# Speeding Up LS-DYNA Implicit with Mixed Precision, Low Rank Approximations, and Accelerators.

Cleve Ashcraft[1], Jason Cong[2], Florent Lopez[1], Roger Grimes[1], Robert Lucas[1], Francois-Henry Rouet[1], and Linghao Song[2]

[1]Ansys
[2]UCLA

## 1 Introduction

The multifrontal method of Duff and Reid [1] dominates the runtime of most LS-DYNA implicit analyses. Its complexity will range from $O(N^{1.5})$ to $O(N^2)$, depending on the model. This paper will give an overview of attempts to reduce the run time of solving large systems of linear equations, both on the host processor as well as with accelerators. Most of what is discussed herein is available today in the development version of LS-DYNA and should be released with R15. Everything discussed herein only applies to our symmetric indefinite solver.

One can mix single and double precision arithmetic, to reduce storage and increase the computational rate. Block low-rank approximation (BLR) takes this further, replacing off-diagonal blocks in the factors of the matrix with the product of two, much smaller matrices. In both cases, the multifrontal direct solver is converted into a high-quality preconditioner for an iterative solver. We will explain the new keyword fields that are used to enable these features.

Meanwhile, Graphics Processing Units (GPUs) are once again of interest. They offer greater arithmetic processing power and memory bandwidth than their host processors. Like GPUs, today's Field Programmable Gate Arrays (FPGAs) also have High Bandwidth Memory (HBM), and hence are candidates for memory bandwidth constrained algorithms such as preconditioned Conjugate Gradients. We have teamed with the University of California Los Angeles's (UCLA) Center for Domain Specific Computing (CDSC) to explore that possibility.

This paper is organized as follows. In section 2, we will discuss how mixed precision can be used to reduce the run time and storage consumed by our default multifrontal linear solver. Second 3 discusses how we can further reduce time and storage by exploiting BLR. Section 4 explains our renewed interest in GPU acceleration, and our progress to date. Section 5 will discuss our joint FPGA research with UCLA. Finally, we will give an overall summary and comment on future directions.

## 2 Mixed precision

LS-DYNA implicit linear solver calculations are performed in double precision, whether real or complex. If one is using a single precision (i.e., i4r4) build of LS-DYNA, the matrix coefficients and right-hand-side vector (RHS) are both promoted to double precision (i4r8), and a double precision solution is returned. Ansys Mechanical (MAPDL) has long exploited mixed precision [2] and was the inspiration for extending that capability to LS-DYNA.

The fourth field of the optional, third line of the *CONTROL_IMPLICIT_SOLVER keyword is the single precision flag, SINGLE. The default value is zero, and both computation and storage are all double precision. If SINGLE is set to one, then computation is double precision, but the factors are stored in single precision. The factors usually dominate the storage, so this

can lead to a significant reduction, often 40% or more. Perhaps enough to keep an LS-DYNA implicit job in-core.

Of course, if the factors are stored in single precision, then their coefficients have 24-bit mantissas, or approximately seven decimal digits of accuracy. The double precision RHS values have 53-bit mantissas, giving them almost sixteen decimal digits of accuracy. For the solution vector to have accuracy approaching that of the RHS, we treat the single precision factors as a high-quality preconditioner for an iterative solver. If the matrix is symmetric and positive definite (SPD), the Conjugate Gradient (CG) method is used. If not, we use Iterative Refinement, an expedient choice that we will revisit in the future. The user can select the convergence tolerance by setting an absolute value and a value relative to the L2 norm of the RHS. The absolute tolerance is the fourth field of the second, optional line of the *CONTROL_IMPLICIT_SOLVER keyword, RPARM1. The relative tolerance is the fifth, RPARM2. Note, if the linear system being solved is poorly conditioned, the iterative algorithm may fail to converge, and then LS-DYNA itself will likely fail as well.

Once one accepts the concept of lower precision storage of the factors, it begs the question, why insist on always performing double-precision arithmetic during the factorization. In a multifrontal linear solver, each frontal matrix is divided into two parts, the fully assembled equations (FA) that are to be eliminated and their Schur complement, or contribution block to their parent (CB). For SPD matrices, the values in the CB are usually of lower magnitude than those in the FA, which include the initial non-zero values of the original sparse matrix. If SINGLE is two, we factor FA equations in double precision, demote the resulting factor to single precision, and then make a left-looking update to the CB with single precision computations.

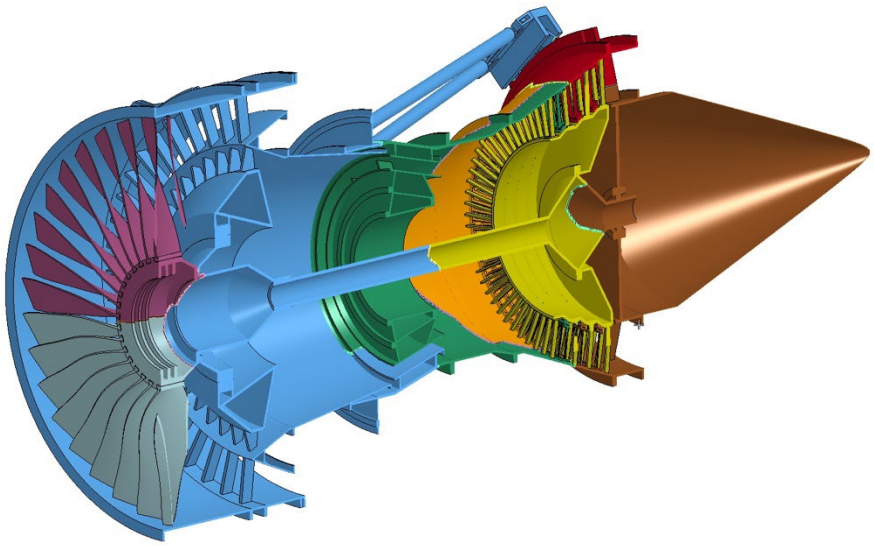

*Fig.1: The Rolls-Royce Large Representative Engine Model*

We illustrate the use of the SINGLE parameter by examining its impact on the 35 million element Rolls-Royce Large Representative Engine model (LREM) [3] depicted in Figure 1. Using LS-GPart for reordering, it takes $1.76 * 10^{15}$ floating-point operations to perform the first of four sparse matrix factorizations and $1.38 * 10^{11}$ words to store the lower triangle of its factors. We ran the LREM on eight nodes of an Ansys compute cluster composed of AMD Naples processors, each with 512GB of memory. Using four MPI ranks per node, limiting the job to three time-steps, and computing entirely with double precision, the job takes 49,377 seconds., 46,043 of which were expended in the four sparse matrix factorizations. Note, this time is the "Imp. LA Factor" time reported in the timing summary at the end of the mes0000 message file

and includes 2,575 seconds for four passes through reordering and symbolic analysis, as well as 283 seconds for four sparse matrix redistributions.

When SINGLE is set to two, the factorization runtime drops from 46,043 to 34,720 seconds. However, overall run time only dropped to 44,875 seconds as the 629 seconds need to perform seven double-precision triangular solves grows to 7851 seconds to perform seven CG iterations. The CG time was dominated by 166 mixed-precision triangular solves.

Storage reduction is the other virtue of using SINGLE. For the 32 MPI ranks, most saw the minimum storage needed for in-core factorization reduced by 35-45%. Worst case, it was reduced by 28%.

## 3 Block low rank approximation

When factoring a sparse matrix, diagonal blocks of the factors need to be full rank. Off-diagonal blocks don't. Therefore, one could imagine using a Singular Value Decomposition (SVD) to probe the rank of an $M$ by $N$ off-diagonal block, and if its rank $r$ is less than half of the minimum of $M$ and $N$, then replace the block with the product of $U$ and $V^T$[4][5]. Storage drops from $M * N$ to $r * (M + N)$. If one can tolerate a loss of accuracy in the off-diagonal blocks, beyond that normally expected due to finite precision and round-off, then the $U$ and $V$ that result from the SVD can be trimmed to only contain only those columns whose singular values exceed a specified threshold, epsilon. As illustrated in Figure 2, BLR allows LS-DYNA to make an explicit trade-off between storage and operations during factorization versus precision in the resulting solution vectors. Just as with mixed precision, the factors of the matrix become a high-quality preconditioner for a subsequent iterative solver such as CG. And, as with mixed precision, the use of BLR is not without risk. If the error tolerance is too high during the factorization, the subsequent iterative solve may take too long to converge, or even fail to do so.

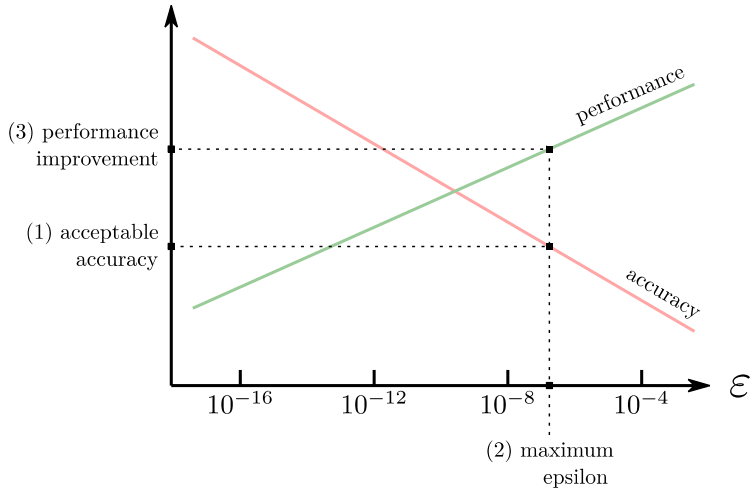


*Fig.2: The figure above illustrates the tradeoff space between factorization performance (and size) versus accuracy, as a function of error tolerance.*

The parameters for engaging BLR are in the *CONTROL_IMPLICIT_SOLVER keyword. SEGLEN, the nominal number of rows and columns in an off-diagonal block that might be subjected to BLR is specified in the seventh field of the optional second line. In our experience,

240 is a good setting. If segments are too small, there is little compression. If they are too large, the compression, whose complexity scales as $O(N^3)$, takes too long. In addition, the nominal panel width in frontal matrices being factored by multiple MPI ranks is 256.

Instead of an SVD, we use rank revealing QR factorization as in our experience, it's faster. The BLR error tolerance, BLRTOL, is the eighth field in the optional second line. The fifth field in the optional third line, BLROPT, specifies which BLR options are to be used. If BLROPT is set to one, then the factorization of the entire matrix will be performed with full precision, and BLR will be applied to the factors of the larger frontal matrices, reducing the storage needed to factor the matrix. Of course, the low-rank compression requires additional operations beyond those needed to factor the matrix, and runtime will suffer.

If BLROPT is two, then the FA columns of large frontal matrices are factored with full precision. Then they are compressed with BLR before their Schur complement is computed. This can reduce the operation count significantly, however the impact on runtime is much smaller. When computing a full-rank update to the CB, we use calls to the matrix-matrix multiplication routines DGEMM and DGEMMT in vendor supplied math libraries (e.g., Intel's MKL). The dimensions of the matrices are hundreds, and throughput approaching 100 GFlop/s per core is expected if dual AVX512 Arithmetic Logic Units (ALUs) are available. When performing low-rank updates, one or two of the dimensions of the calls to DGEMM is much smaller, often less than ten, and throughput drops to O(10) GFlop/s per core. A thirty-fold reduction in operations performed can be coupled with a tenfold reduction in throughput, leading to a speedup of only three. Low rank updates to the CB also introduce more error into the factors, as an update passed from one frontal matrix to its parent now contains more than just round-off error.

If BLROPT is three, then we extend the use of low-rank updates into the FA columns of large frontal matrices. This further reduces run time but also further introduces error in the factors. Finally, if BLROPT is four, then BLR compression is applied to the contribution blocks too. This reduces the real stack, i.e., the primary working storage needed to factor with the multifrontal method. Of course, the additional CB BLR computations increase the run time, and introduce yet another opportunity to add error to the factors.

Figure 3 illustrates the impact of BLR on the 35 million element Rolls-Royce LREM. Because the model ran successfully with SINGLE set to two, that setting is repeated throughout. BLROPT is set to three in an effort to minimize runtime. As in section one above, the data was collected while running with 32 MPI ranks on eight nodes of an AMD Naples cluster. BLRTOL varies from $10^{-11}$ to $10^{-7}$. Even with the lowest setting of BLRTOL, the run time is 71% of that for full rank. The sweet spot is a BLRTOL of $10^{-8}$, where the run time is 59% of full rank. The minimum storage reduction for a processor to stay in-core is 32% of that for full rank. When BLRTOL gets too aggressive, the number of iterations required for the CG solves rises dramatically. When BLRTOL was set to $10^{-6}$ the first CG solve exited after 32,654 seconds. Having run for 1000 iterations, it had only reduced the L2 norm of the residual by six orders of magnitude, not the ten that was requested, and the job was canceled.

If BLROPT is set to four, the worst-case storage reduction is 34%. The additional BLR computations increased the runtime to 35,849 seconds. In our limited experience, the reduction in the working storage is not worth the time expended when BLROPT is four.

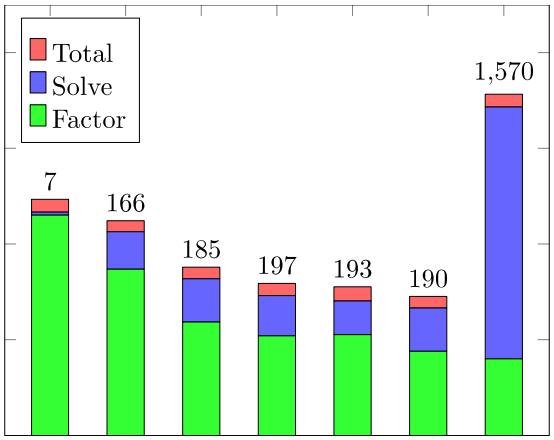


*Fig.3: The result of varying BLRTOL on the Rolls-Royce 35 million element LREM. SINGLE is set to two, except for the full-rank column, labeled FR.*

BLR is applied to one frontal matrix at a time, and its impact on storage and operations varies dramatically. Figures 4 plots the total number of words needed to store the factors at each level of the supernodal elimination tree [6], where leaves of the tree are on the left and the root is on the right. Figure 5 plots the number of floating-point operations performed at each level.

Notice that in both figures, there is no difference in the lowest seven levels of the elimination tree. That is because in our experience, BLR compression is not effective when applied to either small frontal matrices, or those whose coefficients contain a significant fraction of initial values from the matrix. Therefore, we have heuristics to skip over them. Notice also that the top two levels of the elimination tree also show no difference. The reason is that near the root of the tree, the processor count per frontal matrix is high enough that we use our tiled factorization kernels, which are faster. Our tiled factorization kernels do not yet support BLR. The absence of BLR at the lower and upper levels of the elimination tree create Amdahl fractions that limit the overall impact of BLR, both on storage and runtime.

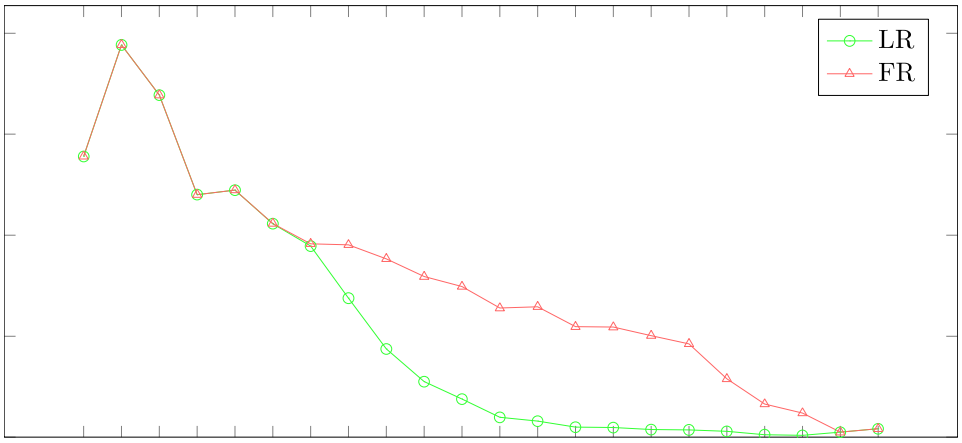


*Fig.4: Words of storage needed to hold the lower triangle of the factors of the 35 million element LREM, at each level in the elimination tree, for both BLR (LR) and full rank (FR) factorizations.*

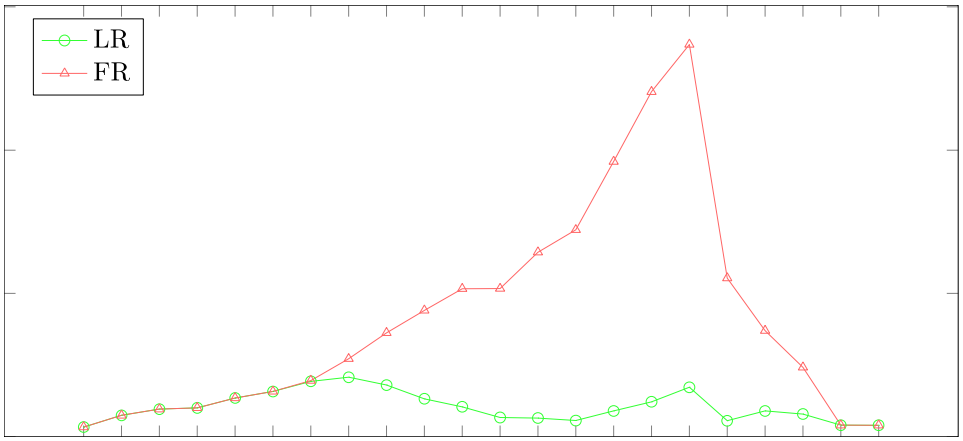


*Fig.5: Operations performed at each level in the elimination tree for both BLR (LR) and full rank (FR) factorizations of the 35 million element LREM.*

One problem with BLR is that the error introduced can change the sign of diagonal entries in the factors of the matrix, and LS-DYNA might think that an SPD matrix was indefinite. If the user is confident that the matrix was in fact positive definite, the user can set SPD, the seventh field of the optional third line of *CONTROL_IMPLICT_SOLVER to one. For the LREM model, SPD had to be set when BLRTOL was $10^{-7}$ and $10^{-6}$.

As with mixed precision, BLR does not work for every model we have tried it on. Sometimes the error tolerance must bet set so small, that it's little different from the round-off error introduced by a full-rank, double precision factorization. Each model is different, and user's must be careful when attempting to trade precision for time and storage.

## 4 Graphics processing units

As depicted in Figure 6, GPUs can perform dense matrix factorization an order of magnitude faster than their general purpose, multi-core hosts. LS-DYNA developers were among the first to try to exploit this for implicit calculations. Our initial attempt to use Nvidia GPUs as accelerators was abandoned when we found that our users' models were too sparse to make the GPUs practical. For example, when factoring the stiffness matrix for a model of a Chevrolet Silverado pickup truck, that had 5,278,379 equations, the root frontal matrix only had 4,413 equations in it.

Today's automotive models are larger, and denser. In particular, the relatively dense battery packs of electric vehicles are creating frontal matrices with over 100,000 equations. Therefore, LS-DYNA developers are once again experimenting with GPUs. The networks connecting today's GPUs to their hosts are much faster than the ones in use a dozen years ago, and kernel launch times are down. That means it's now profitable to factor almost all frontal matrices on a GPU, as long as they fit in its memory. An Nvidia A100 has 80 GBytes of High Bandwidth Memory (HBM), so only the very largest frontal matrices are a challenge.

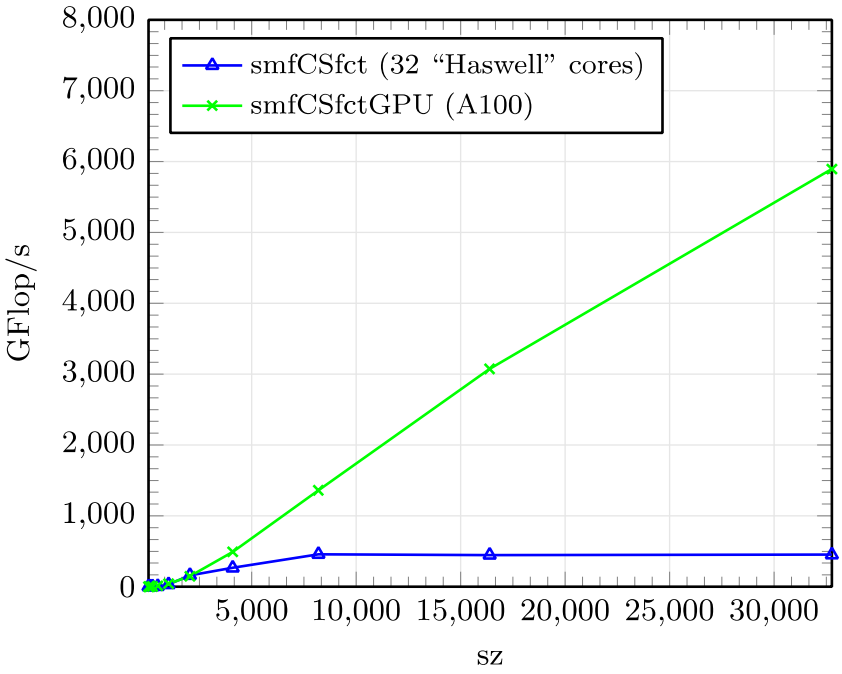


*Fig.6: The relative performance of an Nvidia A100 GPU and 32 Intel Haswell cores when factoring increasingly large dense matrices.*

| | | | Walltime (s) | |
|---|---|---|---|---|
| Mat | flops × $10^9$ | Arch. | CPU Haswell 16 thr. | GPU A100 |
| Yaris_suspension | 1758.01 | | 22.11 | 31.11 |
| Ventricle | 3957.44 | | 21.38 | 8.70 |
| imp2 | 10098.3 | | 55.97 | 21.64 |
| cyl0p5e6 | 32309.9 | | 133.70 | 17.04 |

*Table 1: The relative performance of sparse matrix factorization on an Nvidia A100 compared to that of an Intel Haswell multicore microprocessor.*

Unlike the last time around, we are not only factoring frontal matrices on the GPU, but also stacking their contribution blocks and assembling the frontal matrices as well. This minimizes the time expended communicating with the host. As shown below in Table 1, the results are very encouraging. The three larger models all exhibit significant speedups.

The GPU work described herein is not yet available in the development (DEV) version of LS-DYNA. It still needs to be extended to work with multiple MPI ranks. We anticipate it will be in the R15 release.

## 5 Field programmable gate arrays

Attempts to use FPGAs as computational accelerators date back to the late 1980s [7]. Early FPGA circuits were larger, slower, and more power hungry than equivalent functions on their hosts, and hence tended be used for things like gene sequence alignment, with two-bit operands [8]. Any notion of accelerating floating-point computations would have been ridiculous.

Over the last three decades, Moore's Law has enabled FPGA circuits to get denser and faster, and they can now host processor cores and floating-point ALUs. In addition, today's FPGAs can come with HBM, giving them more memory bandwidth than their hosts. This begs the

question, could an FPGA be used to accelerate a memory-bandwidth limited linear solver, such as CG. Ansys initially teamed with Xilinx to explore this question. Xilinx later directed us to work with UCLA's CDSC. We have chosen to begin with Jacobi (i.e., diagonally) preconditioned CG (JPCG). It has a very simple preconditioner yet is powerful enough to be the default solver for thermal problems in LS-DYNA.

LS-DYNA has an interface for a user-supplier linear equation solver. Reference code is provided with the usermat distribution of LS-DYNA. The userLE_PCG.F90 routine is called when LSOLVR, the first field of the required first line of *CONTROL_IMPLICIT_SOLVER, is set to 90. PCG.c is reference code for the CG linear solver.

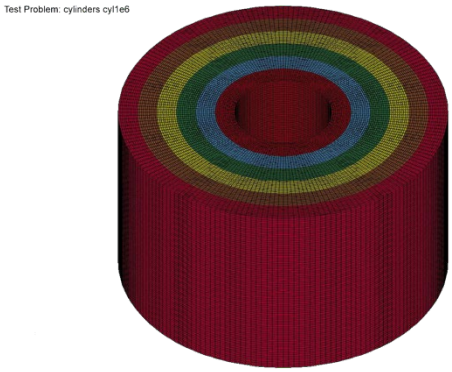


*Fig.7: The Atomic Weapons Establishment's cyl0p5e6 500,000 element nested cylinder benchmark model.*

| | 1 Intel Xeon Platinum 8260L + AVX512 | 8 MPI ranks on an Intel Xeon Platinum 8260L | 1 NVIDIA A100 GPU | 1 Xilinx U55c |
|---|---|---|---|---|
| Memory Bandwidth (GB/s) | 130 | 130 | 2039 | 460 |
| LS-DYNA End-to-end Time (s) | 694 | 103 | 55.5 | 75 |
| FPGA Speedup | 9.25 | 1.37 | 0.74 | 1 |
| Energy (KJ) | 114.51 | 17.00 | 14.45 | 9.78 |
| Cost (KUSD) | 8.8 | 8.8 | 23.8 | 7.4 |
| End-to-end Throughput (GFLOP/s) | 1.82 | 12.27 | 22.78 | 16.86 |
| Throughout / Power (GFLOP/s/KW) | 11.03 | 74.36 | 87.52 | 129.33 |
| Throughout / Cost (GFLOP/s/KUSD) | 0.21 | 1.39 | 0.96 | 2.28 |

*Table 2: The comparison of performance, energy, and efficiency of four platforms.*

UCLA investigated the JPCG algorithm and determined that for a broad range of problems from SuiteSparse [9], an acceptable double precision result could be achieved even when the matrix is stored in single precision, as long as the vectors in the JPCG algorithm were stored in double precision. They then designed the Callipepla dataflow architecture which can sink almost the entire bandwidth supplied by the FPGA's HBM memory [10]. Callipepla was built on top of the Serpens sparse matrix-vector multiplication engine that UCLA had already implemented [11]. They replaced PCG.c with a new C++ function that initializes the FPGA, downloads the matrix and RHS vector to the FPGA, executes the JPCG algorithm, and returns the solution

vector. When called a second time using the same matrix, only the RHS vector needs to be transmitted to the FPGA.

Initial results are very promising. We considered the 500,000 element AWE cyl0p5e6 model, depicted in Figure 7, which is small enough to fit in the FPGA's memory, and runs quickly enough to support multiple experiments. Results are reported in Table 2. The baseline is an Intel Xeon Skylake processor, using one core. We also include results for the same machine, using eight MPI ranks, an Nvidia A100 with a different Intel host, and a Xilinx u55c with an AMD host. The A100 used the same user-supplied linear solver interface as did the FPGA.

In Table 2, we not only report the runtime, but also provide estimates of both the relative costs of the systems and their power consumption, using specifications drawn from the vendors web sites. The A100 GPU achieves the lowest time and highest throughput because the GPU has the highest memory bandwidth among the four platforms. The Xilinx U55c FPGA appears to be the best in terms of cost, energy consumption, and power/cost efficiency.

UCLA's Callipepla FPGA design is open source, and any LS-DYNA user can download and use it. For those doing modest-sized thermal calculations, where JPCG is the default solver, a Xilinx u55c could be a very cost-effective accelerator. How broadly applicable it could be to other implicit calculations is an open question.

## 6 Summary

Solving large, sparse linear systems of equations is usually the computational bottleneck in implicit LS-DYNA jobs. Therefore, Ansys invests a lot of time and energy trying to reduce that. Historically, we have done so by exploiting concurrency with MPI and OpenMP. As seen herein, we are now also exploiting mixed procession, low-rank approximations, and accelerators.

The mixed precision work reported here only exploited two floating point formats, single and double precision. It has been shown [12] that one can further compress storage, using non-standard formats. Furthermore, machine learning has created a demand for lower precision floating point numbers, and vendors now provide hardware support a variety of 16 and 20-bit formats. It may be possible to exploit these to further accelerate LS-DYNA implicit.

Today, BLR is only partially implemented for symmetric indefinite problems, and not at all for unsymmetric factorization. The breadth of its applicability and impact is not yet understood. If users find BLR useful, whether to save time or storage, positive feedback will help encourage us to extend BLR to tiled symmetric frontal matrices and the unsymmetric solver as well.

Given the size of today's models, and the increasingly tight coupling of GPUs to their hosts, Ansys is once again developing the capability of using GPUs as accelerators for symmetric indefinite linear systems. Today, we have written a body of CUDA code that enables an Nvidia GPU to accelerate a sequential or shared memory LS-DYNA implicit calculation. Work to extend that to message passing is ongoing, and we anticipate GPU acceleration will be released in 2024 with R15. We will want to extend that to unsymmetric systems and other manufacturers' GPUs in the years to come.

Meanwhile, FPGAs have matured to a point that they too are credible accelerators. While they can't match GPUs for peak floating-point throughput, our UCLA colleagues have shown

FPGAs to be a cost-effective way to acquire and exploit HBM. Our next steps will be to extend the work to use multiple FPGAs as well as more sophisticated preconditioners. There may also be other memory bandwidth bound parts of LS-DYNA could also be accelerated with FPGAs.

Finally, GPUs and FPGAs are not the only potential accelerators that LS-DYNA could exploit. At last report there were over one hundred projects World-wide to develop specialized computational devices to sate the demand for computing power to underpin Machine Learning. Ansys is beginning to explore that space as well.

## 7 Literature


[1] I Duff, J Reid, “The Multfrontal Solution of Indefinite Sparse Symmetric Linear Equations“, ACM Transactions on Mathematical Software, Vol 9, No 3, pp 302-325, Sept. 1983

[2] Y Liu and J Beisheim, private communication

[3] F Rouet, C Ashcraft, J Dawson, R Grimes, E Guleryuz, S Koric, R Lucas, J Ong, T Simons, T Zhu, “Running jet engine models on thousands of processors with LS-DYNA Implicit“, Proceedings of the 12th European LS-DYNA Conference, 2019

[4] M Bebendorf, “I Why finite element discretizations can be factored by triangular hierarchical matrices“, SIAM Journal on Numerical Analysis 45 (4), 1472-1494

[5] P Amestoy, C Ashcraft, A Buttari, J L’Excellent, C Weisbecker, “Improving Multifrontal Methods by Means of Block Low-Rank Representations“, SIAM Journal on Scientific Computing, 11 June 2015

[6] R Schreiber, “A New Implementation of Sparse Gaussian Elimination“, ACM Transactions on Mathematical Software, 8 (3) 256-276

[7] P Bertin, D Roncin, J Vuillemin, "Introduction to Programmable Active Memories", Technical Report 3, DEC Paris Research Lab, 1989.

[8] M. Gokhale, W Holmes, A Kopser, S Lucas, R Minnich, D Lopresti, D Sweely, "Building and using a highly parallel programmable logic array," in *Computer*, vol. 24, no. 1, pp. 81-89, Jan. 1991, doi:10.1109/2.67197.

[9] T Davis, Yifan Hu, „The University of Florida sparse matrix collection“, ACM Transactions on Mathematical Software, Vol 38, Issue 1, 2011 pp 1:1 – 1:25

[10] L Song, L Guo, S Basalama, Y Chi, R Lucas, J Cong, “Callipepla: Stream Centric Instruction Set and Mixed Precision for Accelerating Conjugate Gradient Solver,” Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 247-258

[11] L Song, Y Chi, L Guo, J Cong, “Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication,” Proceedings of the 59th ACM/IEEE Design Automation Conference, 211-216

[12] P Amestoy, O Boiteau, A Buttari, M Gerest, F Jezequel, J L’Exellent, T Mary, “Mixed precision low-rank approximations and their application to block low-rank LU factorization“, IMA Journal of Numerical Analysis, Vol 43, Issue 4, July 2023, Pages 2198 – 2227