Evaluation and FPGA Implementation of Sparse Linear Solvers for Video Processing Applications

Pierre Greisen, Marian Runo, Patrice Guillet, Simon Heinzle, Aljoscha Smolic, Hubert Kaeslin, Markus Gross

***Abstract*—Sparse linear systems are commonly used in video processing applications, such as edge-aware filtering or video retargeting. Due to the 2D nature of images, the involved problem sizes are large and thus solving such systems is computationally challenging. In this work, we address sparse linear solvers for real-time video applications. We investigate several solver techniques, discuss hardware trade-offs, and provide FPGA architectures and implementation results of a Cholesky direct solver and of an iterative BiCGSTAB solver. The FPGA implementations solve 32K×32K matrices at up to 50 fps and outperform software implementations by at least one order of magnitude.**

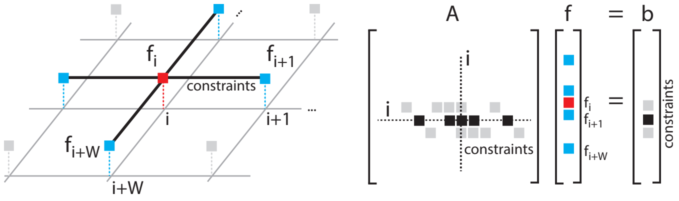

Fig. 1. Discrete-grid problems in video processing. Left: image structure with grid positions i and unknowns f_i. Right: the corresponding matrix system $Af = b$. Constraints are represented by the black bars and black rectangles.

I. Introduction

Many algorithms in computer vision and video processing boil down to finding a solution to a large-scale sparse linear system. The corresponding matrices typically have non-zero elements only on the main diagonal and a small set of off-diagonals. Examples of visual computing problems that encounter this particular sparse form of matrix are image domain warping (IDW) applications such as video retargeting [1], stereo mapping and stereo to multiview conversion [2]; computational photography problems [3] such as high-dynamic range compression and edge-aware filtering; and computer vision problems such as in-painting [4].

Although a multitude of algorithms for solving general linear systems have been reported, the application to real-time video processing has not been addressed thoroughly. The main difficulty lies in the involved problem size, resulting in a huge number of floating-point operations (FLOPs), as well as in huge memory and bandwidth requirements. While these linear systems can often be solved on lower resolution discretization grids without noticeable quality loss, the current trend towards ever higher frame-rates and image resolutions poses significant challenges on solving such systems in real-time.

In this work, we address FPGA architectures of sparse linear systems for computer vision and video processing. Common solver techniques are revisited regarding computational efficiency at the example of IDW applications. To achieve high computational power, we design custom FPGA architectures for an iterative solver (bi-conjugate gradient stabilized (BiCGSTAB)) and a direct solver (CHOLESKY). We compare the two FPGA implementation results and discuss the general trade-offs of iterative and direct solvers on FPGAs in terms of hardware resources, memory bandwidth, and on-chip storage requirements. Furthermore, we compare our FPGA implementations to software implementations. In contrast to programmable hardware (CPUs, GPUs), our dedicated hardware architectures are more energy-efficient and can achieve very high resource utilization, since the hardware resources can be matched to the specific algorithm.

Related FPGA architectures discuss solvers for application-independent sparse matrices with considerably lower dimensions. Morris et al. [5] use a Jacobi solver, which does not converge for our applications. Sun et al. [6] use a direct Cholesky decomposition with a mixed-precision format to reduce computations, which does not scale with problem size and can therefore not be applied to large-scale video processing applications. In addition, they perform the final solve step on a CPU and finally also apply an iterative solver step to compensate for precision losses.

II. Linear Systems in Video Processing

A large class of image processing algorithms can be formulated as energy minimization problem [7]

$$\min_f(E(f)) = \min_f \left(E_{\text{data}}(f) + E_{\text{smooth}}(f)\right) \tag{1}$$

where $E(f)$ is a quadratic energy functional and f an unknown function, discretized on a 2D grid. The energy expression $E(f)$ originates from quadratic constraints on known sampling points ($E_{\text{data}}(f)$) and from smoothness constraints that propagate the known information through the image ($E_{\text{smooth}}(f)$). For example, in IDW applications, f describes a spatially-varying deformation grid that has some known grid positions and requires an interpolation in between these positions. Other applications include high-dynamic range compression (tone mapping), optical flow, disparity estimation, and edge-aware filtering [3]. The solution to (1) is obtained by solving a linear system obtained from the application-specific data and smoothness constraints, introduced in the following.

A. Constraints

In the following, consider a 2D grid of size $W \times H$ and a linearized 1D grid index i, i.e., a point i is neighbored by points $i+1$ and $i+W$ (see Fig. 1). *Data constraints* enforce function values at specific locations

$$\forall i: \qquad C_{\mathrm{d}}^i := s_i(f_i - p_i) = 0, \tag{2}$$

where s_i are the constraint weights and p_i are the required function values. Typically, the majority of grid points does not have a data constraint ($s_i = 0$). *Smoothness constraints* try to propagate these properties to the rest of the image by defining the relative behavior with respect to neighboring function values

$$\begin{aligned}\forall i: \qquad & C_{\mathrm{s},x}^i := s_i^x(f_{i+1} - f_i - d_i^x) = 0 \\ & C_{\mathrm{s},y}^i := s_i^y(f_{i+W} - f_i - d_i^y) = 0.\end{aligned} \tag{3}$$

The parameters d_i^x and d_i^y are the smoothness constraint values and s_i^x, s_i^y are weights indicating the relative importance of the constraints. Since the number of constraints is typically larger than the number of unknowns, the constraints are squared and summed up to form the energy expression (1)

$$E(f) = \sum_i (C_{\mathrm{d}}^i)^2 + \sum_i (C_{\mathrm{s,x}}^i)^2 + (C_{\mathrm{s,y}}^i)^2. \tag{4}$$

Minimizing this energy expression yields a least-squares solution that approximatively satisfies all constraints.

B. Linear system

In the following we show that finding the global minimum of (4) can be obtained by solving a linear system. To simplify notation, we use the placeholder C_γ to denote any of the constraints C_d, $C_{s,x}$, $C_{s,y}$. First, we arrange the constraints into $C_\gamma^i := [A_\gamma f - b_\gamma]_i$, where A_γ and b_γ are a matrix and a vector, respectively. The energy expressions can then be reformulated as

$$E_\gamma(f) := \sum_i (C_\gamma^i)^2 = ||A_\gamma f - b_\gamma||^2. \tag{5}$$

The global minimum is achieved if $d/df E(f) = 0$; with

$$d/df E_\gamma(f) = 2A_\gamma^T(A_\gamma f - b_\gamma)$$

we obtain the final linear system:

$$(\sum_\gamma A_\gamma^T A_\gamma) f = \sum_\gamma A_\gamma^T b_\gamma.$$

In practice, one avoids to first construct A_γ and b_γ and calculate the products and transpositions. Rather, by noting that $A_\gamma^T b_\gamma = -d/df E_\gamma(f)|_{f=0}$ and $A_\gamma^T A_\gamma = d^2/df^2 E_\gamma(f)$, one can deduce expressions for $A_\gamma^T A_\gamma$ and $A_\gamma^T b_\gamma$ directly by analytically deriving the constraints. The combination of smoothness and data energies leads to a symmetric (non-strictly) diagonally dominant (SDD) matrix. The matrix is highly sparse and contains at most five non-zero entries per row: the diagonal plus four off-diagonals (Fig. 1). Note that, when downsampling data constraints onto a lower-dimensional grid, four additional diagonal neighbors appear (grey points in Fig. 1), which results in a similar matrix structure where each of the two outmost off-diagonals are surrounded by two additional diagonals, i.e., eight off-diagonals in total.

In video applications, we have an additional temporal dimension, leading to a temporal constraint relating $f_i(t)$ and $f_i(t-1)$, where t indicates the temporal dimension. Since $f_i(t-1)$ is a known constant at time instance t, temporal constraints are analogous to data constraints.

C. Linear Systems for IDW Applications

One example for an IDW application is video retargeting, which is concerned with content-aware aspect ratio resizing. In video retargeting, the smoothness constraint tries to retain the aspect ratio in visually important regions, whereas the distortions are moved to visually unimportant regions. The weights s_i are thus describing a visual importance (saliency) map [8]. The unknown values f_i describe the new pixel coordinates. In order to reduce the computational complexity, the problem is usually solved on a lower grid resolution instead of solving the problem for each pixel individually. Hence, problem sizes of 1/10 of the image dimensions are realistic (e.g., 190×110≈20k problem size for full HD). Further constraints can include line and edge constraints, temporal constraints, or, for stereo applications, disparity constraints (see [1], [2] for details).

III. Sparse Linear Solvers

There exists a variety of algorithms for solving linear systems [9], [10]. Selecting the best solver algorithm depends on the matrix structure as well as on different trade-offs such as computational complexity, memory bottlenecks, convergence properties, and numerical behavior. In the following, we summarize and compare some of the most widely used algorithms, with a particular focus on dedicated hardware and linear systems for video processing.

A. Classification

1) Direct methods: Direct solvers apply a matrix decomposition technique such as Gaussian elimination or LU/Cholesky/QR decomposition to calculate an exact solution. Unfortunately, the run-time is prohibitively large for general large-scale problems, since the complexity is in the order of $\mathcal{O}(n^3)$ for a quadratic $n \times n$ matrix. Also, memory requirements and numeric stability pose significant challenges for general large-scale problems.

Sparse matrices do not significantly reduce the complexity of direct solvers due to so-called *fill-ins*: zero elements in the initial matrix are replaced by non-zeros in the decomposed matrix. However, for the particular class of band matrices in our application, the fill-ins only appear in between the main diagonal and the outermost side diagonal. The computational complexity then reduces to $\mathcal{O}(nW^2)$, where W is the grid width and hence the offset to the outermost diagonal.

2) Iterative methods: Iterative methods provide an approximate solution to the linear system and reduce the approximation error in each iteration if the system is converging. Each iteration is computed at much lower complexity than direct solvers. Furthermore, there is no fill-in issue since the initial matrix structure is left unchanged during the iteration process. The most simple iterative method, the Jacobi method, has computational complexity $\mathcal{O}(n)$ per iteration, however, it shows

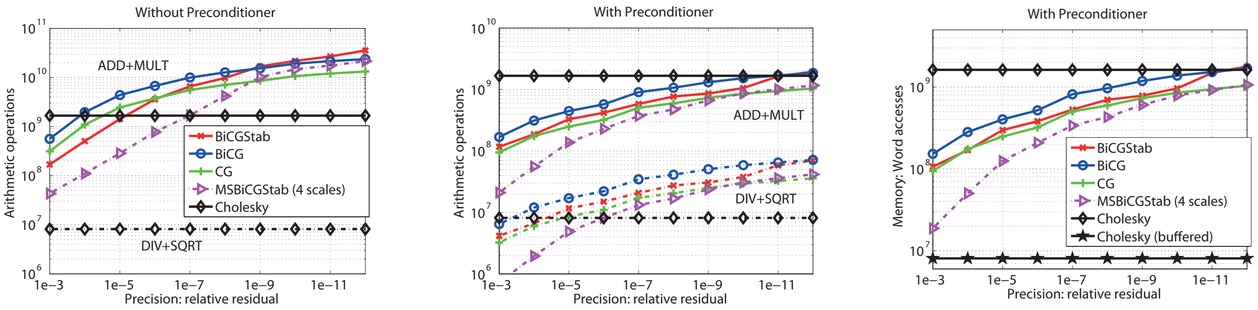

Fig. 2. Arithmetic operations for a multiview synthesis problem without (left) and with (center) diagonal pre-conditioning, and (right) memory accesses. The relative residual is defined as $||Ax - b||/||b||$. The operations are split into addition/multiplications (solid lines) and division/square-root (dotted lines).

very slow or even no convergence. More advanced iterative methods, such as the Krylov-subspace methods, show much better convergence but also higher computational complexity per iteration.

The condition number of matrix A directly impacts the speed of the iteration convergence, and often a pre-conditioner is applied to A to decrease its condition number. However, choosing a good pre-conditioner is a hard task [9] and very specific to the particular problem.

B. Evaluation

Fig. 2 compares the number of arithmetic operations and memory accesses of the most common solvers [9], [10] in the context of IDW applications. For the iterative methods, BICG always performs slightly worse than BiCGSTAB, whereas CG is slightly more efficient in terms of operations. The multi-scale (MS) approach considerably improves all iterative methods, as illustrated by BiCGSTAB for example. Note that Jacobi did not converge for our problem. The inverse diagonal pre-conditioner (PRE) noticeably reduces the computational burden. More advanced pre-conditioners can further reduce the number of iterations but at the price of more complex pre-conditioning architectures.

The banded Cholesky decomposition is computationally more challenging than the iterative methods, but it remains in the same order of magnitude for low error tolerances. One big advantage of the banded Cholesky decomposition regarding hardware efficiency is its data locality: iterative methods require all available data in each iteration, direct methods sequentially work through the matrix once. This makes it possible to devise a local buffering architecture which significantly reduces external bandwidth for the direct method (see Fig. 2), which is not possible for iterative methods.

IV. Iterative Solver FPGA Architecture

In this section a hardware architecture for MS-PRE-BiCGSTAB is discussed (BiCGSTAB shows better convergence than CG, in general [9]). The BiCGSTAB algorithm is summarized in Fig. 3 and consists of a sequence of matrix-vector operations, vector additions, and scalar products. The algorithm starts with an initial solution $\mathbf{x}_0$ which can be random, or, for video processing, the solution from a previous frame. The algorithm works on the residual vector $\mathbf{r} = A\mathbf{x} - \mathbf{b}$ and produces several intermediate vectors $\mathbf{p}, \mathbf{p}', \mathbf{v}, \mathbf{y}$ and scalars $\alpha, \beta, \rho, \omega$. See [9] for more details. The pre-conditioner matrix is $A_d = \text{diag}(A)^{-1}$, where $\text{diag}(A)$ is a matrix with only the diagonal elements of A. The multi-scale approach is performed by first solving a similar problem on a lower resolution and then using the upsampled version of the solution as initial solution on the finer grid. The upsampling is computed with bilinear interpolation.

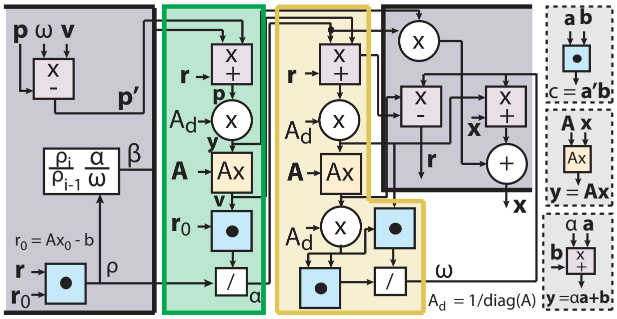

Fig. 3. BiCGSTAB algorithm represented as data flow graph. The algorithm is partitioned into three groups (indicated by the background color) corresponding to the employed hardware resource sharing (see Section IV-A). Regular, small letters denote scalars, bold, small letters denote vectors, and bold, capital letters denote matrices.

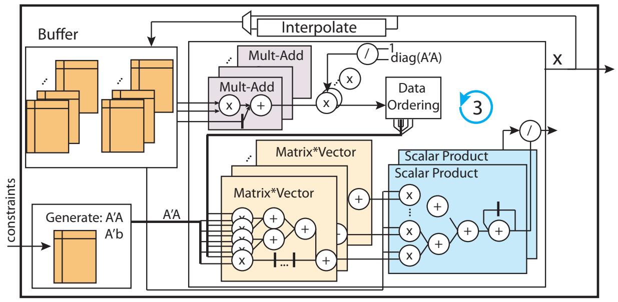

Fig. 4. Simplified top level view of the BiCGSTAB architecture. The architecture implements one third of the data flow graph Fig. 3 (resource sharing) and thus processes one iteration in three phases.

A. Architecture

Fig. 4 provides a block-diagram of the FPGA architecture. Essentially, the architecture is a direct mapping of the data flow graph in Fig. 3, with two key architectural differences. First, the graph is divided into three parts. Each part is mapped onto the same hardware using resource sharing (illustrated in Fig. 3 and Fig. 4). This is motivated by the presence of the scalar products, which form a bottleneck: subsequent operations will stall until the full product has been processed. Together with the iterative nature of the algorithm, pipelining across scalar products is impossible, and the throughput is decreased by the number of scalar products. Our resource-shared architecture retains this throughput but reduces the area by roughly a factor of three and thus increases AT-efficiency.

The second architectural choice is to increase throughput on a finer level by parallelizing the different arithmetic operations (scalar-vector, matrix-vector, scalar products). Further, each matrix-vector unit can process one row-vector multiplication per cycle, since we assume a fixed and small number of entries per row. Matrix A and vector b are generated on the fly from the constraints to reduce memory bandwidth. All other intermediate result vectors need to be buffered due to the scalar product. In total eight vectors (such as $\mathbf{x}$ or $\mathbf{r}$) need to be stored intermediately either on-chip or off-chip.

B. Implementation Aspects

All arithmetic primitives (add, mult, div) are implemented as highly-pipelined floating-point operations. The required resources of the prototype implementation are shown in Table I. Each iteration takes approximately

$$t_{\text{iter}} = \frac{3WH}{f_{\text{clk}}p}[\text{s}]$$

where p is the number of parallel calculation units. The factor '3' comes from the iterative decomposition and f_{clk} is the operating frequency. The nominal memory bandwidth and required memory size are

$$\text{BW} = 32 \cdot 8pf_{\text{clk}}[\text{bit/s}] \qquad \text{size} = 32 \cdot 8WH[\text{bit}]$$

for 32-bit single precision words and 8 different vectors required per iteration. Thus, we have a nominal bandwidth requirement of almost 6 GB/s for $p = 1$ and more than 50 GB/s for $p = 8$, assuming a clock frequency of 200 MHz.

Increasing the amount of parallelism in the matrix-vector and scalar-vector units increases the throughput, but also increases the bandwidth linearly as numerous intermediate results also need to be accessed in parallel. The required memory size only depends on the problem (grid) size. In our current implementation, we opted for on-chip SRAM blocks, as these can provide a very high bandwidth but very limited overall storage space. Alternatively, off-chip memory could be used for larger storage at the price of much lower bandwidth. A hybrid solution (caching) is of limited use in this case, since the complete vector data needs to be addressed linearly in each iteration. External memory bandwidth against internal memory size is therefore the most prominent design decision for iterative solvers, see Fig. 5 for a quantitative illustration.

V. Direct (Cholesky) Solver FPGA Architecture

The Cholesky decomposition generates a triangular matrix L such that $A = LL^T$ (see Fig. 6) for a symmetric and positive definite A system (as introduced in Section II).

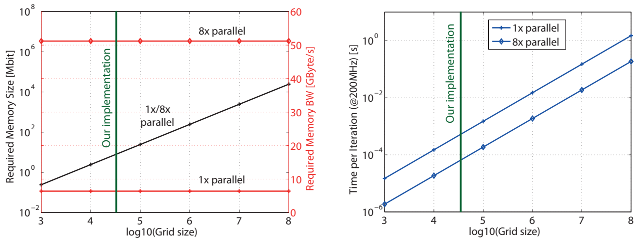

Fig. 5. Left: required memory bandwidth (red) and memory size (black) of BiCGSTAB for different grid sizes for two architectures variants (1x parallel and 8x parallel) Right: corresponding performance.

Defining $y := L^T x$, a new system $Ly = b$ can be efficiently solved using forward substitution. Then x can be solved from $L^T x = y$ with a backward substitution. The generation of L is computationally demanding: for each element in L a scalar product of two n-dimensional vectors is required for matrix sizes $n \times n$. The total complexity is thus $\mathcal{O}(n^2 \cdot n)$. This reduces to $\mathcal{O}(W^2 \cdot n)$ for banded matrices with bandwidth W, since all elements outside the bands remain zero.

A. Architecture

The architecture (Fig. 7) is divided into two blocks: the decomposition and forward substitution block, and the backward substitution. The decomposition and forward substitution share the same hardware. Whenever a new row of L is computed, the forward substitution for this row can already be executed. The backward substitution has the same data flow as the forward substitution, however, it requires the transposed matrix and thus requires to wait for the last element of L. While the same hardware could be shared for both substitution steps, using two separate units allows to work in parallel on different matrices in a time-interleaved way at high AT-efficiency. Due to the transposition, L and $\mathbf{y}$ need to be collected in an external memory before being passed to the backward block.

The datapath consists of a large amount of scalar product operations plus relatively few divisions and square-roots. The scalar-products are designed in a tree structure where the degree of parallelism vs. resource sharing can be adapted based on available FPGA resources. To provide each multiplier with new data in each cycle we incorporate an on-chip L-cache designed such that each multiplier has its own access port. All the previously calculated L and y values are stored in the cache, however, due to the matrix band-structure, the amount of required previous values is limited by the bandwidth W. Since L is a triangular matrix with fixed bandwidth, the cost for calculating one new y value is limited to $W - 1$ old y values and $W - 1$ multiplications/additions, which is negligible compared to the scalar product of the decomposition step.

B. Implementation Aspects

Similar to the iterative solver, we use highly pipelined floating point cores with single precision for the arithmetics. Evaluations revealed that single precision is sufficient to achieve a relative error of less than 10^{-3} for matrix sizes in the order of $10^5 \times 10^5$. In fact, we could use custom floating-point formats below single precision at limited precision penalty;

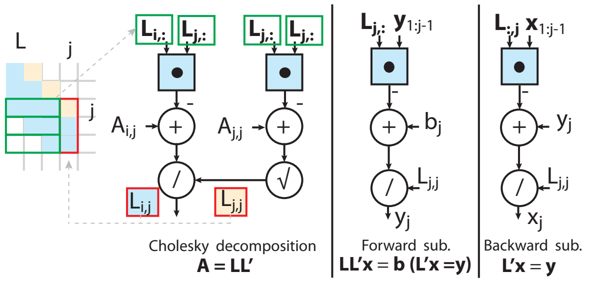

Fig. 6. Cholesky decomposition algorithm represented as data flow graph.

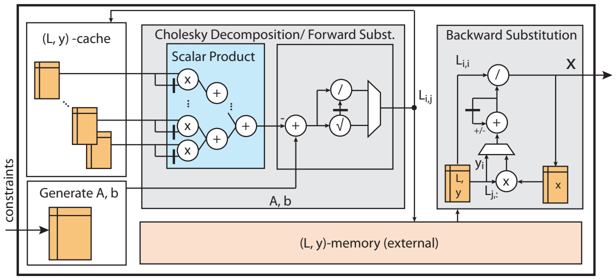

Fig. 7. FPGA Architecture of the Cholesky decomposition.

however, due to fixed-precision multiplier units in FPGAs, there is hardly any gain in resources. Fixed-point formats were found to perform poorly for such large problem sizes. The time per solve with W parallel multipliers in the scalar product is

$$t_{\text{solve}} = 1/f_{\text{clk}} W \cdot (WH) \quad [\text{s}].$$

The L cache is implemented using on-chip FPGA SRAM memory blocks, whereas the full L matrix of size $W \cdot n$ words is stored in an external DDR2 RAM for the matrix transposition. Due to the linear access pattern, the effective bandwidth is close to the nominal bandwidth. In summary

$$\text{BW} = 32 \cdot 2 \cdot f_{\text{clk}} [\text{bit/s}] \qquad \text{size} = 32 \cdot W(WH)[\text{bit}],$$

where the bandwidth is required for reading/writing L and y.

VI. Results and Comparisons

Interestingly, our implementations of iterative and direct solver for banded sparse matrices are similar in terms of hardware resources and performance. A summary of all key figures is given in Table I. However, the main bottlenecks are different: iterative solvers are rather memory-bandwidth limited as vectors of the full problem size need to be accessed in each iteration. The direct solver is computation limited, as only vectors with length equal to the width of the matrix band are required for each Cholesky decomposition step. Note that BiCGSTAB typically needs several 100 iterations (10 to 100 ms/solve) for convergence, depending on the desired precision.

Thus, the most adequate solver type depends on the characteristics of the available hardware. Modern FPGAs with a large amount of on-chip SRAM and high-performance DDR memory interfaces are the platform of choice for iterative solvers, whereas ASICs with very high logic density and much faster operating frequencies would favor direct solvers.

TABLE I
Resource and Performance on Altera Stratix IV 530 GX.

	BiCGSTAB	Cholesky
Problem (A size $n \times n$)	33K $\times$ 33K	33K $\times$ 33K
Entries per row	5	$1 \ldots W + 1$
Possible grids	$WH \leq n$	$WH \leq n$, $W_{\max} = 2^7$
Logic (LUTs)	70k (17%)	100k (24%)
Registers (1-bit FFs)	90k (22%)	165k (41%)
SRAM (on-chip)	11 Mbit (50%)	1 Mbit (5%)
18-bit DSP slices	490 (48%)	548 (54%)
External memory BW	0	$\approx$ 1.5 GB/s
f_{clk} (Worst/Best PTV)	120/203 MHz	186/268 MHz
Performance (W/B PTV)	0.1/0.06 ms/iter	23/15 ms/solve
C-code CPU [11]	35 ms/iter	250 ms/solve
MATLAB	5 ms/iter	200 ms/solve

We also compare performance numbers against MATLAB and a C++ matrix library [11]. As shown in Table I, both FPGA implementations outnumber the CPU implementation by at least one order of magnitude. All timing tests have been performed on an Intel Xeon 3.2 GHz CPU with 24 GB RAM.

VII. Conclusion

Linear solvers for video processing are computationally demanding. The use of dedicated hardware offers at least one order of magnitude speed-up against modern CPU-based computing platforms. More importantly, dedicated solver hardware can be integrated into next generation mobile devices, due to their high energy efficiency (performance per Watt). The use of direct or iterative solver depends on the available hardware resources and the application: iterative solvers require a lot of memory bandwidth and benefit from strong correlations among frames; direct solvers are computation limited and cannot use previous frames to speed-up calculations. A very interesting direction for future work is to investigate recent and upcoming graph-theory based pre-conditioner approaches for dedicated hardware [3], [4].

References

[1] P. Krähenbühl, M. Lang, A. Hornung, and M. Gross, “A system for retargeting of streaming video,” *ACM Trans. Graphics*, vol. 28, no. 5, pp. 1–10, 2009.

[2] A. Smolic *et al.*, “Disparity-aware stereo 3D production tools,” in *Proc. 2011 Conf. Visual Media Production*, Nov. 2011, pp. 165 –173.

[3] D. Krishnan and R. Szeliski, “Multigrid and multilevel preconditioners for computational photography,” *ACM Trans. Graphics*, vol. 30, no. 6, pp. 177:1–177:10, Dec. 2011.

[4] I. Koutis, G. Miller, and D. Tolliver, “Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing,” *Computer Vision and Image Understanding*, 2011.

[5] G. Morris and V. Prasanna, “An FPGA-based floating-point Jacobi iterative solver,” in *Proc. Int. Symp. Parallel Architectures, Algorithms, and Networks*. IEEE, 2005.

[6] J. Sun, G. Peterson, and O. Storaasli, “High-performance mixed-precision linear solver for FPGAs,” *IEEE Trans. Comput.*, vol. 57, no. 12, pp. 1614–1623, 2008.

[7] M. Lang, O. Wang, T. Aydin, A. Smolic, and M. Gross, “Practical temporal consistency for image-based graphics applications,” *ACM Trans. Graphics*, vol. 31, no. 4, pp. 34:1–34:8, 2012.

[8] L. Itti, C. Koch, and E. Niebur, “A model of saliency-based visual attention for rapid scene analysis,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 11, pp. 1254–1259, 1998.

[9] Y. Saad, *Iterative methods for sparse linear systems*. Society for Industrial Mathematics, 2003.

[10] T. Davis, *Direct methods for sparse linear systems*. Society for Industrial Mathematics, 2006, vol. 2.

[11] G. Guennebaud *et al.*, “Eigen v3,” http://eigen.tuxfamily.org, 2010.