General-Purpose Sparse Matrix Building Blocks using the NVIDIA CUDA Technology Platform

Matthias Christen, Olaf Schenk, *Member, IEEE,* and Helmar Burkhart, *Member, IEEE*

***Abstract*— We report on our experience with integrating and using graphics processing units (GPUs) as fast parallel floating-point co-processors to accelerate two fundamental computational scientific kernels on the GPU: sparse direct factorization and non-linear interior-point optimization. Since a full re-implementation of these complex kernels is typically not feasible, we identify e.g. the matrix-matrix multiplication as a first natural entry-point for a minimally invasive integration of GPUs. We investigate the performance on the NVIDIA GeForce 8800 multicore chip. We exploit the architectural features of the GeForce 8800 GPU to design an efficient GPU-parallel sparse matrix solver. A prototype approach to leverage the bandwidth and computing power of GPUs for these matrix kernel operation is demonstrated resulting in an overall performance of over 110 GFlops/s on the desktop for large matrices. We use our GPU algorithm for PDE-constrained optimization problems and demonstrate that the commodity GPU is a useful co-processor for scientific applications.**

***Index Terms*— GPGPU, graphical processing units, sparse matrix decomposition, sparse direct solvers, large-scale nonlinear optimization**

I. Introduction

GRAPHICS processing units (GPUs) have evolved into a very attractive hardware platform for general purpose computations due to their extremely high floating-point processing performance, huge memory bandwidth and their comparatively low cost [1]. The rapid evolution of GPUs in performance, architecture, and programmability can provide application potential beyond their primary purpose of graphics processing. High-end GPUs [2] or the STI Cell-processors [3], which are integrated in the Sony PlayStation 3, typically deliver performance of at least one order of magnitude higher compared to that of the CPU, while at the same time equipped up to 1 GB of GPU main memory. This commodity graphics hardware can become a cost-effective, highly parallel platform to solve scientific problems.

In this paper we present an extensive matrix algorithmic performance study on GPUs using the novel NVIDA CUDA technology platform to build general-purpose sparse matrix building blocks. Following the current trend to perform computationally intensive operations on a specialized processor rather than on the CPU, we will use a GPU as a mathematical co-processor to accelerate sparse direct linear solvers [4], [5], [6]. Our stream computing unit is based on the NVIDIA GeForce 8800 which features a scalable ultra-threaded architecture, high performance parallel processing on 128 shader processors and is equipped with 768 MB on-board memory.

Our primary goal is to investigate the performance acceleration of dense and sparse matrix solution kernels. These matrix linear algebra algorithms are of importance and represent fundamental kernels in many computationally intensive scientific applications such as nonlinear optimization, computer tomography, geophysical seismic modelling, semiconductor device simulations, and in the solution of partial differential equations in general. The performance of all these applications rely heavily on the availability of fast sparse matrix solution kernel routines on CPUs or GPUs.

There are many algorithms for factoring large sparse linear systems of equations. Since the early 1990's, it is clear that exploiting cache memories has become crucial for achieving high performance in sparse matrix factorization [7], [8]. The key is to group consecutive columns with identical nonzero structure together in order to exploit cache memories in sparse matrix factorizations. Multifrontal and supernodal codes have been developed that can effectively exploit the memory hierarchies of cache-based microprocessors. With the right data-structure, the vast majority of floating-point operations can be performed within highly-tuned BLAS 3 operations [9] such as the matrix-matrix multiplication routines and near peak performance can be expected on modern architectures. For a recent detailed survey on sparse matrix techniques for large linear systems of equations, the interested reader should consult [10].

A. Contributions

We map two fundamental computational kernels as general-purpose sparse matrix building blocks onto the GPU: a sparse direct linear factorization method for nonsymmetric and symmetric indefinite matrices based on the PARDISO framework [4], [5], [6] and an interior-point optimization solver for large-scale nonconvex PDE-constrained optimizations [11]. Both are workhorses of physical modeling and optimization applications. We analyze their performance on NVIDIA's GeForce 8800 in realistic large-scale applications.

B. Organization

The remainder of the paper is organized as follows: In Section II we present a brief overview of related work. In Section III we will investigate the parallel performance on GPUs for several dense matrix computational kernels that arise in sparse matrix factorizations. We then present our algorithmic design to parallelize sparse matrix factorizations on the GPU and discuss strategies to optimize the GPU performance in Section IV. In Section V we use the GPU to accelerate large-scale nonconvex interior-point optimizations problems that arise e.g. in PDE-constrained optimizations.

II. Related Work

A. Scientific Computations on GPUs

There have been several GPU-parallel implementations and investigations of dense matrix kernels on emerging multiprocessing architectures such as GPUs and the STI Cell processor. Galoppo et

M. Christen, O. Schenk and H. Burkhart are with the Computer Science Department of the University of Basel, Basel, Switzerland.

al. [12] analyzed the peak performance of a cache and bandwidth efficient GPU solver for dense matrix decompositions. Dongarra al. [13] propose to exploit single-precision operations whenever possible and resort to double-precision at critical stages while attempting to provide the full double precision results and present results for dense matrices on the IBM Cell processor.

Several GPU-based algorithms for sparse matrix multiplication on emerging architectures have been proposed e.g. in [14], [15], [16], [17]. These are all iterative methods such as conjugate gradient and multigrid solvers as described e.g. by Goeddeke et. al. [14]. Our work is related in spirit to these frameworks for linear algebra kernels and nonlinear optimizations on these architectures. However, there is little research on using GPUs for large-scale sparse direct factorizations or interior-point methods for nonlinear problems although these two algorithmic kernels represent workhorses in scientific computations.

III. Basic Linear Algebra Kernels on CPUs and GPUs

NVIDIA provides a highly tuned library containing basic linear algebra routines, CUBLAS. CUBLAS is a high-level API designed for compatibility with the original FORTRAN subprograms. It is built solely on top of CUDA. The entire set of single precision real BLAS routines [9] are available through CUBLAS as well as some single precision complex functions.

We performed benchmarks for the computationally intensive routines of interest in respect of the sparse direct linear solver PARDISO[4], [5], [6], namely the matrix-matrix multiplication (`sgemm`), the solving of a triangular matrix equation (`strsm`), and the LU and LDL^T decomposition (`sgetrf, ssytrf`). Unfortunately, the factorization routines being a LAPACK rather than a BLAS routine, are not part of CUBLAS. We have therefore implemented it ourselves using existing CUBLAS functions. Although therefore not being fine-tuned, it yields reasonable results for large matrices.

We have compared the performance results with benchmarks using the Intel Math Kernel Library (MKL) done on a dual-core 3.4 GHz Intel Pentium D CPU, which has 16 KB of L1 cache and 2 MB of L2 cache.

A. Matrix-Matrix Multiplication

Fig. 1 and Fig. 2 show the result of the 32-bit `sgemm` benchmark. The performance of an optimized CPU version and two versions of the CUDA implementation of the matrix-matrix multiplication $\mathbf{A} \cdot \mathbf{B}$ have been measured, where $\mathbf{A} \in \mathbb{R}^{m \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times n}$. The two plots in Fig. 1 display the performance results of the CPU using the Intel MKL 9.1 library, whereas the plots in Fig. 2 show the GPU performance results. The first GPU performance measurement, which is displayed in the first row, includes the data transfer to the GPU before the multiplication as well as the data transfer back to the CPU system after the computation. The second measurement found in the bottom row omits GPU data transfers.

For the benchmarks, m and n were varied, whereas k was fixed at 50 for the two plots in the left and at 4096 for the plots in the right column.

The parameters m and n vary along the horizontal and vertical axes, respectively. The color shades correspond to the performance. Note that the two columns and rows use different color scales.

The plots show that for large matrices the GPU outperforms the CPU by nearly an order of magnitude: the CPU performs the multiplication at a relative constant rate of 12 GFLOP/s, while the GPU reaches a performance of more than 100 GFLOP/s for large matrices when omitting data transfers.

Unfortunately, multiplying matrices sized 200×200 and below has the contrary effect. The multiplication of small matrices is carried out faster by the CPU – even if no data transfer to and from the GPU system is involved. This is due to the fact that the multiplication on the GPU is carried out by many threads in parallel, which require some startup overhead.

B. Triangular Matrix Equation With Multiple Right-Hand-Sides

Another important dense matrix routine for sparse direct linear solvers is the LAPACK `strsm` method. In this section we will evaluate the GPU performance of `strsm` for quadratic matrices. The CUBLAS implementation achieves an impressive performance of up to 70 GFLOP/s for large matrices. Again, the CPU performs at a relative constant rate of circa 10 GFLOP/s and hence, the GPU `strsm` is almost an order of magnitude faster than the GPU `strsm` for large quadratic matrices.

The plot again shows two measurement versions of the computation performance on the GPU, once without taking data transfers into account and once including the GPU up- and downloads, which entails a constant performance penalty of circa 10 GFLOP/s. Also, two versions of the CPU implementation of the solver are depicted. The upper curve being the performance of a single precision (32-bit) and the lower curve that of a double precision (64-bit) solver. It is demonstrated that a performance gain of factor 2 is achieved when precision is reduced from 64 to 32 bit.

The plot at the right of Fig. 3 is a zoom into the lower left corner of the plot at the left to identify the the computational cross-over point. It shows that for matrix sizes as small as from 150×150 the GPU outperforms the CPU in our hardware configuration.

C. Dense Linear Factorization Solvers

As already mentioned, `sgetrf` is not part of CUBLAS. The following benchmark has been done using a one-to-one translation of the original LAPACK FORTRAN code to a code using existing CUBLAS functions. As in LAPACK, both blocked and non-blocked versions have been implemented.

Fig. 4 displays the time used by the constituents of the blocked LU decomposition that are computationally most intensive. Also, the total time consumed by the factorization is shown. The horizontal axis represents the block size, the vertical axis shows the time used by the decomposition. The blocked LU decomposition consists of three computational main components, the matrix-matrix multiplication (`sgemm`), the triangular solve (`strsm`), and a rank 1 update $\mathbf{A} := \alpha \cdot \mathbf{x}\,\mathbf{y}^T + \mathbf{A}$ (`sger`). The Figure suggests that nearly no time (in relation to the other constituents) is consumed by `strsm`, which is the bottom most curve. For small blocks, many matrix-matrix multiplications are required, which then dominate the calculation. As the block size increases fewer matrix-matrix multiplications are required, and also with larger matrix blocks, the performance of the multiplication increases as outlined in Section III-A. The time used by `sger` almost increases linearly with block size. Other function calls to BLAS level 1 routines are omitted in the plot as they consume a constant amount of time.

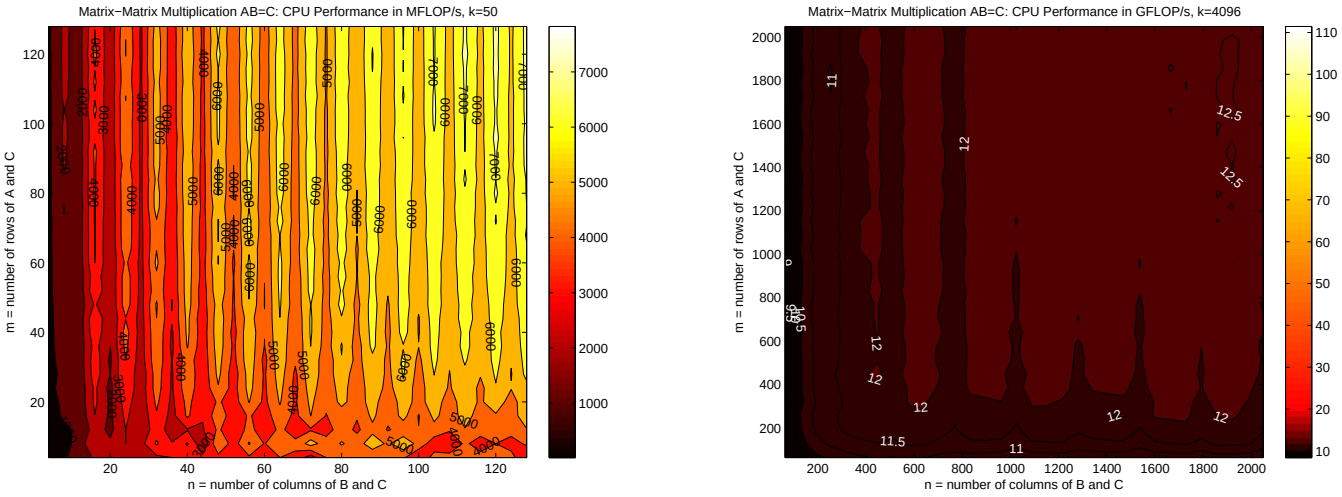

Fig. 1. Performance measurements for 32-bit `sgemm` on the CPU for fixed $k = 50$ (left) and $k = 4096$ (right).

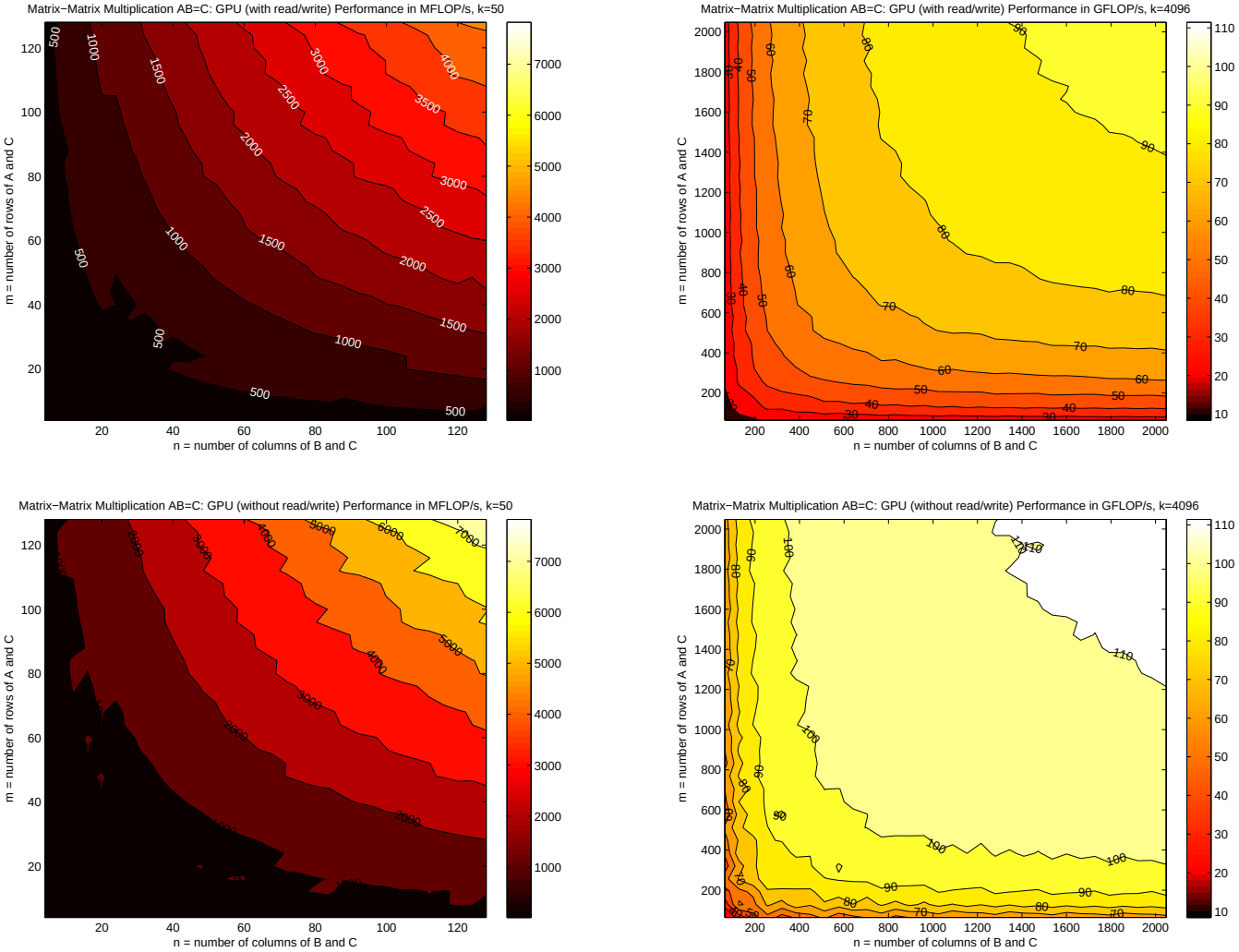

Fig. 2. Performance measurements for 32-bit `sgemm` GPU for fixed $k = 50$ (left) and $k = 4096$ (right). The top row includes the read/write data transfer from CPU main memory into GPU memory, whereas the bottom row assumes that all data can be stored on GPU memory.

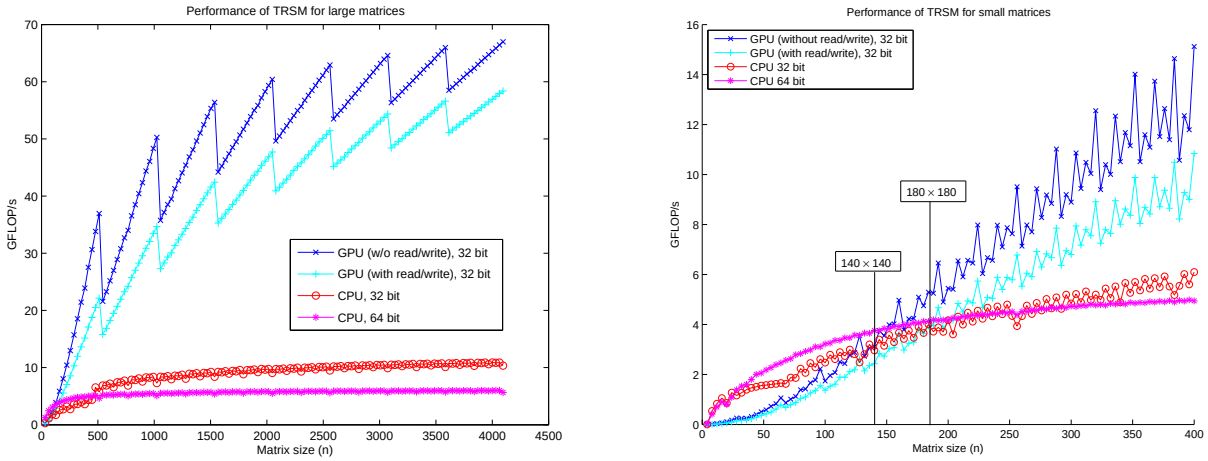

Fig. 3. Performance of triangular solve `[d/s]trsm` on CPU and GPU for large (left; $n < 4096$) and small (right; $n < 256$) $n \times n$ matrices.

For our hardware and software environment, a block size of 32-by-32 has proven to give the best speed-up. This is due to the facts that

- the CUBLAS implementations of `sgemm` and `strsm` yield best performances if the matrix sizes are divisible by 16 because of the GPU's memory organization;
- for larger block sizes less matrix-matrix multiplications and solves are required while the performance of the routines increase with matrix size;
- for all block sizes larger than a threshold value, the rank 1 update being a BLAS level 2 routine dominates the calculation. In the case of factoring a 2048-by-2048 matrix the CUBLAS implementation of `sger` merely reaches 1.3 GFLOP/s, whereas the performance of `strsm` is as high as 50 GFLOP/s.

Choosing fixed block sizes of 32-by-32 (Fig. 4), for matrices sized 3968×3968, a speed-up factor of 2.5 with respect to the optimized single precision implementation on the CPU is reached, as depicted in Fig. 5

D. Analysis and Computational Complexity

The performance measurements show that in general the GPU version of a linear algebra kernel performs well, i.e. outperforms the CPU, if the kernels are applied to large matrices, whereas in the case of small matrices the performance of the CPU yields better results than that of the GPU implementation. This fact suggests that, if using the GPU as a hardware accelerator, for computations involving small matrices the CPU should be favored.

In the scatter plot in Fig. 6 the number of floating point operations used for the matrix-matrix multiplication has been plotted against the performance rate. The number of operations appears on the horizontal axis, while the performance is plotted on the vertical axis. The data has been taken from Fig. 1 and plotted in a different fashion in order to find the cross-over point where the performance of the GPU surpasses that of the CPU. There are three sets of data: the performance of the matrix-matrix multiplication on the GPU excluding and including data transfers and the CPU's performance of `sgemm`. Each dot in the Figure represents the `sgemm` performance of a matrix-matrix

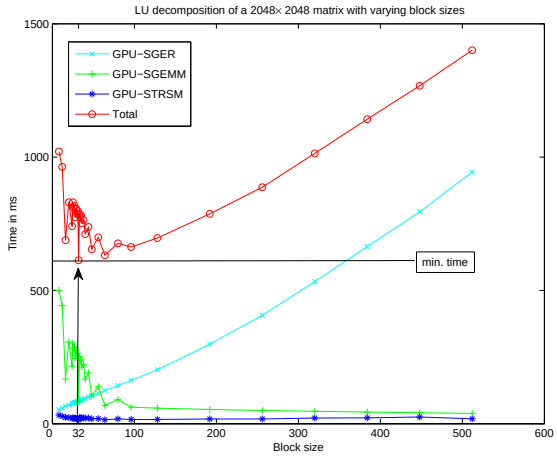

Fig. 4. Influence of block size for a 32-bit GPU factorization with our own CUDA-based `sgetrf` routine for a matrix of size $n = 2000$. The important components of `sgetrf` are: `sger`, `sgemm` and `strsm`.

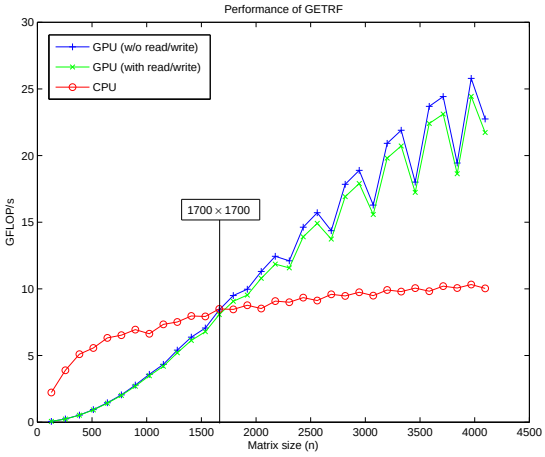

Fig. 5. Performance of dense linear factorization `sgetrf` on CPU and GPU.

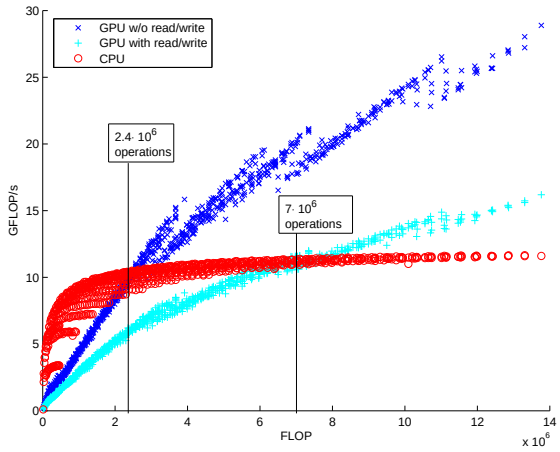

Fig. 6. Number of operations for a matrix-matrix multiplication plotted against the performance of CPU and GPU versions of the computation kernel.

multiplication $\mathbf{A} \cdot \mathbf{B}$, where $\mathbf{A} \in \mathbb{R}^{m \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times n}$ and $m, n \leq 240$ and $k = 120$.

The plot indicates that for our hardware and software configuration the cross-over point is at 7×10^6 floating-point operations if all `sgemm` calls are simply to be substituted by a corresponding function call involving the download of the matrices to the GPU, the multiplication on the GPU and the retrieval of the result. If a more code-invasive approach is chosen and the data happens to be already present in GPU memory at the time of the multiplication, the lower threshold value of 2.4×10^6 operations could be chosen for optimal results.

The cross-over point for `strsm` lies below the matrix size of 150×150.

Fig. 5 suggests that the CUBLAS enhanced version of `sgetrf` should only be used if the matrices become as large as 1600×1600. In order to further improve the performance of the dense solver, a proper CUDA kernel should be implemented and optimized.

IV. General-Purpose Sparse Direct Linear Solvers on the GPU

In this section we concentrate on enhancing the parallel direct solver PARDISO [4], [5], [6] with GPU code – specifically by using the CUBLAS library. PARDISO is a sparse direct solver for large systems of linear equations, which has been developed at the University of Basel and is part of the Intel Math Kernel Library [4]. According to [18] it is currently among the fastest direct solver available for sparse linear systems.

PARDISO uses a supernodal approach to the matrix factorization. A *supernode* is collection of adjacent matrix columns such that the sparsity pattern below the diagonal block is identical in each column. In order to arrive at a matrix structure partitioned into supernodes, a permutation matrix is applied to the original matrix. The motivation to use supernodes is twofold: firstly, since supernodes could be considered as dense matrices that scattered into the matrix to factor to use highly optimized dense high-performance BLAS Level 3 routines. The second observation is on the algorithmic level. While eliminating variables from a sparse system it is desirable to maintain the sparsity of the system, i.e. to minimize fill-in. One strategy e.g. for sparse matrices is to use the minimal degree algorithm, which picks the variables to eliminate by the degree of the node (i.e. the number of adjacent nodes) corresponding to the variable when the matrix is viewed as the adjacency matrix of a graph. It could be shown that the minimum degree algorithm assigns the same priority to all of the rows – viewed as nodes of the graph – belonging to a supernode.

Once the supernode structure is given by means of a permutation matrix, the basic structure of the supernodal factorization algorithm is as follows:

The algorithm in PARDISO implements a left-looking factorization procedure. The matrices involved in the algorithm are schematically depicted in Fig. 7 and Fig. 8. In fact, the supernodes L_i, U_i are sparse block matrices with hopefully many zero rows. By omitting all the zero rows the blocks could be thought of as dense matrices with which Level-3 linear algebra operations can be performed. The main work consists of three computationally intensive dense linear algebra operations: the matrix-matrix multiplications `d/sgemm` in lines 3 and 4, the LU decomposition `d/sgetrf` in line 6, and two triangular solve with `d/strsm` in lines 8 and 9.

A. GPGPU Strategy I — Sparse Factorization using CUBLAS Kernels

As a minimally invasive approach to enhancing the sparse solver with a GPU, we identified all the computationally intensive linear algebra routines in the solver core schematically outlined in Figure 8, and replaced them by a code switching to the GPU equivalent when the number of floating-point operations exceeds the threshold value of 7×10^6 determined in the previous section. Alone this little effort resulted in a considerable speedup of the numerical solving process An acceleration of up to 7 compared to 64-bit CPU can be achieved using the 32-bit GPU on the CUDA platform.

B. GPGPU Strategy II — Mapping Linear Algebra Kernels to the GPU

The other extreme of strategy I, is to adapt the entire numerical factorization code, i.e. the sparse LU decomposition, to the GPU. In this case, ideally the entire matrix would be transferred to the GPU once and be retrieved by the CPU system once the LU or LDL^T decomposition is completed. This, however, works only for matrices with relatively few non-zero elements in the triangular factors, since the GPU memory of the GeForce 8800 is currently limited to 768 MB. A solution to this problem is to develop a hybrid strategy that transfers data to the GPU only if it is currently needed and keeps it on the GPU as long as possible for reuse. Recall from Fig. 7 and Fig. 8 and that a supernode is updated by the supernodes to its left, which have potentially already been loaded from the CPU. When running out of memory it should be examined whether supernodes currently loaded into GPU memory could be truncated (in the updating process not the entire supernodes are used) or if certain supernodes have to be discarded. Preliminary tests have shown that this strategy could further speed up the solver considerably. The implementation, however, is still work in progress.

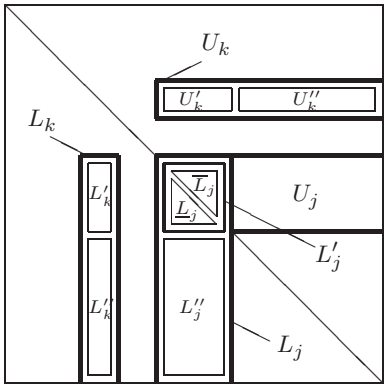

Fig. 7. A step in the left-looking factoring algorithm used by PARDISO to factor structurally symmetric matrices: The supernodes L_j and U_j receive updates from L_k and U_k while being factored.

for $j = 0$ *to # supernodes* **do**
  for $k = 0$ *to* $j - 1$ **do**
    $L_j \leftarrow L_j - L_k \cdot U'_k$
    $U_j \leftarrow U_j - U''_k \cdot L'_k$
  end
  $L'_j \leftarrow$ LU decomposition of L'_j with partial pivoting
  apply the pivots to U_j
  solve $X \cdot \overline{L}_j = L'_j$ for X, $L'_j \leftarrow X$
  solve $X \cdot \underline{L}_j = U_j$ for X, $U_j \leftarrow X$
end

Fig. 8. Supernodal LU decomposition for non-symmetric matrices.

C. GPU/CPU Sparse Factorization for Non-Symmetric Matrices

This section gives an overview of the non-symmetric matrices that are used for the numerical experiments. Some general information about the matrices is given in Table I. Those marked with a (SD) in the column "source" are from semiconductor device simulation. Others, labeled with (UF) are from a public sparse matrix collection [19] and all other matrices are either from automobile crash simulation (AF) or electromagnetic wave simulation (EM). The Table lists the number of unknowns in the matrix, the number of nonzero elements and the number of floating-point operations in GFlops to factor the matrix.

TABLE I
GENERAL INFORMATIONS AND STATISTICS OF THE NONSYMMETRIC MATRICES USED IN THE NUMERICAL EXPERIMENTS.

	name	unknowns	elements	GFlops	source
1	iis-para-19	155,924	8,374,204	444	(SD)
2	iis-para-14	155,924	8,374,204	444	(SD)
3	para-10	155,924	2,094,873	336	(UF)
4	barrier2-9	115,625	2,158,759	198	(UF)
5	S-1300	7,800	40,560,000	123	(AF)
6	747-200k	198,098	3,043,006	72	(EM)
7	747-400k	402,902	6,231,526	246	(EM)

In all following numerical experiments we used the following architectures:

- an Intel Pentium 4 CPU with 3.40 GHz and 2MB L2 Cache,
- an NVIDIA GeForce 8800 GPU.
- the Intel MKL library Version 9.1.

The computational results are shown in the left graphic of Fig. 9. The Figure shows the GFlop/s rate on 64-bit CPU, 32-bit CPU and 32-bit CPU using the GPU strategy I. An acceleration of up to 4 can be achieved using the 32-bit GPU on the CUDA platform.

D. GPU/CPU Sparse Factorization for Symmetric Indefinite Matrices

This section gives an overview of the symmetric indefinite matrices that are used for the numerical experiments and information about the matrices is given in Table II. Those markeded with an (FL) in the column "source" are from fluid dynamics, those labeled with (UF) are from the public sparse matrix collection [19] and those labeled with (CP) arise in computational physics.

TABLE II
GENERAL INFORMATIONS AND STATISTICS OF THE SYMMETRIC INDEFINITE MATRICES USED IN THE NUMERICAL EXPERIMENTS.

	name	unknowns	elements	GFlops	source
1	stokes1	307,995	10,900,171	3,956	(FL)
2	stokes2	505,940	8,609,590	1,805	(FL)
3	ldoor	952,203	23,737,339	125	(UF)
4	crankseg_1	52,804	5,333,507	49	(UF)
5	cop300k_kkt_b	373,990	4,950,222	404	(FL)
6	anderson-50	125,000	500,000	882	(CP)
7	af_shell9	504,855	9,046,865	71	(UF)
8	bmw3_2	227,362	5,757,996	51	(UF)
9	anderson-80	512,000	2,048,000	6,780	(CP)

The computational results are shown in the right graphic of Fig. 9. The Figure shows the GFlop/s rate on 64-bit CPU, 32-bit CPU and 32-bit CPU using the GPU strategy I. An acceleration of up to 7 can be achieved using the 32-bit GPU on the CUDA platform.

V. GENERAL-PURPOSE NONLINEAR INTERIOR-POINT OPTIMIZATION ON A GPU

Nonlinear programming is the problem of optimizing a nonlinear nonconvex objective function to satisfying a set of nonlinear constraints, either of equalities or of inequalities. The standard form is to minimize an objective to find a local solution of the optimization problem

$$\min_{x \in \mathbb{R}^n} \quad f(x) \tag{1a}$$
$$\text{s.t.} \quad c(x) = 0 \tag{1b}$$
$$x \geq 0. \tag{1c}$$

The nonlinear programming problem is given by an objective function $f : \mathbb{R}^n \to \mathbb{R}$ and constraint functions $c : \mathbb{R}^n \to \mathbb{R}^m$, which are both assumed to be twice continuously differentiable. For simplicity in the notation we assume without loss of generality that all variables have only a lower bound.

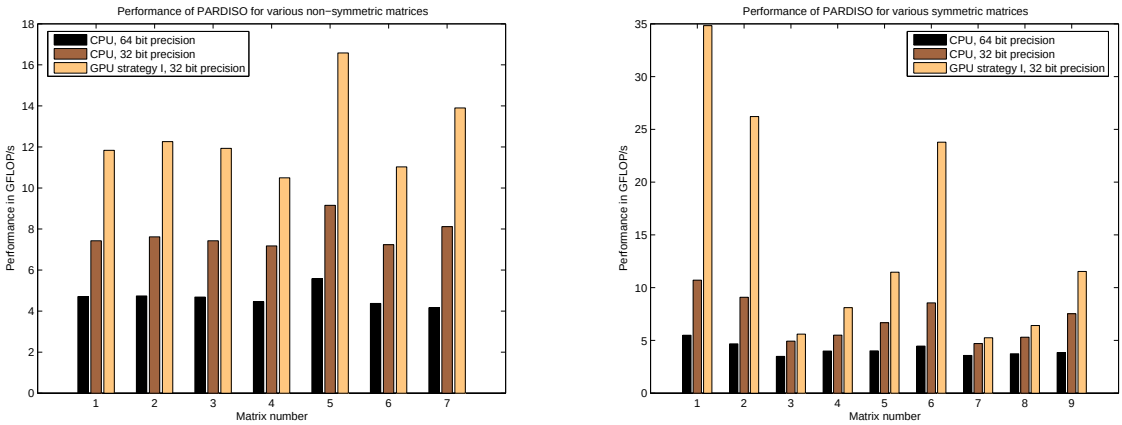

Fig. 9. Performance in GFlops/s for sparse factorization for nonsymmetric (left) and symmetric indefinite matrices (right) on 64-bit CPU, 32-bit CPU and 32-bit GPU using GPU strategy I.

If we use interior-point methods, a sequence of corresponding barrier problems

$$\min_{x\in\mathbb{R}^n} \quad \varphi_\mu(x) = f(x) - \mu\sum_{i=1}^{n}\ln(x^{(i)}) \tag{2a}$$

$$\text{s.t.} \quad c(x) = 0, \tag{2b}$$

is solved to increasingly tighter tolerances, while the barrier parameter μ is driven to zero. Eventually, a series of large-scale symmetric indefinite Karush-Kuhn-Tucker (KKT) linear systems

$$\begin{bmatrix} W_k+\delta_x I & A_k \\ A_k^T & -\delta_c I \end{bmatrix}\begin{pmatrix} \Delta x_k \\ \Delta\lambda_k \end{pmatrix} = -\begin{pmatrix} \nabla\varphi_\mu(x_k)+A_k\lambda_k \\ c(x_k) \end{pmatrix}, \tag{3}$$

have to be solved during the optimization process. Here $A_k = \nabla c(x_k)$ denotes the gradient of the constraints and W_k denotes the Hessian of the Lagrangian function for (1) with respect to x.

In the following, we will use graphics processing units to solve the series of symmetric indefinite KKT matrices that arise in interior-point optimization as implemented in IPOPT [11], which is a primal-dual interior point software package for large-scale nonlinear programming. We will use a 64-bit CPU, a 32-bit CPU and a 32-bit GPU version of the sparse direct solver PARDISO [5], [6], which is one of the sparse direct solvers integrated in IPOPT, to factor the KKT matrices.

As a large-scale nonlinear programming example we choose a nonlinear PDE-constrained optimization problem with Neumann boundary conditions. The domain $\Omega = (0,1)\times(0,1)\times(0,1)$ is represented by a three-dimensional cube and the goal is to compute the optimal boundary control $u(x)$ and state $y(x)$ with respect to $x = (x_1, x_2, x_3)$ that minimizes the objective function

$$f(y,u) = \frac{1}{2}\int_\Omega (y(x)-y_t(x))^2\,dx + \frac{\alpha}{2}\int_\Omega (u(x)-u_t(x))^2\,dx \tag{4}$$

with a Tikhonov regularization of $\alpha = 0.01$. The constraints are

TABLE III
SIZE OF NONLINEAR PROGRAMMING PROBLEM FOR THE 3D PDE-CONSTRAINED OPTIMIZATION PROBLEM WITH BOUNDARY CONTROL AS A FUNCTION OF THE DISCRETIZATION PARAMETER N.

Problem Size	Variables with upper bounds	Variables with lower/upper bounds	Variables with equality constraints
N	N^3	$6N^2$	N^3
$N=20$	8,000	2,400	8,000
$N=30$	27,000	5,400	27,000
$N=40$	64,000	9,600	64,000

defined by the following nonlinear elliptic operator:

$$\begin{aligned} &\text{on } \Omega: \\ &-\Delta y(x) - y(x) + y(x)^3 = 0 \\ &y(x) \le 2.7 \\ &y_d(x) = 2 - 2(x_1(x_1-1) \\ &\qquad +x_2(x_2-1)+x_3(x_3-1)) \\ &\text{on } \Gamma: \\ &\partial_\nu\, y(x) = u(x) \\ &1.8 \le u(x) \le 2.5 \\ &u_d(x) \equiv 0 \end{aligned} \tag{5}$$

These equations represent a simplified Ginzburg-Landau model for super-conductivity in the absence of internal magnetic fields and the state y represents the wave function [20]. We will use second-order finite-difference approximation to discretize Eqn. 4 and Eqn. 5 and the size of the nonlinear programming problem as a function of the discretization parameter N is shown in Table III.

Figure 10 shows the performance of our optimized GPU parallel sparse direct solver PARDISO for various discretization sizes N in comparison to a 32-bit and 64-bit CPU version. Is is clearly visible that for larger-scale optimization problems can be achieved up to a factor of 6.5 while at the same time computing the correct local solution of the optimization problem.

VI. CONCLUSION

We have reported on our experience with using graphics processing units as fast co-processors for general-purpose sparse

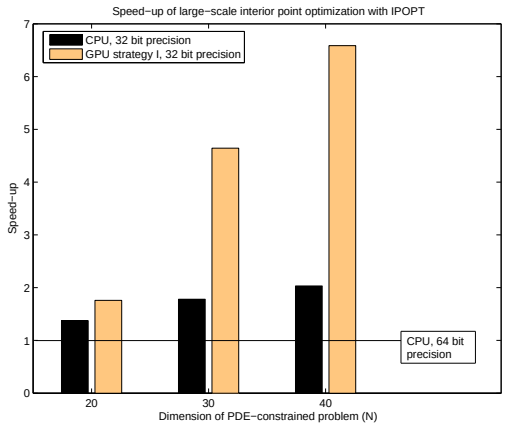

Fig. 10. Nonlinear interior-point optimization speedup against 64-bit CPU with 32-bit CPU and 32-bit GPU using GPU strategy I.

matrix building blocks. We have shown that a minimally invasive approach to instrumenting a direct solver for large sparse systems of linear equations already results in considerable speed-up. The approach consisted in identifying and replacing computationally intensive operations by their GPU counterparts, which, compared to a CPU, typically perform better by one order of magnitude. We have applied the GPGPU-enhanced KKT solver to a nonlinear PDE-constrained optimization problem and have seen that for increasing problem dimension our optimization problem example achieved a speed-up up to a factor 6.5 compared to the original 64 bit CPU implementation. This demonstrates that current commodity GPUs are powerful, yet inexpensive devices that can act as fast numerical co-processors for sparse matrix scientific applications.

REFERENCES

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware." Computer Graphics Forum, Blackwell Publishing, March 2007, pp. 80–113 (34).

[2] "AMD Stream Processor: http://ati.amd.com/products/streamprocessor/index.html."

[3] "The Cell project at IBM Research: http://www.research.ibm.com/cell/."

[4] "Intel Math Kernel Library 9.1 — Sparse Solvers: http://www.intel.com/cd/software/products/asmo-na/eng/266853.htm."

[5] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *Journal of Future Generation Computer Systems*, vol. 20, no. 3, pp. 475–487, 2004.

[6] ——, "On fast factorization pivoting methods for symmetric indefinite systems," *Electr. Trans. Num. Anal.*, vol. 23, no. 1, pp. 158–179, 2006.

[7] E. Rothberg and A. Gupta, "Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations," in *Supercomputing '90*. ACM-IEEE, 1990.

[8] E. Ng and B. Peyton, "Block sparse Cholesky algorithms on advanced uniprocessor computers," *SIAM Journal on Scientific Computing*, vol. 14, pp. 1034–1056, 1993.

[9] J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 1–28, 1990.

[10] T. Davis, *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[11] A. Wächter and L. T. Biegler, "On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006.

[12] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware." ACM/IEEE SC 2005 Conference (SC'05), 2005, p. 3.

[13] J. Kurzak and J. Dongarra, "Implementation of the Mixed-Precision High Performance LINPACK Benchmark on the CELL Processor," University of Tennessee Computer Science, Tech. Rep. UT-CS-06-580, LAPACK Working Note 177), September 2006.

[14] D. Göddeke, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 22, no. 4, pp. 221–256, Aug. 2007.

[15] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 908–916, 2003.

[16] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*. New York, NY, USA: ACM Press, 2005, p. 171.

[17] S. W. Williams, L. Oliker, R. Vuduc, K. Yelick, J. Demmel, and J. Shalf, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms." Proceedings of the Supercomputing '07, Nov. 2007.

[18] N. Gould, Y. Hu, and J. Scott, "A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, 2007.

[19] T. Davis, *University of Florida Sparse Matrix Collection*, University of Florida, Gainesville, http://www.cise.ufl.edu/davis/sparse/.

[20] K. Ito and K. Kunisch, "Augmented Lagrangian-SQP methods for nonlinear optimal control problems of tracking type," *SIAM J. Optimization*, vol. 6, pp. 96–125, 1996.

Matthias Christen received his M.S. degree in Mathematics from the University of Basel, Switzerland, in 2006. Currently he is a Ph.D. student in Computer Science at the University of Basel.

His research interests are in computational science with emphasis on simulation and optimization, as well as high-performance computing.

Olaf Schenk received his M.S. degree in applied mathematics and computer science from the University of Karlsruhe, Germany in 1996 and the Ph.D. degree in technical sciences from the Swiss Federal Institute of Technology (ETH), Zürich, Switzerland in 2000. Since 2001 he is a Research Associate at the Computer Science Department of the University in Basel, Switzerland. His general research interests are in high-performance computing and computational science. He is in particular interested in the solution of large-scale problems which involves information and communication technologies on high-performance computing architectures.

Helmar Burkhart is a Computer Science Professor at the University of Basel since 1987. He received a diploma in computer science from the University of Stuttgart, Germany, and a PdD degree and Venia Legendi (Habilitation) from the Swiss Federal Institute of Technology (ETH) Zurich, Switzerland. He have held several positions such as President of the Swiss Informatics Society SI / Swiss Chapter of the ACM (1990-92), member of the expert group Swiss Priority Programme in Informatics Research (1991-96), and cofounder and board member of the SPEEDUP association. His research interests include parallel and distributed processing, web technologies, and e-learning.