL (or DIA), utilizing the memory bandwidth efficiently, and the flexibility of COO. It is usually the fastest format for a wide range of unstructured matrices.

5.2 Sparse Matrix Operations

Sparse matrix-vector multiplication is arguably the most important operation in sparse matrix computations.Iterative solvers generally require hundreds, if not thousands, matrix-vector products to reach convergence. Over the past decade, there has been significant amount work on optimizing SpMV. Most of the work has focused on optimizing sparse matrix kernels on general-purpose architectures. SpMV being a memory bound kernel, most optimizations target performance improvements at various memory levels in the memory hierarchy. The optimizations include optimal data structure for storing sparse matrix, exploiting block structures in sparse matrix and blocking for reuse at the level of cache, TLB and registers [12]. Various optimizations have been proposed taking into account the complex memory hierarchy and unconventional mapping of computation to the coresident threads on GPUs.

Let us take an example of a sparse matrix A, as represented in Figure 1 with its CSR representation and a dense vector x of length n. The objective is to compute $y = Ax$, where y is the output dense vector of length n. In Figure 1, we present a sequential implementation of the SpMV procedure using CSR format. There are several ways to parallelize



a) Sparse Matrix A b) CSR format of Matrix A

Figure 1: An example of CSR representation

```
__host__ void
spmv_csr_serial ( const int num_rows ,
                  const int * row_ptr ,
                  const int * col_ind ,
                  const float * val ,
                  const float * x,
                  float * y)
{
  for ( int row = 0; i < num_rows ; i ++){
    float rowsum = 0;

    int row_start = row_ptr [row];
    int row_end = row_ptr [row +1];

    for (int jj = row_start ; jj < row_end ; jj ++)
      rowsum += val [jj] * x[col_ind[jj]];

    y[row] += rowsum;
  }
}
```

Figure 2: Serial routine for CSR SpMV

```
__global__ void
spmv_csr_scalar_kernel ( const int num_rows ,
                         const int * row_ptr ,
                         const int * col_ind ,
                         const float * val ,
                         const float * x,
                         float * y)
{
  int row = blockDim.x * blockIdx.x + threadIdx.x ;

  if(row < num_rows ){
    float rowsum = 0;

    int row_start = row_ptr [row];
    int row_end = row_ptr [row +1];

    for (int jj = row_start ; jj < row_end ; jj ++)
      dot += val[jj] * x[col_ind[jj]];

    y[row] += rowsum;
  }
}
```

Figure 3: A naive CUDA kernel for CSR SpMV

this procedure. A simple translation of this sequential routine into a CUDA kernel is given in Figure 3. In this naive approach of parallelizing the main loop, each row is assigned to one thread. One of the major problems with this parallel implementation on GPU is the way how threads within a warp access the *col_ind* and *val* arrays. Despite these values are stored contiguously, each thread reads the ele-

ments of its row sequentially, resulting in non-contiguous accesses to off-chip memory and thus hampering the overall performance of the memory bound kernels. Additionally, if the nonzero elements are unevenly distributed across all the rows, it may lead to poor utilization of the resources keeping many threads idle. Several algorithmic as well as architecture specific optimizations have been proposed to improve the performance of both serial and parallel implementations [17, 16, 12, 11]. We experiment with the following three libraries to implement the linear solvers. NVIDIA's CUBLAS is used for dense vector operations.

5.2.1 NVIDIA SpMV Library

NVIDIA made available a library for SpMV computation and is one of the prominent efforts in this direction. In [16] authors discuss the implementation details of various sparse matrix formats and their efficient representation on NVIDIA GPUs. The library supports DIA and ELL formats suitable for small sparse/dense matrices. The other formats suitable for large sparse matrices include COO, CSR, HYB and Packet (PKT) format.

The *COO-vector* kernel based on segmented reduction is robust with respect to variations in row sizes. It is reliable and complements the deficiencies of the other SpMV kernels. *COO-scalar* kernel uses only one thread and is equivalent to sequential algorithm. The *CSR-scalar* kernel uses one thread per matrix row. It has low bandwidth utilization and hence poor computational performance, as it does not exploit memory coalescing. On the contrary, *CSR-vector* uses a 32-thread warps per row results in contiguous memory access. This may result in poor load balancing by increasing the number of idle threads when the number of nonzero coefficients is less than the warp size. Performance of *CSR-scalar* is rarely competitive with alternative choices, while the vector kernel efficiently handles large row sizes. The *HYB kernel* is a combination of *COO-vector* and ELL kernels. All the kernels we used in our implementation benefit from the read-only *texture cache* present on all CUDA-enabled devices.

5.2.2 IBM SpMV Library

In [12], authors propose GPU specific, application oriented optimizations for efficient execution of SpMV kernels using CUDA. The GPU specific optimizations include i) Exploitation of synchronization-free parallelism for achieving intra-thread-block synchronization across the rows, ii) Optimized thread mapping based on the *affinity towards optimal memory access pattern*, iii) Enabling hardware optimized global memory coalesced accesses and iv) Exploiting data reuse of input vector elements by caching the elements in on-chip memories like texture (hardware) cache, registers or shared memory (software) cache. Texture memory is used to store the input vector to utilize the read-only texture cache in order to achieve performance gains due to input vector reuse.

The proposed optimizations are for the CSR format and the authors claim to achieve better performance than the NVIDIA SpMV library. We evaluate the performance of our solvers with Padded CSR format, in which zeros are padded to ensure that the number of entries in each row is a multiple of 16 in order to achieve aligned global memory access.

Feature	C870	E5420
Multiprocessors	16	2
Processor cores	128	8
Processor Clock	1.35 GHz	2.6 GHz
Off-chip Memory Size	1.5 GB	4 GB
Peak Performance	500 GFlops	80 GFlops

Table 2: Architectural configurations of NVIDIA Tesla C870 and Intel Xeon E5420.

5.2.3 CNC SpMV Library

In [11], authors present SpMV package which includes efficient implementation for BCSR format. The sparse storage format groups non-zero values in blocks of size BN x BM in order to maximize the memory fetch bandwidth of GPUs, to take advantage of registers to avoid redundant fetches (register blocking), and to reduce the number of indirections due to the reduced size of lookup tables. Although each indirection results in dependent memory fetches, it introduces memory latencies that need to be hidden by the GPU to achieve a good efficiency. We experiment with implementations of 2x2 and 4x4 blocks for lower and higher filling ratio respectively.

6. NUMERICAL RESULTS AND PERFORMANCE

In this section, we discuss the results of a set of experiments using NVIDIA GPU to demonstrate the performance behavior of BiCGStab and CGS.

6.1 Architectures

The sequential, double precision versions of BiCGStab and CGS solvers were developed and experimented on Intel Xeon E5420, see Table 2. For the dense vector operations we used BLAS library (1.2-1.3ubuntu3) and supported only CSR sparse format. The sequential solvers and any other CPU bound code was compiled using gcc compiler at *-O3* optimization level. The GPU based solvers were executed using Tesla C870 (G80 GPU) with configuration presented in Table 2, connected to a host x86/Linux system through 16-x PCI Express bus. The CUDA kernels for SpMV operations were compiled using NVIDIA CUDA compiler 2.1 (nvcc) to generate the device code. The device code was compiled with *-arch=sm_10* flag and *-O3* optimization level. Note that due to limitations of Tesla C870, experiments for GPU based solvers were done with single precision arithmetic.

6.2 Validation of the results: Sparse Grid, Full Grid and Monte Carlo prices

We consider the test case presented in Section 2, with an up-and-out best-of barrier option on d assets. The payoff is written $h(\mathbf{S}) = (\max_i S_i - K)^+$ on the domain $[0, B]^d$. Our numerical results are presented in Table 3. The following parameters have been used to obtain these results: $\sigma_i = 0.20, \rho_{ij} = 0.0, r = 0.03, T = 1, K = 100, B = 200$. We solve the PDE using the combination technique at level $n = 6, \ldots, 10$ with 5000 timesteps. We present the resulting linear systems for $d = 3$ and $n = 10$ in Table 5. For lower dimensions and levels the linear systems are smaller in size, however, the number of unknowns increases with the increase in dimensions and approximation level.

d	MC interval	l=6	7	8	9	10
2	[15.97-16.04]	16.68	16.49	16.10	16.06	15.99
3	[20.87-20.94]	23.49	22.59	21.69	20.80	20.85
4	[24.63-24.69]	26.98	31.43	NA	NA	NA
5	[27.61-27.68]	19.6145	NA	NA	NA	NA

Table 3: Numerical results. Monte Carlo confidence intervals computed using 1E6 paths and 250 timesteps. Note that MC prices are positively biased (see text).

d	Level	Total Time(s)	Solver Time(s)
3	6	15.71	15.10
3	7	70.26	65.30
3	8	283.08	242.13
3	9	1155.27	818.10
3	10	5375.96	2553.31

Table 4: Time in seconds for pricing a basket of three equities with different refinement levels.

As a rough validation, we simply check a few prices for at-the-money option, i.e. in our case where all assets have the initial value $S_0 = 100$. The accuracy of our numerical results is verified by comparison with a Monte Carlo 95% confidence interval obtained with $1E6$ simulation paths and 250 timesteps. It is known that these confidence intervals are positively biased, since the evaluation of the crossing of the barrier is not evaluated continuously. Table 3 shows that convergence is achieved, at least for $d = 2, 3$. For error analysis of the combination technique please refer to [6].

We also compute the standard full grid finite difference solutions. Using the notation introduced in section 2, standard full grid solutions $\tilde{V}_{\mathbf{i}}$ are computed directly by solving the PDE (1) discretized on an isotropic grid $\Omega_{\mathbf{i}}$ with $\mathbf{i} = (l, l, \ldots, l)$. For $d = 3$ and $l = 6$, the GPU computation of the solution takes more than 8 hours (of which the *Solver Time* is <1%), and the price for all assets at the money (i.e. as in Table 3) is 20.66. This can be compared to the sparse grid combination technique solution at level $l = 7$ (such that finest one-dimensional discretization are the same in both cases), which takes only 70 seconds, but has a larger error (see Tables 3 and 4). This can also be compared to a sparse grid combination technique with similar error, which takes only 20 minutes at level $l = 9$. Theses few comparisons establish the benefit that one may get from an efficient implementation of the sparse grid combination technique. Next subsection is dedicated to the performances of the GPU linear solvers that are used in the study.

6.3 Performance of the linear solvers

For the experiments, the convergence tolerance of the iterative methods is set to $1E$-6. The execution times using sequential solvers for $d = 3$ with $n = 6, \ldots, 10$ are presented in Table 4. The column *Solver Time* represents the time required for solving the linear discrete systems. The remaining of the *Total Time* is spent on constrcuing partial grids and combining the subsolutions. It usually remains constant and is independent of parallel or sequential solvers

$\prec l_1, l_2, l_3 \succ$	N	NNZ	$\prec l_1, l_2, l_3 \succ$	N	NNZ
$\prec$ 8,1,1 $\succ$	255	763	$\prec$ 5,3,1 $\succ$	217	1009
$\prec$ 7,2,1 $\succ$	381	1645	$\prec$ 5,2,2 $\succ$	279	1563
$\prec$ 6,3,1 $\succ$	441	2065	$\prec$ 4,4,1 $\succ$	225	1065
$\prec$ 6,2,2 $\succ$	567	3195	$\prec$ 4,3,2 $\succ$	315	1863
$\prec$ 5,4,1 $\succ$	465	2233	$\prec$ 3,3,3 $\succ$	343	2107
$\prec$ 5,3,2 $\succ$	651	3895	$\prec$ 6,1,1 $\succ$	63	187
$\prec$ 4,4,2 $\succ$	675	4095	$\prec$ 5,2,1 $\succ$	93	397
$\prec$ 4,3,3 $\succ$	735	4627	$\prec$ 4,3,1 $\succ$	105	481
$\prec$ 7,1,1 $\succ$	127	379	$\prec$ 4,2,2 $\succ$	135	747
$\prec$ 6,2,1 $\succ$	189	813	$\prec$ 3,3,2 $\succ$	147	847

Table 5: A set of discrete systems for $d = 3$ and $l = 10$ with level of discretization in each direction given by the permutation of $\prec l_1, l_2, l_3 \succ$. N is dimension of a matrix and NNZ is number of nonzeros. The total number of discrete systems is 85.

used. Hence, for the performance evaluation, we only consider the *Solver time*, the total time spent in solving the all grids. The speedups achieved for *Solver Time*s using GPU based BiCGStab and CGS are presented in Figure 4 and Figure 5, respectively. The speedup is computed as the ratio of the execution time using one CPU(T_1) and the parallel execution time (T_p) using Tesla. Overall performance of these solvers in terms of mega FLOPs (MFLOPs) is presented in Figure 6 and Figure 7. The total number of FLOPs are computed as described in Table 5.

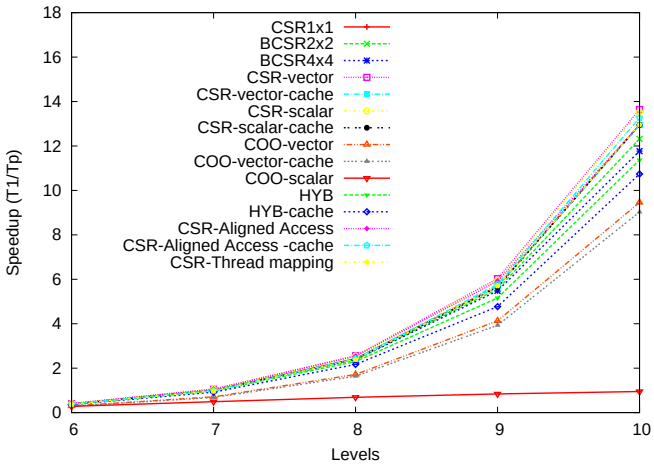


Figure 4: Speedup for BiCGStab using GPU for increasing problem sizes.

We evaluate the performance of GPU based BiCGStab and CGS with respect to the underlying SpMV kernels. For the analysis, we call both the solvers by the SpMV kernel name used therein (e.g. *CSR-Scalar* stands for a BiCGStab or CGS solver using NVIDIA's scalar CSR SpMV kernel). The performance characteristics of both the solvers follow similar behavior for scalability, hence in the following discussion we do not distinguish between both the solvers, if not mentioned otherwise. The other solver kernels (*dot*, *axpy* and *scal*) in

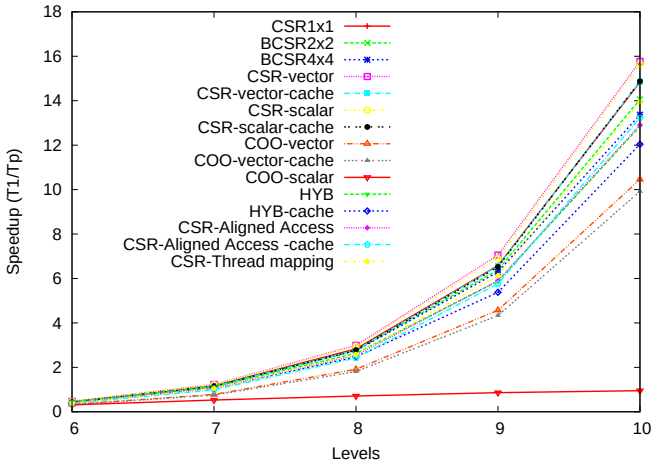


Figure 5: Speedup for CGS using GPU for increasing problem sizes.

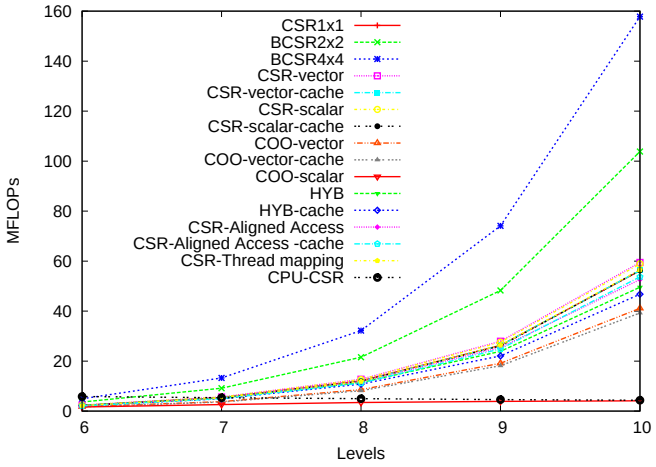


Figure 6: Overall speed (megaflops) of BiCGStab using GPU.

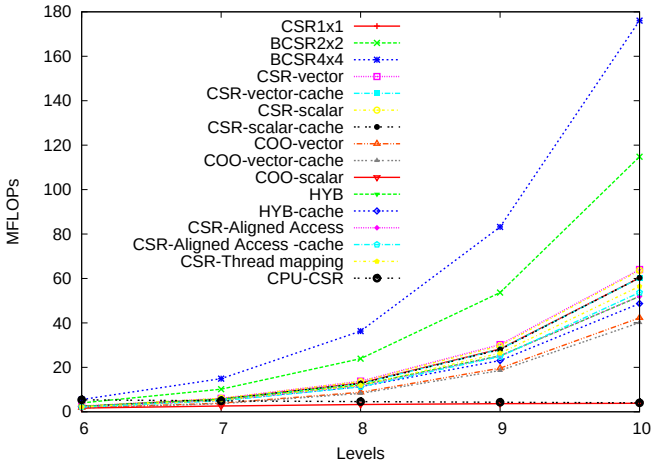


Figure 7: Overall speed (megaflops) of CGS using GPU.

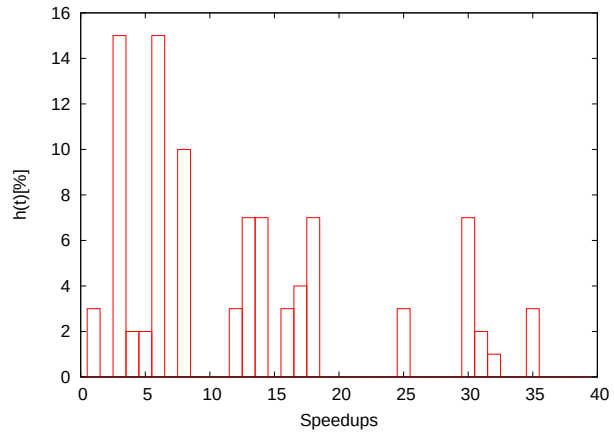


Figure 8: Histogram h[t] of Speedups achieved by GPU based BiCGStab using *NVIDIA-CSR-vector* kernel for solving discrete systems represented by Table 5.

both the solvers are assumed to take equivalent time for solving a perticular grid.

With CNC library: BCSR2x2 and *BCSR4x4* kernels take advantage of square dense block patterns in a sparse matrix, saving in locations and reduction in indirect addressing. The sparse grids resulted in the combination technique does not exhibit any of such pattern. This format leads to register level data reuse resulting coarse grained parallelism, however in our case, suffers from non-optimal global memory accesses. This attributes to its poor performance compared to other variants of CSR formats that we investigated. *CSR1x1* is uses one thread per row and 16 threads per block. The equivalent NVIDIA *CSR-scalar* kernel uses 256 threads per block. *CSR1x1* comparatively results in poor thread granularity due to more number of blocks. *CSR1x1* results in an inefficient utilization as the multiprocessor spends a large fraction of time in block switching. Optimal block size (and grid size) is necessary to ensure maximal utilization of the resources.

With NVIDIA's library: The *CSR-vector kernel* uses one 32-thread warp per matrix row. This approach benefits from contiguous global memory access. However, as our testcases have small scale, the resulting matrices have fewer than 32 non-zeros per row, which causes underutilization of this kernel. Although this kernels outperforms our other SpMV considerations. A *CSR-scalar* which uses one thread per matrix row performs poorly compared to its vector counterpart. It uses 256 threads per block and outperforms *CSR1x1* which uses 16 threads per block. The *COO-vector* kernel is based on segmented reduction. The COO format has the worst computational intensity of all. The segmented reduction operation seems more expensive than alternative approaches that distribute work across threads of execution. *COO-scalar* approach which allocates only one thread has the worst computational complexity, equivalent of sequential CSR. Nevertheless, *COO-vector* kernel can be used to compensate the deficiencies of the other kernels. The *HYB* format couples the speed of ELL and the flexibility of COO. The ELL alone cannot be used in our matrices as there are very small number of nonzero entries per row. But

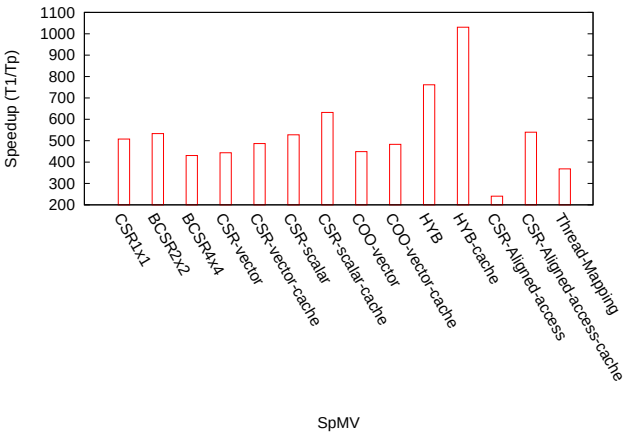


Figure 9: The speedup for solving by full grid method using GPU based BiCGStab. A linear system with $N = 29791$, $NNZ = 202771$. Sequential time required is 5884.6 seconds.

when combined with *COO-vector* in *HYB-kernel* we observe the performance improvement over *COO-vector* and *COO-scalar* kernels.

With IBM's Library: We experiment with two main compile-time optimizations proposed in [12], *CSR-Aligned Access*, aligned global memory accesses, and *Thread Mapping*, optimized thread mapping, for the CSR format. *CSR-Aligned Access* kernel is similar to *CSR-vector* but uses 16-thread warp per matrix row and makes alignment adjustments. The matrices resulting from combination technique have very low number of nonzeros per row, usually less than 16, which contributes to the poor performance of these optimizations.

In terms of speed, the performance of the CPU remains constant (around 4 MFLOPs) while the performance of GPU based solvers boosts with the increasing problem sizes, see Figure 6 and Figure 7. CPUs generally mask the latency with on-chip cache, while GPUs mask the latency with a large number of threads and assigned work load. The increasing work load can promise to hide the latency to do texture accesses. With the problem sizes at our disposal, most of the cached kernels fail to hide the latency causing unexpected slowdown. For small grids, both CPU and GPU implementations exhibit equivalent performance. Figure 8 displays a histogram with the distribution of number of grids as per the speedups achieved using GPU based BiCGStab. We observed that solvers achieve better speedups for the grids with large unknowns than for the smaller grids.

The speedup achieved using GPU based BiCGStab for solving a full grid with $d = 3$ and $l = 5$ is presented in Figure 9. The sequential time required for solving the linear systems associated with this problem is 5884.6 seconds (while the total time is 6280.84 seconds). We can clearly see the solver using the HYB sparse format (with and without cache enabled) achieves the highest absolute performance with the speedup of more than 1000. In the figure we can also observe the benefit of enabling cache for the large size vector, with an average of 20% improvement in the performance. Thus we can observe the influence of cache on the optimum choice of the storage format, and that the exploitation of storage formats is necessary for achieving higher performance. Further performance analysis of HYB-kernels for general set of problems can be found in [16].

7. CONCLUSION AND FUTURE WORK

In this paper we presented parallel implementations of Krylov subspace based iterative solvers harnesssing GPUs for solving high dimensional PDEs stemming from multi-dimensional option pricing problems. The efficiency and performance of various SpMVs and therefore of the linear solvers is demonstrated by numerical experimental results carried out on a NVIDIA GPU. Our results show that the choice of sparse format is not only important for scalability of iterative solvers to solve the sparse grids but the efficient implementation and parameter tuning of matrix-vector kernels is also essential for maximal performance. The solution of grids with low approximation levels results in smaller problem sizes attributing to the overhead of the parallel platform and performs poorly compared to sequential version. For higher approximation levels, we observed that the significant amount of time is spent on creating partial sparse grids. An interesting engineering problem would be to construct these grids on the GPU itself. Further, in order to exploit the inherent parallelism in the combination technique, it is essential to distribute the solving of sparse grids over a cluster of GPUs. Such scheme poses several interesting future research directions. The variable times needed to solve the discrete systems would require either a reasonable distribution scheme or ways to decompose large systems over multiple GPUs.

We will also work toward a comprehensive object-oriented library of high-performance iterative linear solvers using various SpMV routines and effective preconditioners for the solvers, targeting future generation GPU devices. Future work will also include investigating the performance and convergence behavior of these solvers on the latest double precision NVIDIA CUDA enabled devices.

8. REFERENCES

[1] H. J. Bungartz, M. Griebel, D. R. Oschke, and C. Zenger. Pointwise convergence of the combination technique for the laplace equation. *East-West J. Numer. Math*, 1994.

[2] M. Griebel, T. Gerstner, and S. Wahl. *Method and device for evaluation of financial derivatives using sparse grids*. US Patent App. 09/994,114.

[3] M. Griebel, M. Schneider, and C. Zenger. A combination technique for the solution of sparse grid problems. *Iterative Methods in Linear Algebra*, pages 263–281, 1992.

[4] M. Griebel and V. Thurner. Efficient solution of fluid dynamics problems by the combination technique. *International Journal of Numerical Methods for Heat and Fluid Flow*, 5(3):251–269, 1995.

[5] C. Pflaum. Convergence of the combination technique for Second-Order elliptic differential equations. *SIAM J. on Num. Ana.*, 34(6):2431–2455, Dec. 1997.

[6] C. Pflaum and A. Zhou. Error analysis of the combination technique. *Numerische Mathematik*, 84(2):327–350, Dec. 1999.

[7] J. Garcke, M. Griebel, and M. Thess. Data mining with sparse grids. *Computing*, 67(3):225–253, Oct. 2001.

[8] O. Pironneau and Y. Achdou. *Computational Methods for Option Pricing*. Society for Industrial & Applied Mathematics,U.S., illustrated edition edition, 2008.

[9] C. Reisinger. Analysis of linear difference schemes in the sparse grid combination technique. *SIAM Journal on Numerical Analysis*, 2006.

[10] C. Reisinger and G. Wittum. Efficient hierarchical approximation of high-dimensional option pricing problems. *SIAM J. on Scientific Computing*, 29(1):440, 2008.

[11] L. Buatois , G. Caumon and B. Levy. Concurrent number cruncher - A GPU implementation of a general sparse linear solver. *Intl. Journal of Parallel, Emergent and Distributed Systems*, to appear.

[12] M. Baskaran and R. Bordawekar. Optimizing Sparse Matrix-Vector Multiplication on GPUs. *IBM Technical Report RC24704.*, 2008.

[13] J. Hull Options, Futures, and Other Derivative Securities *Prentice Hall, 4th Edition 2000.*, 2nd Edition, 1993.

[14] A. Brandt. Multilevel adaptive solutions to boundary-value problems. *Math. Comp.*, 31:311329, 1977.

[15] R. Barrett et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. *SIAM*, Philadelphia, 1994.

[16] N. Bell and M. Garland Efficient Sparse Matrix-Vector Multiplication on CUDA. *NVIDIA Technical Report*, Dec. 2008.

[17] S. Williams, L. Oliker , R. Vuduc amd J. Shalf , K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Proc. of the 2007 ACM/IEEE conf. Supercomputing*, 1-12, 2007.

[18] J. Bolz, I. Farmer, E. Grinspun, and P. Schörder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917-924, July 2003.

[19] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. on Graphics (TOG)*, 22(3):908-916, 2003.

[20] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. *Proc. of the 2008 ACM/IEEE conference on Supercomputing*, 2008, pp. 1-11.

[21] N. Galoppo , N. K. Govindaraju , M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. Proc. of the 2005 ACM/IEEE conference on Supercomputing, 2005.

[22] G. Quintana-Orti , F. D. Igual , E. S. Quintana-Orti, and R. van de Geijn. Solving dense linear algebra problems on platforms with multiple hardware accelerators. *Proc. of the 14th ACM SIGPLAN Sym. PPPP*, 2009.

[23] A. Cevahir , A. Nukada, and S. Matsuoka. Fast Conjugate Gradients with Multiple GPUs. *Computational Science, ICCS*, Vol.5544 Springer (2009),p.893-903.

[24] D. Göddeke , R. Strzodka , J. Mohd-Yusof , P. McCormick , H. Wobker , C. Becker, and S. Türek. Using GPUs to Improve Multigrid Solver Performance on a Cluster. *Intl. Journal of CSE*, 2008.

[25] http://www.nvidia.com/object/product_tesla_c1060_us.html