Delayed Evaluation and Runtime Code Generation as a means to Producing High Performance Numerical Software

Francis Russell

October 3, 2006

About the Investigation

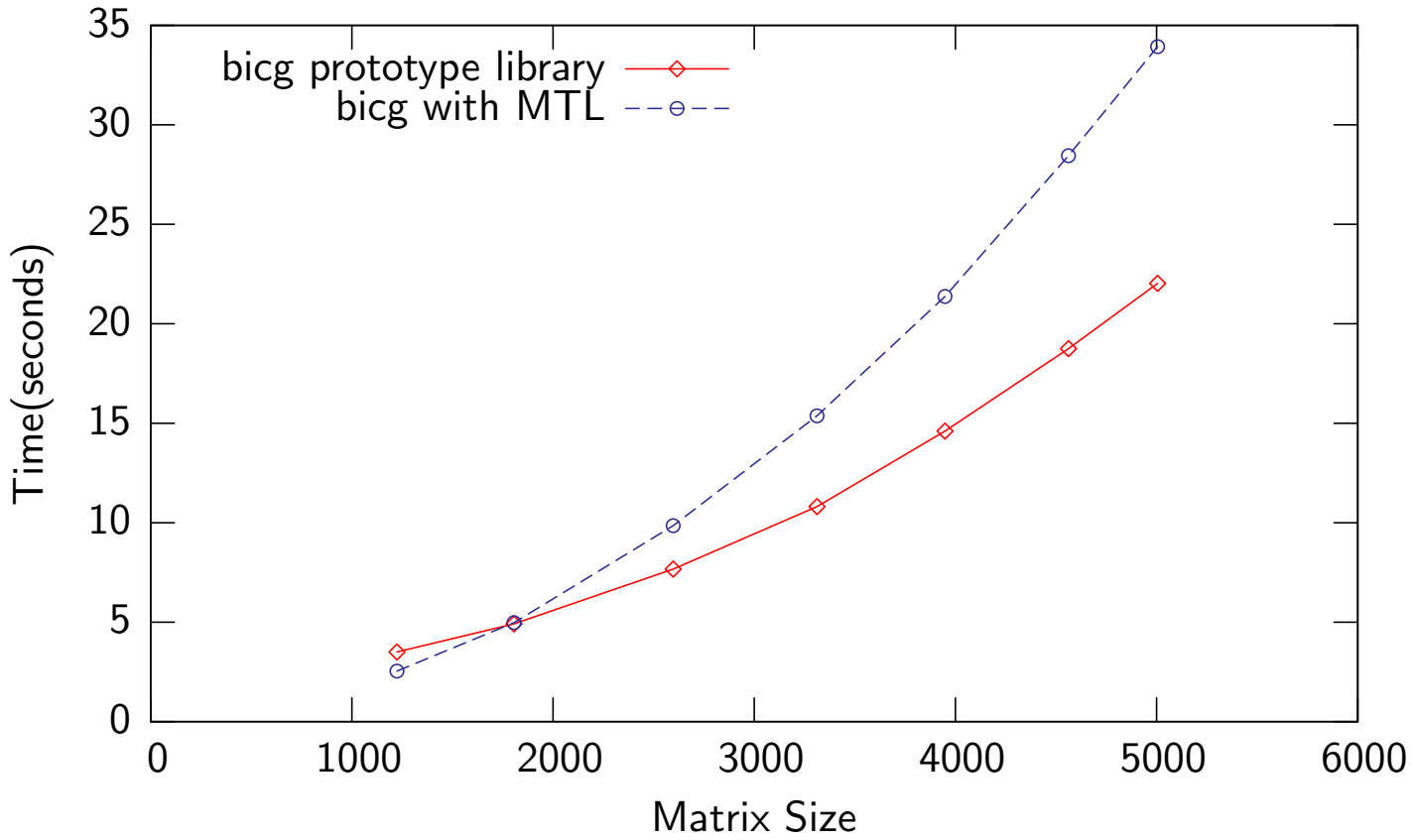
We investigated these techniques with the aim of providing:

- High performance numerical code.
- Object oriented C++ abstractions.

We have adopted a rather radical approach to doing this compared to conventional libraries. We shift work from the application and library's compile time to the application's run time.

How much better can we do?

On one platform[1], we managed to achieve an average 27% speedup across a range of matrix sizes and benchmark applications. 256 iterations of BiConjugate Gradient Solver with prototype library and MTL showing a 50% speedup:



[1] 3.2GHz Hyperthreaded Pentium IV with 2048 KB L2 cache and 1GB RAM

High Performance Maths

Scientists and engineers need high performance maths. The usual solutions include:

Fortran

- First class arrays.
- Easy to optimise.

BLAS

- Routines for basic linear algebra operations.
- Efficient and portable.
- Improving performance well researched.

Key Related Work: The ATLAS Project

ATLAS stands for Automatically Tuned Linear Algebra Software. It was created as part of an ongoing research effort into applying empirical techniques to provide portable performance. ATLAS:

- Supports the BLAS interface.
- Automatically adapts itself to hardware and software.
- Uses code generators to search for the best implementation of different BLAS operations.

The Problem with BLAS

The performance of BLAS/ATLAS comes with a cost:

- Greater complexity for greater performance.
- Lack of abstraction.
- Less understandable code.

What does this do?

void cblas_dgemv(const enum CBLAS_ORDER, const enum CBLAS_TRANSPOSE TransA, const int M, const in N, double alpha, const double* A, const int lda, const double* X, double beta, double* Y, const incY);

The Problem with BLAS

The performance of BLAS/ATLAS comes with a cost:

- Greater complexity for greater performance.
- Lack of abstraction.
- Less understandable code.

What does this do?

void cblas_dgemv(const enum CBLAS_ORDER, const enum CBLAS_TRANSPOSE TransA, const int M, const in N, double alpha, const double* A, const int lda, const double* X, double beta, double* Y, const incY);

$$y = \alpha A^T x + \beta y$$

Enter C++

Using operator overloading in C++ we could express this as:

```
Y = alpha * transpose(A) *  X + beta * y;
```

The problem is, the application of each operator will create a temporary value.

Two numerical libraries for C++, Blitz++ and the Matrix Template library have used the C++ templates system to control expression parsing and compilation.

MTL, the most advanced, has used these techniques to perform optimisations such as loop unrolling and blocking.

Another Approach

This project has investigated another approach to performing high performance numerical computing. A prototype library has been developed using the following techniques:

- Delayed Evaluation.
- Runtime Code Generation.

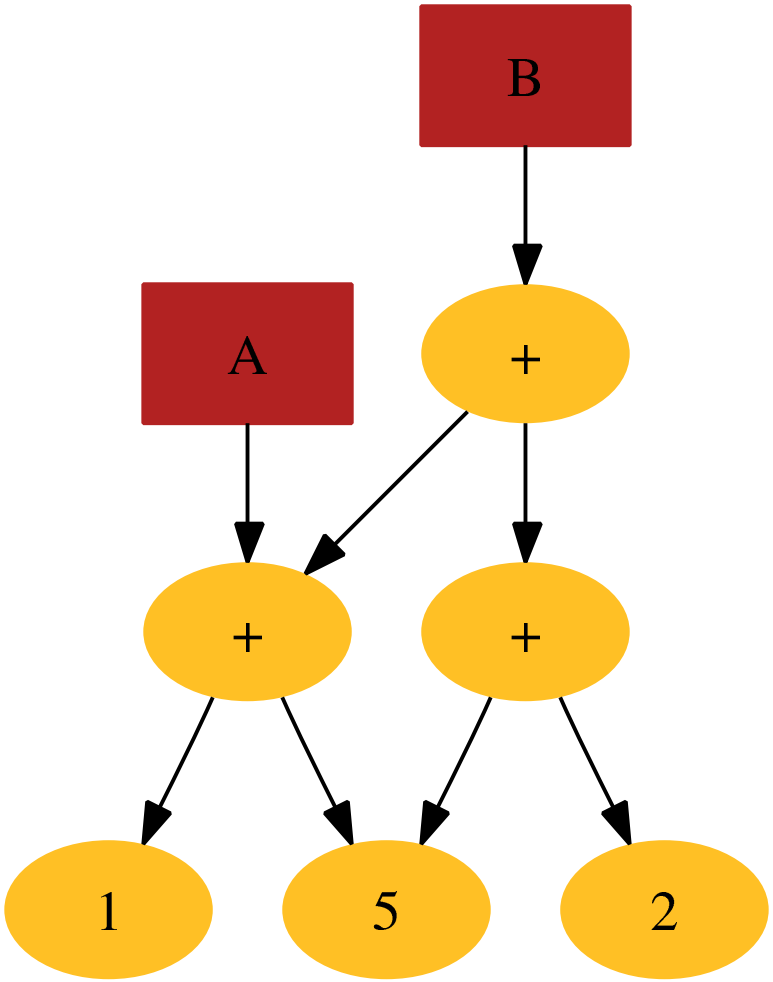
Delayed Evaluation

- Delayed evaluation enables the library to delay the execution of an operation until the result is required. This is called a *force point*.
- Using C++'s abstraction facilities, this can be done with minimal impact on the library's interface.
- Using delayed evaluation, it is possible to collect runtime context information that enables the execution performance of the delayed operations to be improved.
- Here, the print statement is a force point. Delaying evaluation allows us to determine that the expression a+d can be evaluated in a single loop.

```
Vector a, b, c, d, e, f;
a = b + c;
d = e + f;
print(a + d);
```

Delayed Evaluation

Delayed evaluation is implemented using a directed acyclic graph (DAG) of delayed operations.



Runtime Code Generation

- Runtime code generation involves the creation, compilation and execution of code at runtime.
- The code can be specialised using runtime information, improving performance.
- Optimisations can be applied to the generated code.

A loop summing the elements of a vector, could be specialised by vector length.

```
for (int index=0; index<length(vec); index++)
  sum += vec[index];
```

becomes:

```
for (int index=0; index<1803; index++)
  sum += vec[index];
```

The TaskGraph Library

The runtime code generation in the prototype library is done using TaskGraph. TaskGraph enables:

- Code to be constructed using a C-like sub-language.
- Optimisations to be applied to runtime generated code such as loop fusion.
- Compilation of the runtime generated code using GCC or ICC.

Defining a TaskGraph

A TaskGraph to execute a dot product:

```
taskgraph(t)
{
  tParameter(tArrayFromList(float,a,vecSize));
  tParameter(tArrayFromList(float,b,vecSize));
  tParameter(tVar(float, result);
  tVar(int, n);

  tFor(n, 0, vecSize[0]-1) {
    result += a[n] * b[n];
  }
}
```

The code is specialised by the length of the vectors, stored in the array *vecSize*.

The Framework

We now have a framework capable of:

- Delaying numerical operations.
- Generating code at runtime to execute them.
- Specialising generated code using runtime context information.

Investigated Techniques

Four techniques were investigated for improving the performance of the runtime generated code. We investigated the performance of the library with a benchmark suite of *dense linear iterative solvers*:

- Code Caching.
- Loop Fusion.
- Array Contraction.
- Runtime Liveness Analysis.

Code Caching

- It was discovered that almost identical code was being created, compiled and executed during each iteration of the iterative solver.
- Upon evaluation, the delayed expression DAG is converted to another DAG format containing both the high level information about the delayed operations, and information about the generated TaskGraph. The detection and reuse of generated code is performed on this level.
- Detecting repeated delayed expressions is a DAG isomorphism problem.

Simplifying the Isomorphism Problem

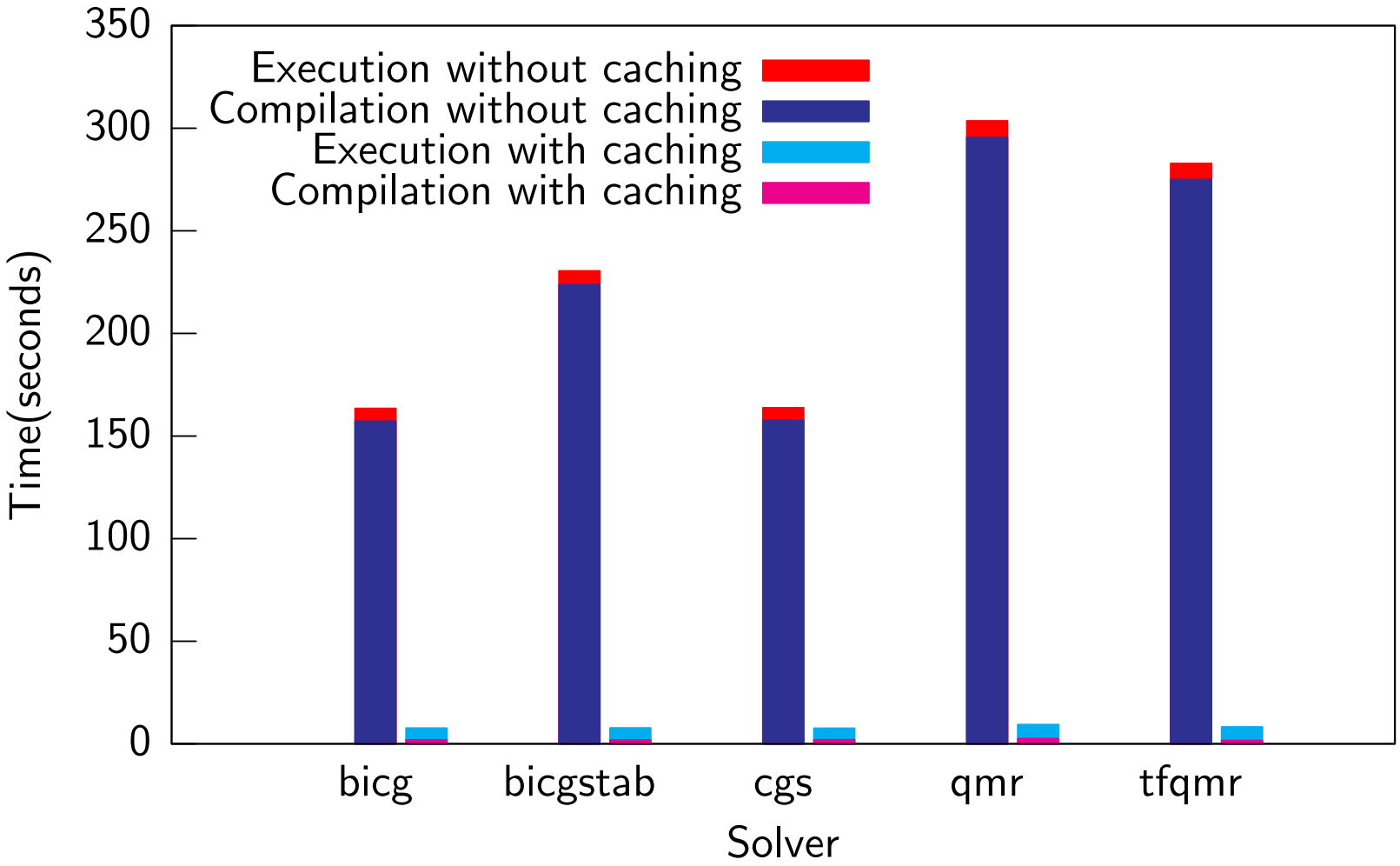
Steps taken to simplifying isomorphism consisted of:

- Graph hashing.
- Flattened DAG matching.

For this to work correctly, the expression DAG must always be flattened in the same order.

Code Caching

256 iterations of each solver for 1806x1806 matrix.



Code Caching

- Speedups for every benchmark.
- Essential for reclaiming performance when code is short running.
- Problem of specialisation versus reuse.
- How useful will it be for other numerical applications?

Loop Fusion

Loop fusion can improve the performance of a program by:

- Reducing loop overhead.
- Improving cache locality.

Before loop fusion:

```
for(int i=0; i<100; i++)
  c[i] = a[i] + b[i];

for(int j=0; j<100; j++)
  e[j] = c[j] + d[j];
```

After loop fusion:

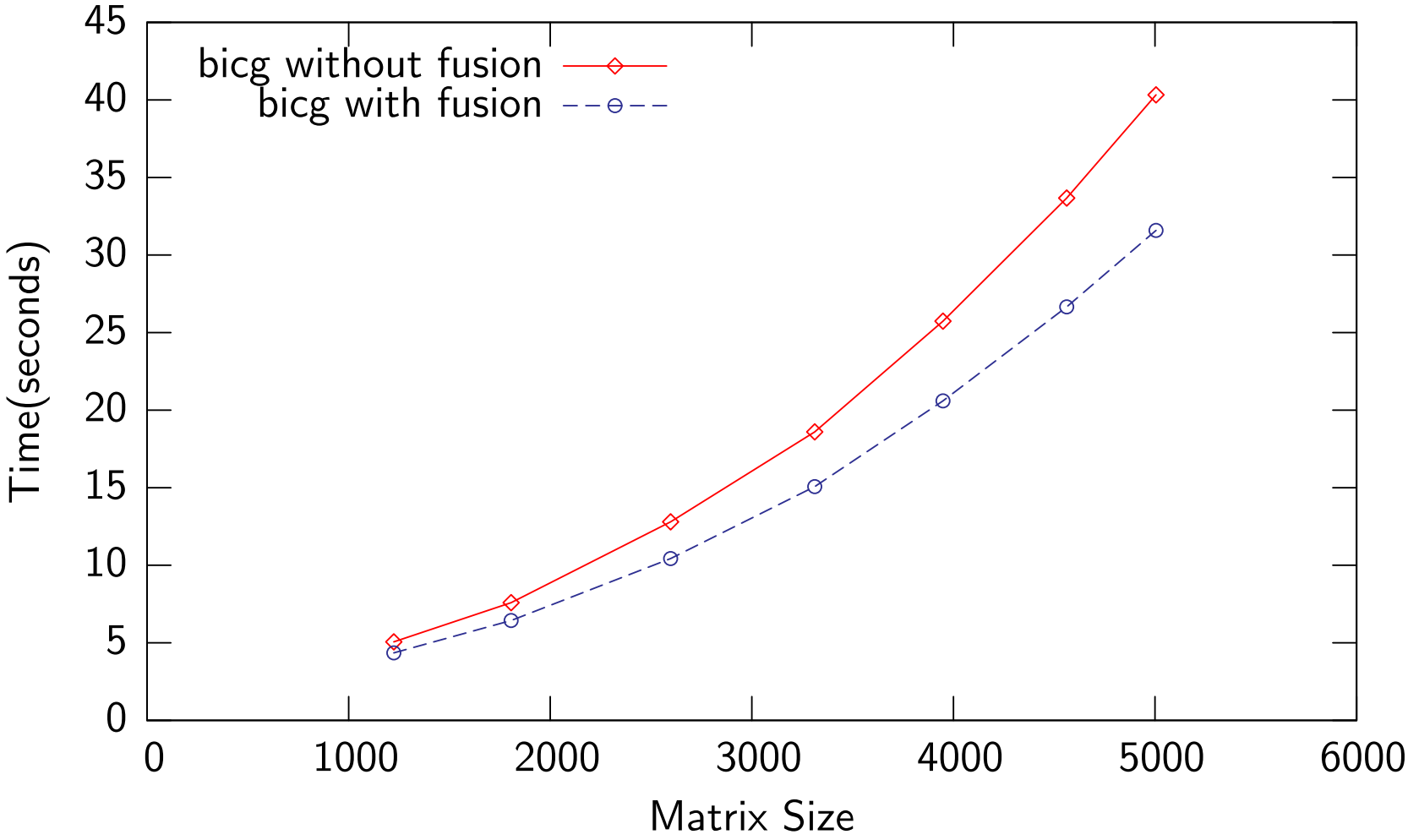
```
for(int i=0; i<100; i++) {
  c[i] = a[i] + b[i];
  e[i] = c[i] + d[i];
}
```

TaskGraph Loop Fusion

- TaskGraph loop fuser had severe limitations.
- I was able to improve the loop fuser to make it more flexible. with regards to the locations of the loops it could fuse and the dependencies between the code fragments involved.
- The improved loop fuser was successful in fusing together multiple loops in all benchmark applications.
- I decided to evaluate operations using the same data together in the hope of obtaining beneficial loop fusions.
- The TaskGraph back-end, SUIF, lacks a full dependence model making it difficult to implement more advanced loop fusion.
- Further development of the loop fuser would allow more flexible positioning of the loops to be fused and statement reordering.
- Even more development would allow the loop fuser to make loop fusion decisions based on cache locality.

Loop Fusion

256 iterations of BiConjugate Gradient Solver[2]



[2] 3.0GHz Hyperthreaded Pentium IV with 512 KB L2 cache and 1GB RAM

Loop Fusion

- Significant speedup on BiConjugate Gradient benchmark.
- No significant performance increases on other benchmark applications.
- Average of 16 loop fusions in commonly executed code.
- Need a good cache locality model to be certain we are choosing useful fusions.
- Most fused operations are vector-vector. In BiConjugate Gradient benchmark, a vector-matrix and transpose matrix-vector multiply have been fused.

Array Contraction

Array contraction allows:

- Memory usage of a program to be reduced.
- Improved cache use.

Array contraction is often facilitated by loop fusion.
Before array contraction:

```
for (int i=0; i<1000; i++) {
  c[i] = a[i] + b[i];
  e[i] = c[i] + d[i];
}
```

After array contraction:

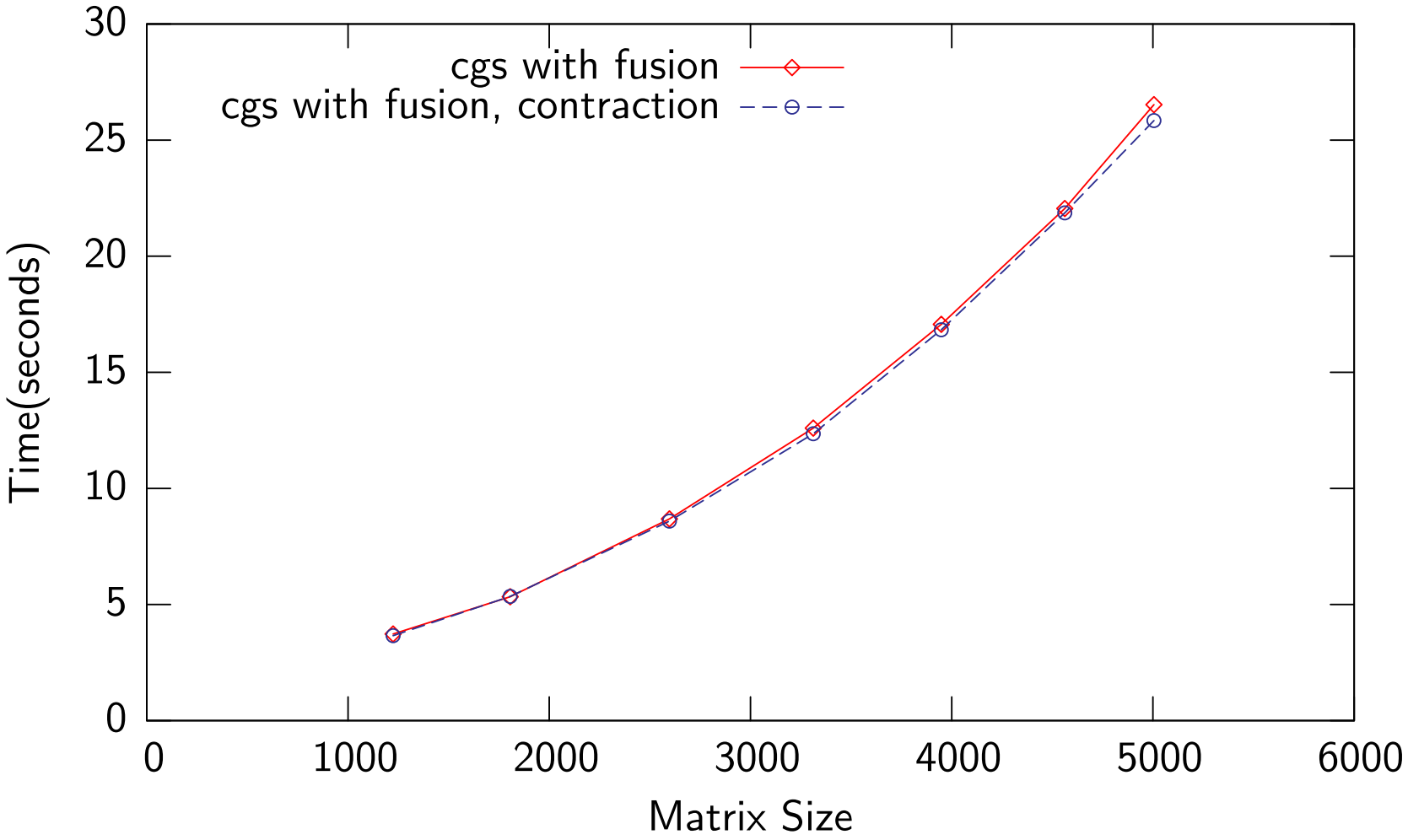
```
for (int i=0; i<1000; i++) {
  c = a[i] + b[i];
  e[i] = c + d[i];
}
```

TaskGraph Array Contraction

- We thought that the Intel C compiler might do array contraction given favourable conditions.
- It didn't.
- I wrote an array contraction pass for SUIF, the TaskGraph back-end.
- It was successful in removing a number of temporary vectors from all the iterative solvers.

Array Contraction

256 iterations of Conjugate Gradient Solver[3].



[3] 3.2Hz Pentium Hyperthreaded IV with 2048 KB L2 cache and 1GB RAM

Array Contraction

- Effect of array contraction isn't noticeable in the benchmarks presented.
- Inspecting transformed code showed an average of 6 array contractions in commonly executed code.
- Array contraction is working, so most likely its effect is being overshadowed by the cost of the matrix-vector multiply.

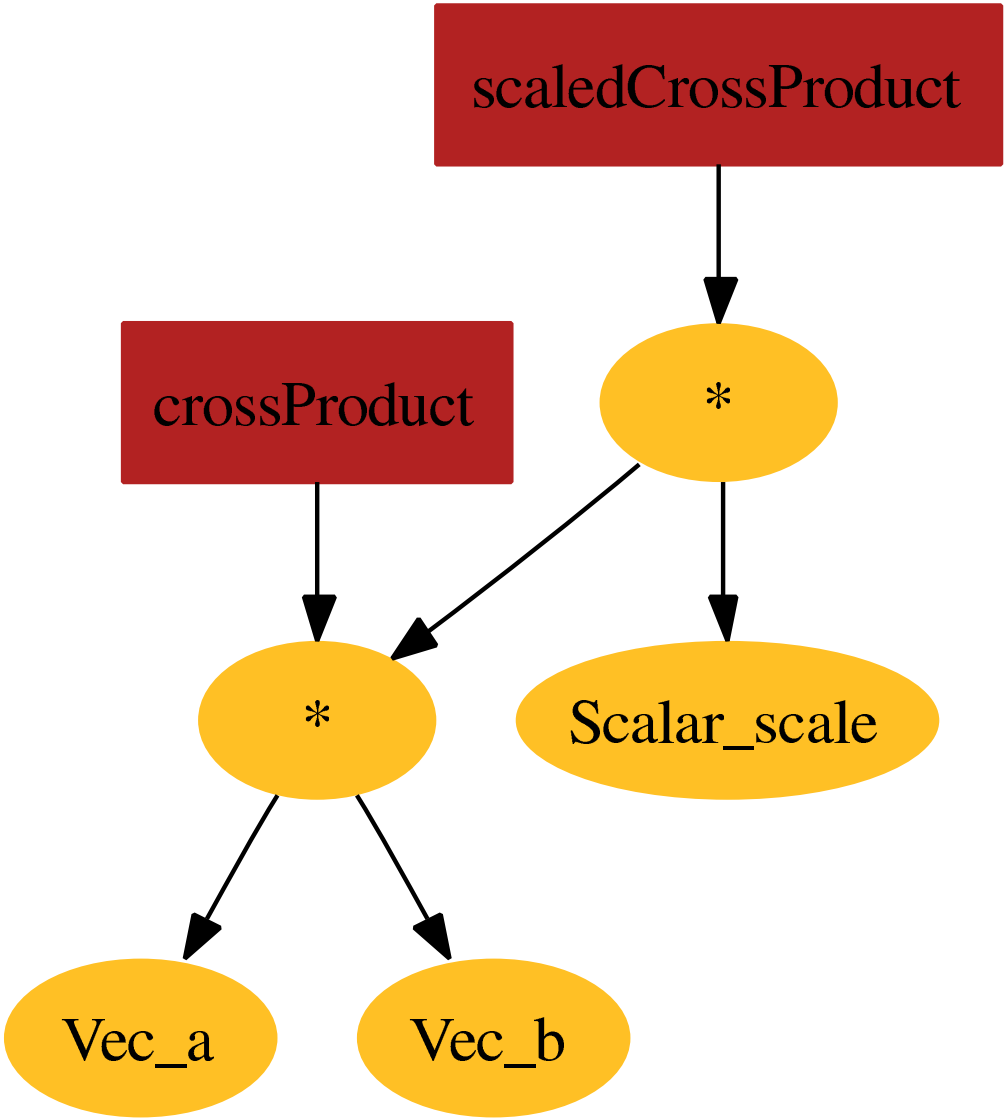
Runtime Liveness Analysis

Consider the following function:

```
void printScaledDotProduct(Vector a, Vector b,
                           Scalar scale)
{
  Vector crossProduct = a * b;
  Vector scaledCrossProduct = crossProduct * scale;
  print(scaledCrossProduct);
}
```

The value *crossProduct* is never used directly. When *scaledCrossProduct* is evaluated, there is no need to keep the result of the cross product. Unfortunately, as there is a handle still pointing to it, it is impossible to reason about whether it can be optimised away.

Runtime Liveness Analysis



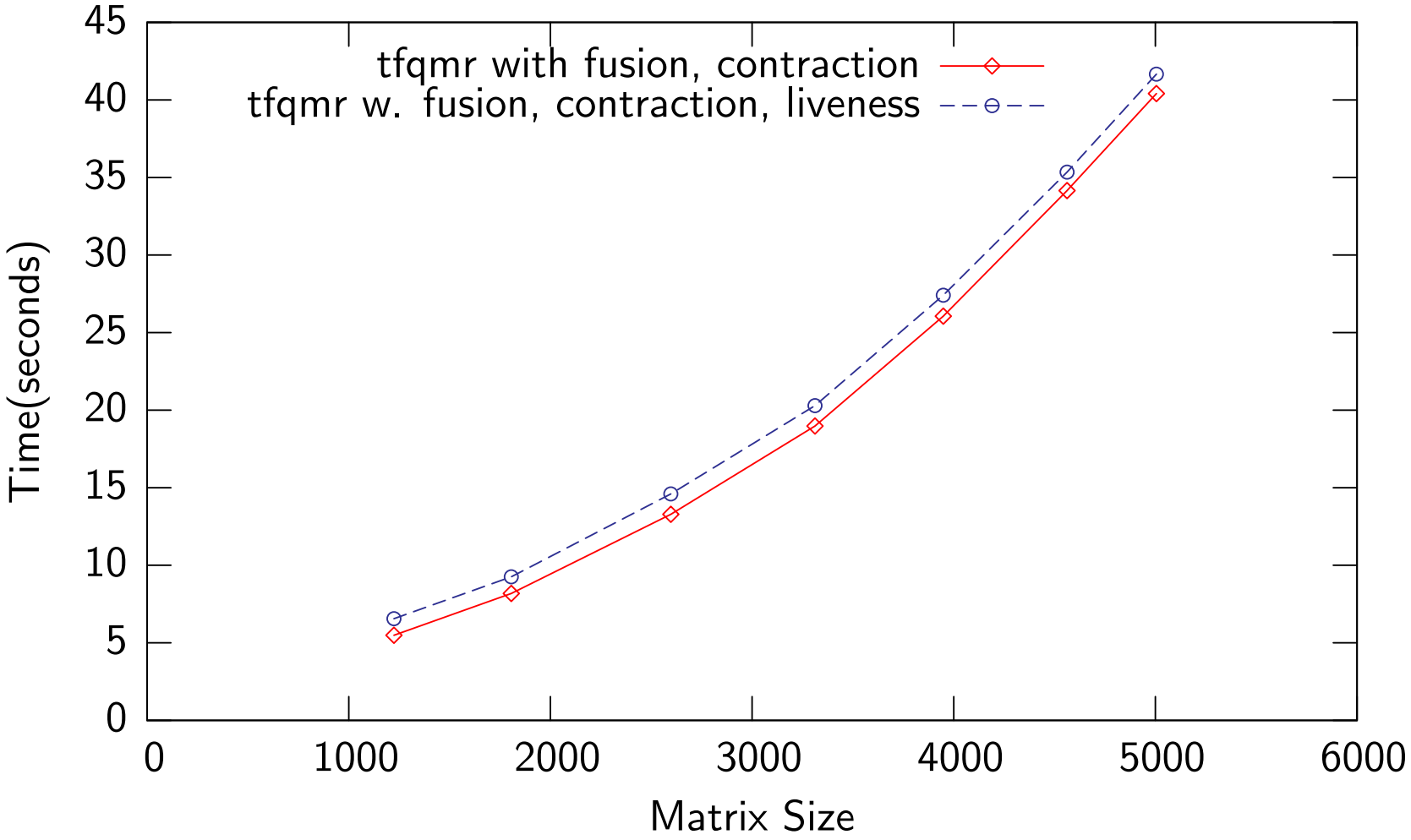
Runtime Liveness Analysis

The prototype library's runtime analysis:

- Builds a profiling DAG mirroring the structure of each expression DAG evaluated.
- The profiling DAG attaches monitors to the expression DAG to obtain liveness information.
- The next time an expression DAG is built matching a profiling DAG, the profiling DAG is used to set flags on each expression DAG node guessing whether that node's values will be used directly.
- Values believed to be dead can be allocated locally to the runtime generated code. If they are not dead, their value must be computed again.

Runtime Liveness Analysis

256 iterations of Transpose Free Quasi-Minimal Residual[4].



[4]3.0GHz Hyperthreaded Pentium IV with 512 KB L2 cache and 1GB RAM

Runtime Liveness Analysis

- In the benchmarks, runtime liveness analysis provides a constant overhead rather than a gain.
- The constant overhead is due to the extra compilations caused by the liveness analysis mechanism changing its mind about what values are live and dead.

Solver	Total compiler invocations without liveness analysis	Total compiler invocations with liveness analysis
bicg	9	10
bicgstab	10	12
cgs	9	11
qmr	12	16
tfqmr	9	14

A Demonstration

The prototype library in action and a look at some runtime generated code:

Evaluating the Library

The library was evaluated on the following two architectures:

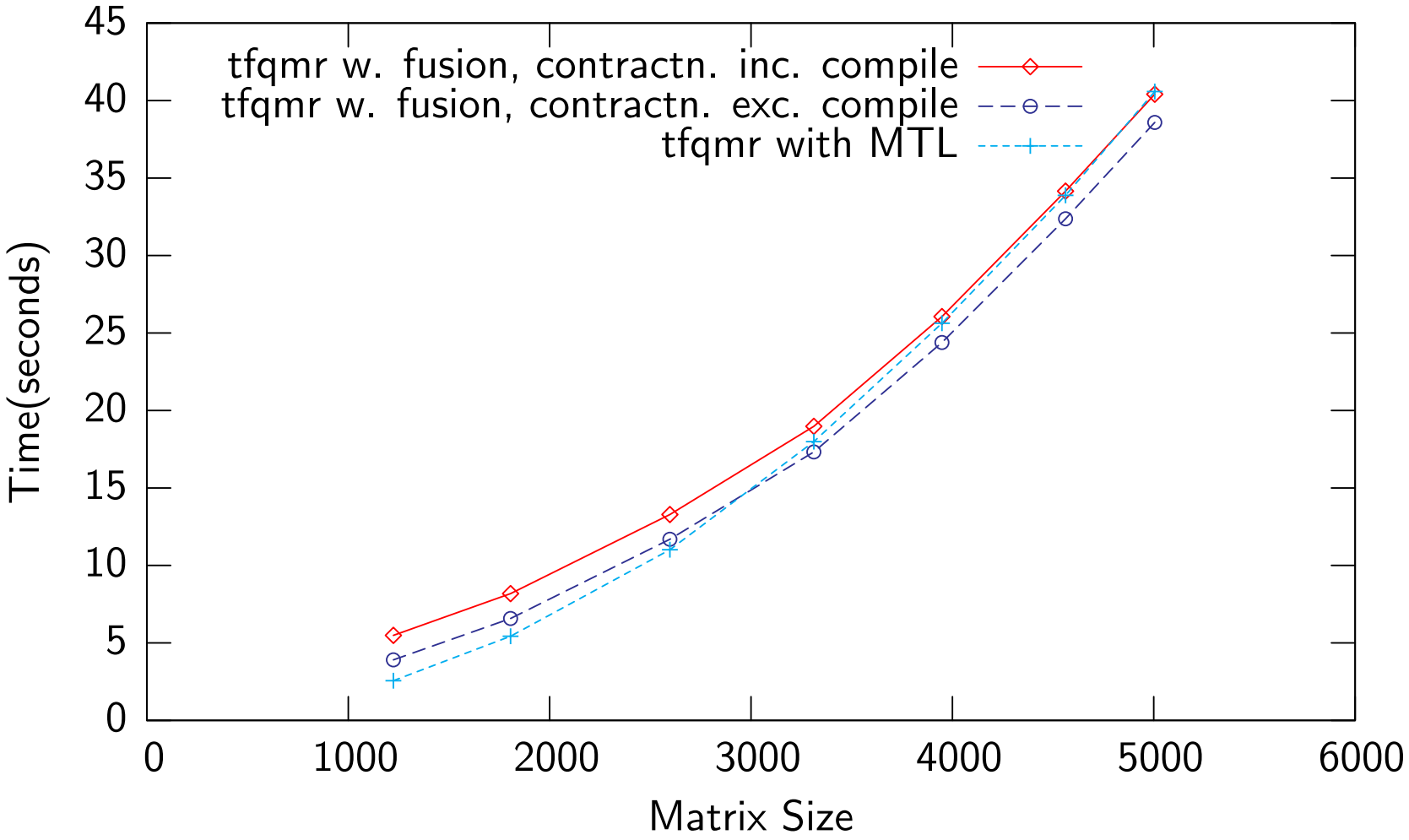
Rays Pentium IV processor running at 3.2GHz with Hyperthreading. 2048 KB L2 cache and 1 GB RAM.

Vertices Pentium IV processor running at 3.0GHz with Hyperthreading. 512 KB L2 cache and 1 GB RAM.

- The effects of loop fusion, array contraction and runtime liveness analysis were extremely similar on both architectures.
- Architectural differences had a significant impact when evaluating the performance of the prototype library against the state of the art Matrix Template Library.

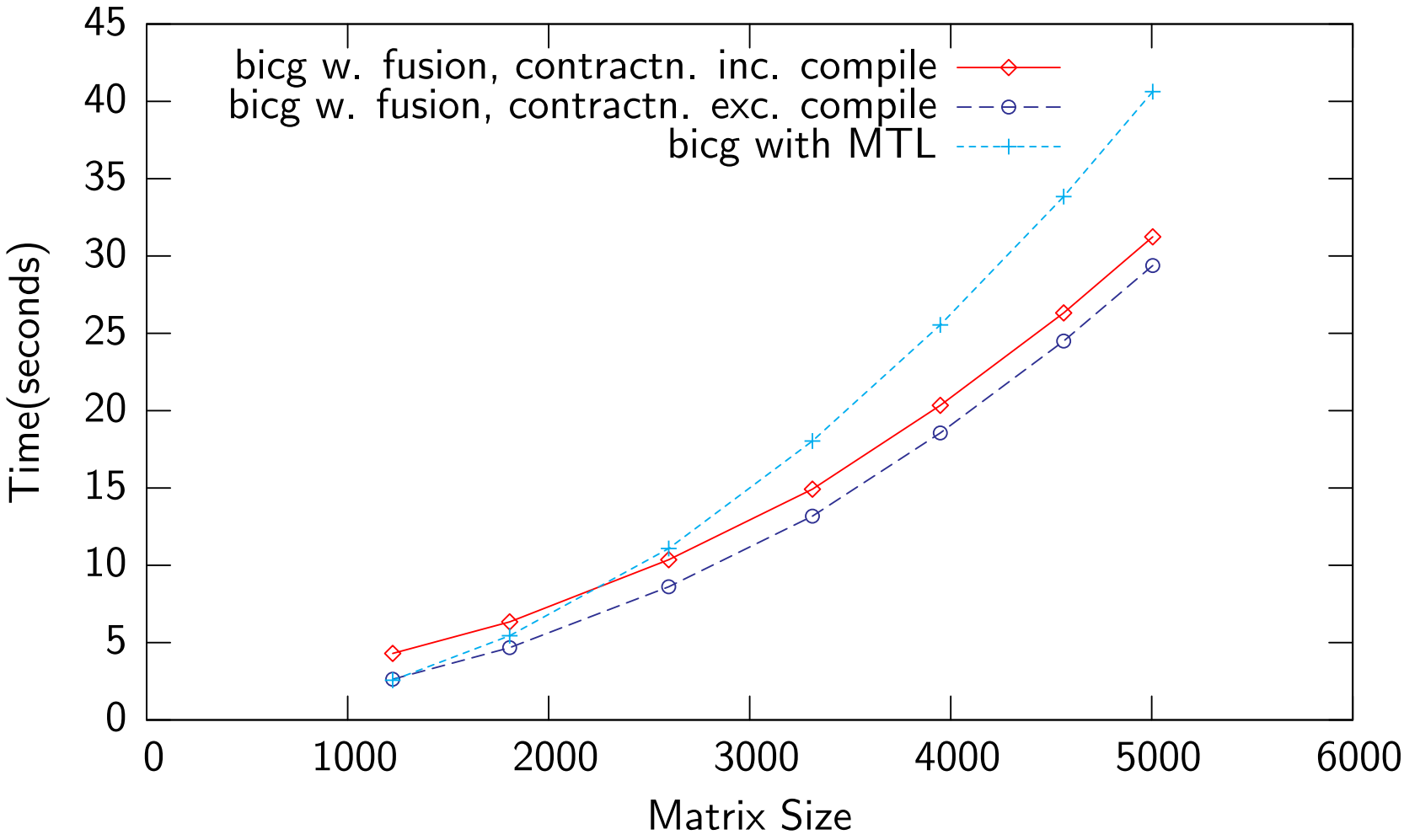
Comparison against MTL

256 iterations of Transpose Free Quasi-Minimal Residual on Vertices.



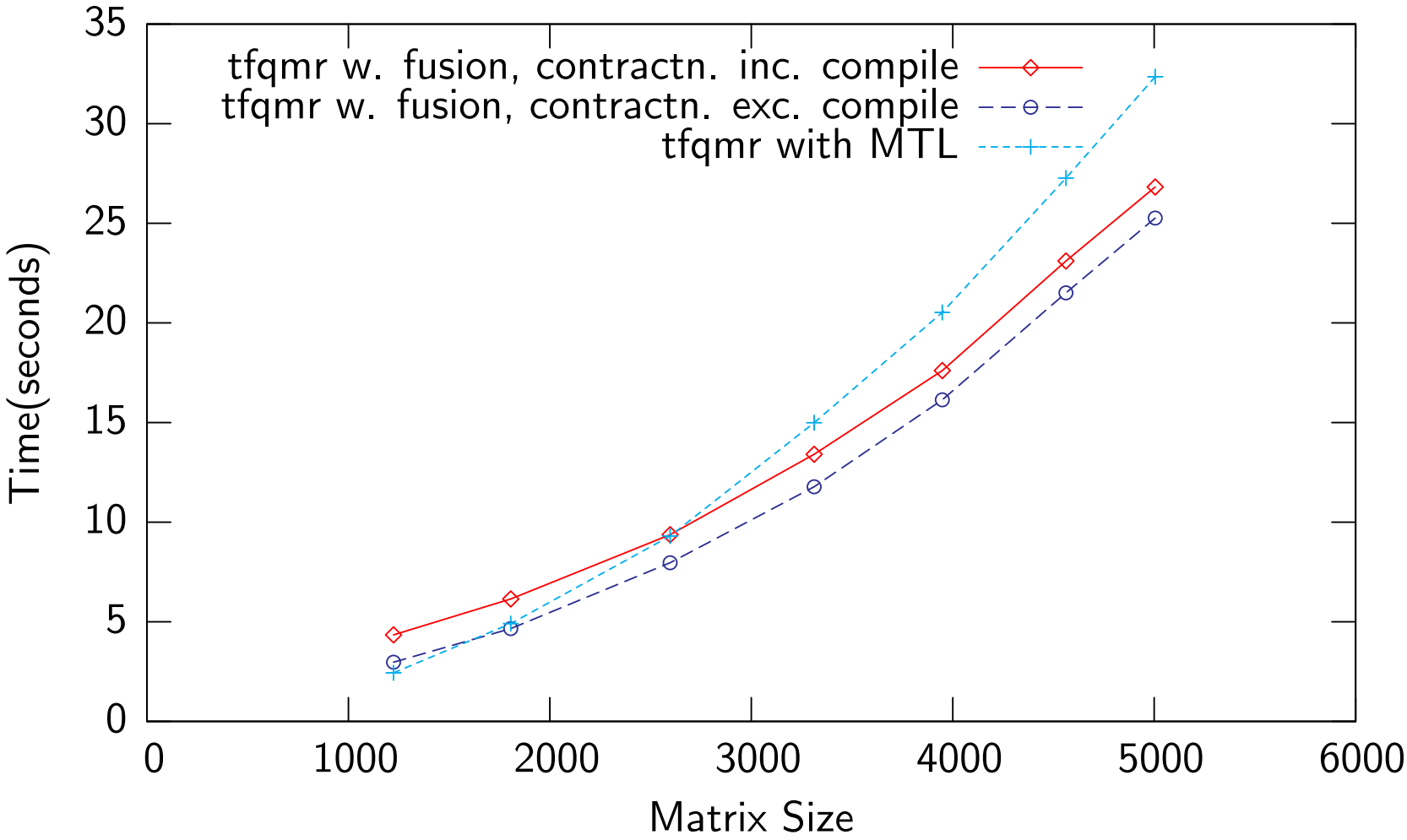
Comparison against MTL

256 iterations of BiConjugate Gradient Solver on Vertices.



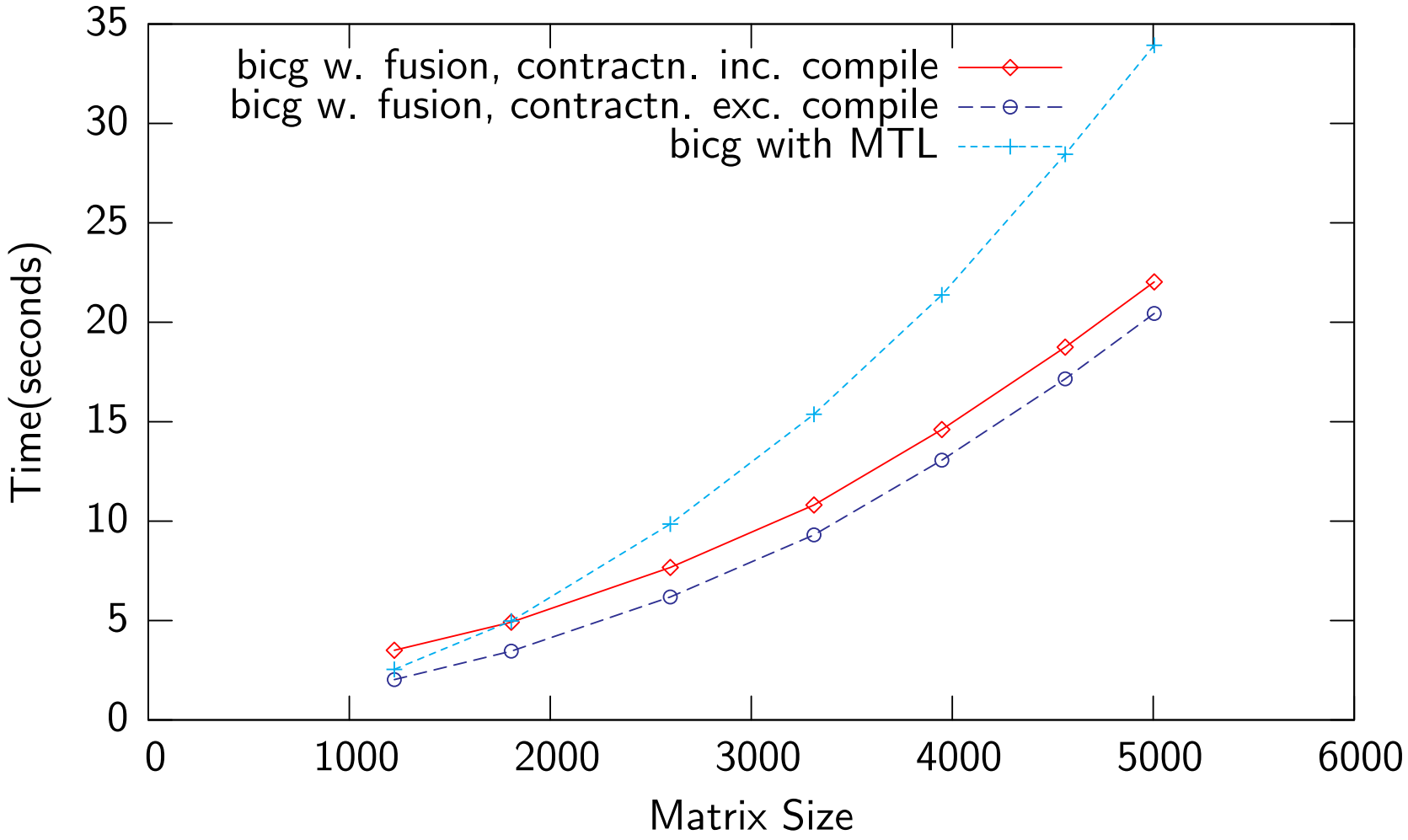
Comparison against MTL

256 iterations of Transpose Free Quasi-Minimal Residual on Rays.



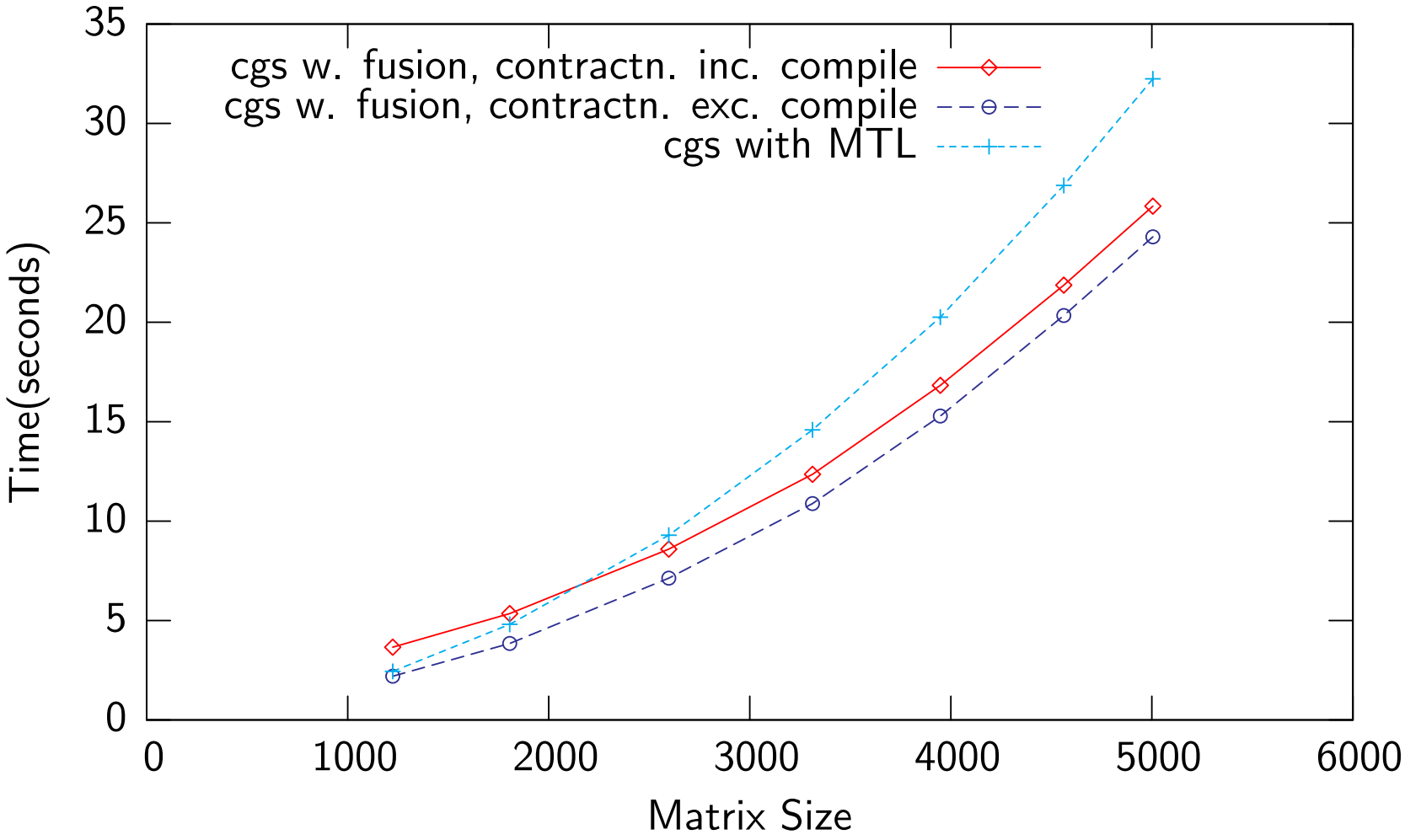
Comparison against MTL

256 iterations of BiConjugate Gradient Solver on Rays.



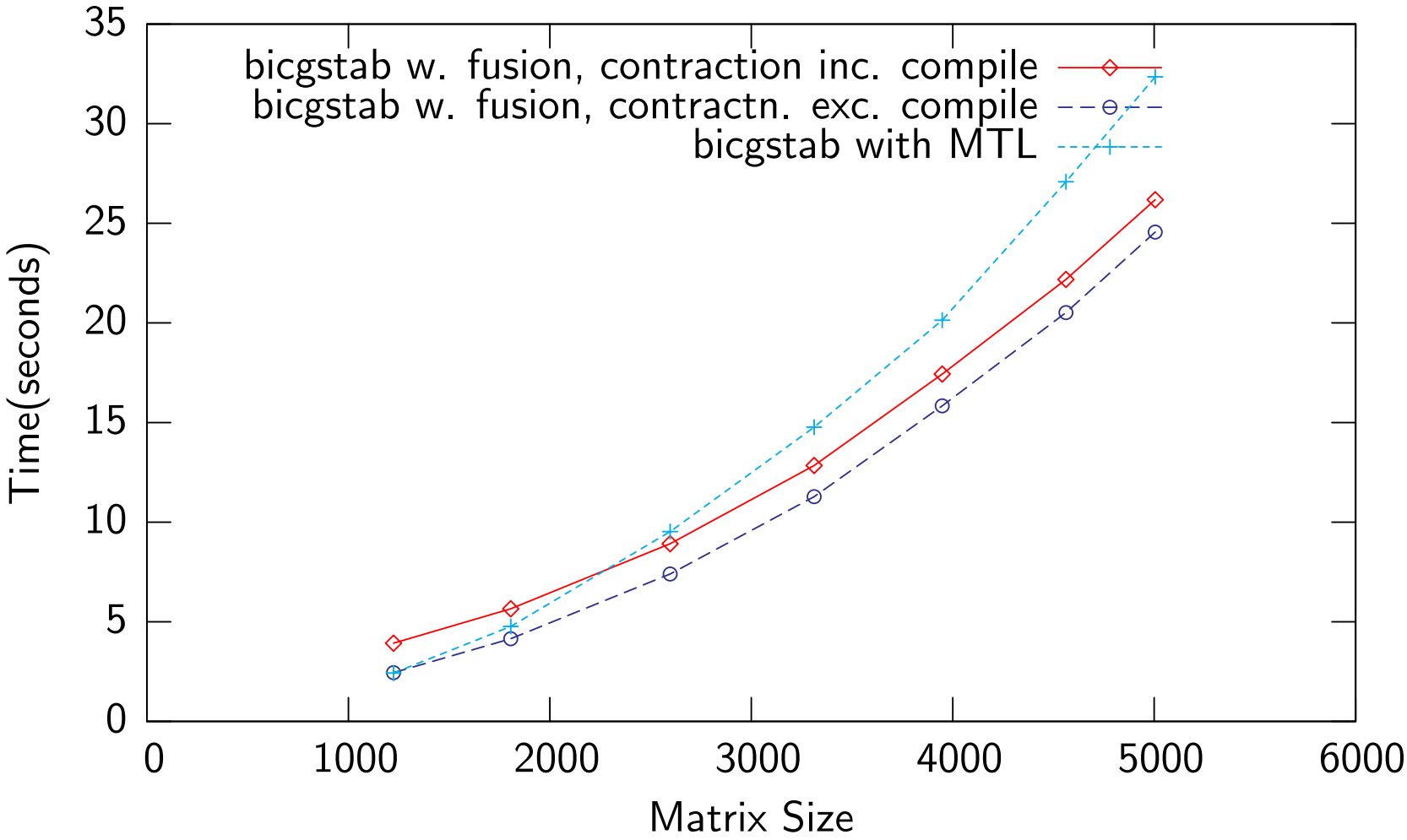
Comparison against MTL

256 iterations of Conjugate Gradient Solver on Rays.



Comparison against MTL

256 iterations of BiConjugate Gradient Stabilised Solver on Rays.



Conclusions

- On Vertices (excluding compilation) we get an average 2% speedup across all solvers and matrix sizes. Best speedup (excluding compilation) is on BiConjugate Gradient solver, with 38% speedup on 5005x5005 matrix.
- On Rays (excluding compilation) we get an average 27% speedup across all solvers and matrix sizes. Best speedup (excluding compilation) is on BiConjugate Gradient solver, with 64% speedup on a 5005x5005 matrix.
- Delayed evaluation and runtime code generation provides results.
- Importance of cross component optimisation.
- Limitations of conventional libraries.
- Importance of trend towards active libraries.

Questions Raised and Future Work?

Questions Raised

- When do we specialise and when do we aim to reuse?
- What should be evaluated and when?
- How well will code caching work on other applications?

Future Work

- Improved loop fusion heuristics.
- Alternative methods of expression DAG evaluation (like BLAS).
- Parallelisation.
- Sparse matrices.
- Storage format independent code generation.
- Persistent code caching.
- Ability for library user to specify algorithms for linear algebra operations.
- Speculative evaluation.