_TRSV: Optimizing triangular solve in CUDA

Jonathan Hogg

STFC Rutherford Appleton Laboratory

ASEArch flagship grant

Aims:

- Deliver a sparse linear solver on GPUs
- Deliver an interior point solver for linear/quadratic programs on GPUs
- Do so in such a way that they can be easily ported to other architectures

ASEArch flagship grant

Aims:

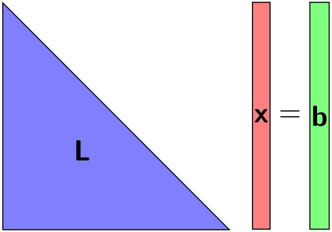
- Deliver a sparse linear solver on GPUs
- Deliver an interior point solver for linear/quadratic programs on GPUs
- Do so in such a way that they can be easily ported to other architectures

Relation of this talk:

- Learning project
- Base kernel we need to perform well — current CUBLAS implementation is poor.

What is _trsv?

- A Level 2 BLAS operation, solves $Lx = b$.
  _trsv — triangular solve.
- ...or $L^T x = b$ or $Ux = b$ or $U^T x = b$.



Usage

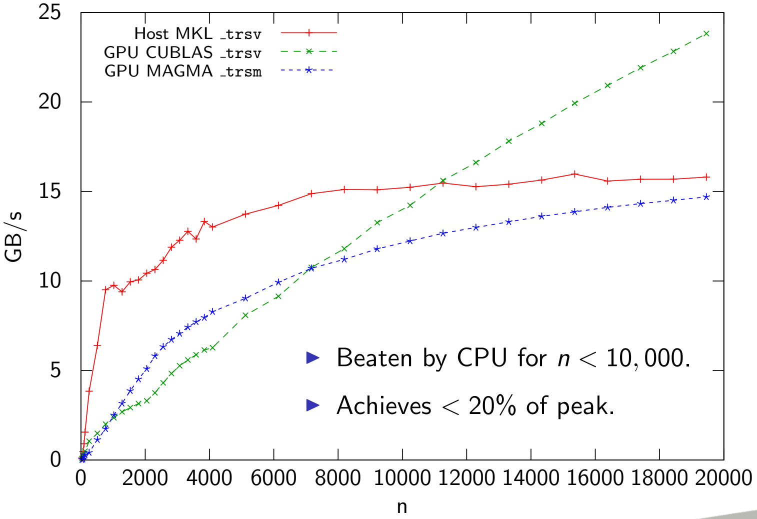
Direct solvers $A = LU$, or $A = LDL^T, A = QR$.

- Solve $Ax = b$ as $Ly = b$, $Ux = y$.
- Sparse solvers use many smaller matrices rather than one large dense one.

Often require 10s or 100s of solves per factorization

- Preconditioning, iterative refinement, FGMRES.
- Interior Point Methods perform multiple solves.

Current libraries



- Beaten by CPU for $n < 10,000$.
- Achieves $< 20\%$ of peak.

Basic (in-place) Algorithm

Input: Lower-triangular $n \times n$ matrix L, right-hand-side vector x.
for $i = 1, n$ **do**

$$\boxed{x(i+1:n)} = \boxed{x(i+1:n)} - \boxed{L(i+1:n,i)} * \boxed{x(i)}$$

end for
Output: solution vector x.

$$\begin{pmatrix} 1 & & & \\ l_{21} & 1 & & \\ l_{31} & l_{32} & 1 & \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \begin{pmatrix} \boxed{x_1} \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

Small matrices are latency bound

1 fmad per entry in $L \Rightarrow$ memory-bound.

- C2050 can deliver approx 9 doubles/sec from main memory
- Global memory latency 200 cycles (optimistic?)
- $n = 32 \Rightarrow$ 195 cycles per column waiting for data
- Require $n > 1800$ to fully hide latency
- Cache doesn't help — no hardware prefetch.

What can we do?

Small matrices are latency bound

1 fmad per entry in $L \Rightarrow$ memory-bound.

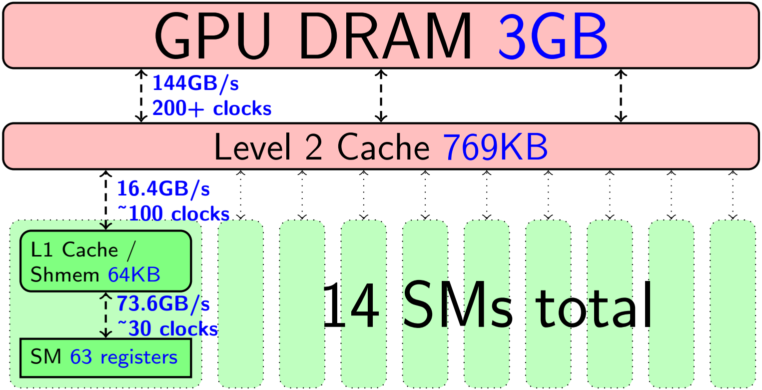
- C2050 can deliver approx 9 doubles/sec from main memory
- Global memory latency 200 cycles (optimistic?)
- $n = 32 \Rightarrow$ 195 cycles per column waiting for data
- Require $n > 1800$ to fully hide latency
- Cache doesn't help — no hardware prefetch.

What can we do?

Bring data closer to core, reducing latency

- Shared memory; or
- Registers

C2050 Memory layout



Registers

- Block on use, not on load.
- Allow Instruction Level Parallelism (ILP).
- See Volkov's *Better Performance at Lower Occupancy*.

Each thread only has 63 registers!
... typically need half of these for normal operation.

Registers

- Block on use, not on load.
- Allow Instruction Level Parallelism (ILP).
- See Volkov's *Better Performance at Lower Occupancy*.

Each thread only has 63 registers!
... typically need half of these for normal operation.

However, doesn't help:

- To use more than 1 thread, need to communicate via shared memory
  (so no latency gain).
- Adds complications to code ⇒ extra overheads.
- Quite quickly leads to register spill ⇒ slowdown.

Shared Memory

A 32×32 matrix of doubles requires 8KiB $\Rightarrow$ lots of room.

Simple code (`blkSize` = 32):

```
template <int blkSize>
void __device__ dblkSolve(const double *a, int lda,
      double &val) {

   volatile double __shared__ xs;

#pragma unroll 16
   for(int i=0; i<blkSize; i++) {
      if(threadIdx.x==i) xs = val;
      if(threadIdx.x>=i+1)
         val -= a[i*lda+threadIdx.x] * xs;
   }
}
```

Just precache a in shared memory!

Shared memory $n > 32$

Quickly run out of shared memory if we try and hold entire matrix!

Instead:

- Cache only 32×32 tiles down diagonal
- Cache next col while solve performed on diagonal

$$\begin{pmatrix} L_{11} & & & \\ L_{21} & L_{22} & & \\ L_{31} & L_{32} & L_{33} & \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix}$$

Shared memory $n > 32$

Quickly run out of shared memory if we try and hold entire matrix!

Instead:

- Cache only 32×32 tiles down diagonal
- Cache next col while solve performed on diagonal

$$\begin{pmatrix} L_{11} & & & \\ L_{21} & L_{22} & & \\ L_{31} & L_{32} & L_{33} & \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix}$$

Execution trace (128×128):

Warp 0	Ld(1)	Slv(1,1)	Mv(2,1)	Slv(2,2)	Mv(3,2)	Slv(3,3)	Mv(4,3)	Slv(4,4)
Warp 1	Ld(1)	Ld(2)	Mv(3,1)	Ld(3)	Mv(4,2)	Ld(4)		
Warp 2	Ld(1)	Ld(2)	Mv(4,1)	Ld(3)		Ld(4)		
Warp 3	Ld(1)	Ld(2)		Ld(3)		Ld(4)		

Small matrix results

$n =$	32	64	96	128
Shared-memory	7	13	19	25
Registers	17	37	68	149*
CUBLAS `dtrsv()`	31	58	85	113

* indicates register spill occurred

Larger matrices

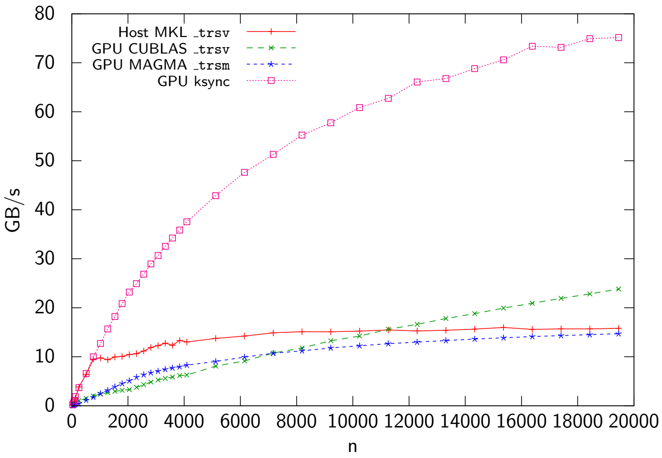
So far using a single SM.

- Quickly L1⟷L2 bandwidth becomes bounding (only 16.4GB/s vs 144GB/s global)
- Need to use multiple SMs!

Why not use small matrix kernel then efficient matrix-vector?

- Driver handles synchronization (different kernels)
- Matrix-vector achieves high bandwidth

Kernel-synchronized results



We can do better!

$n =$	512	1024	4096
`blkSolve()` (μs)	108.3	217.3	904.7
`dgemv()` (μs)	37.8	95.1	842.0
Execution time (μs)	171.0	370.8	2006.5
Launch overhead	**17.0%**	**18.7%**	**14.9%**
Work in `blkSolve()`	**18%**	**9%**	**2%**

- Substantial overheads from using kernel launches for synchronization
- Amount of time in blkSolve() — Amdahl strikes again!

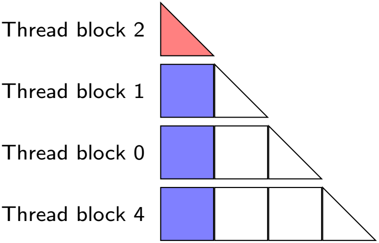
Global-memory synchronized

Aim: Single kernel-launch

- Use global memory for synchronization — costs l2 cache miss + `__threadfence()`.
  (Much cheaper than using kernel launches)
- Fine grained synchronization...
- ...hence matrix-vector product runs concurrently with solve.

Science & Technology Facilities Council

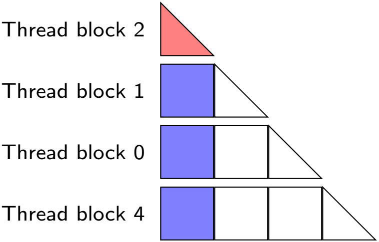
Thread block $\Rightarrow$ block row



CAUTION

Thread blocks are not scheduled in order!

Thread block ⇒ block row

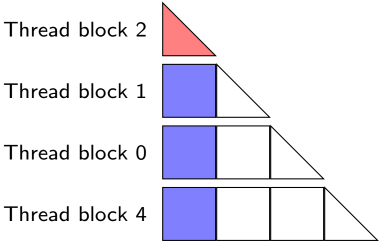


CAUTION

Thread blocks are not scheduled in order!

Dynamically pick row to avoid deadlock

Thread block ⇒ block row



CAUTION

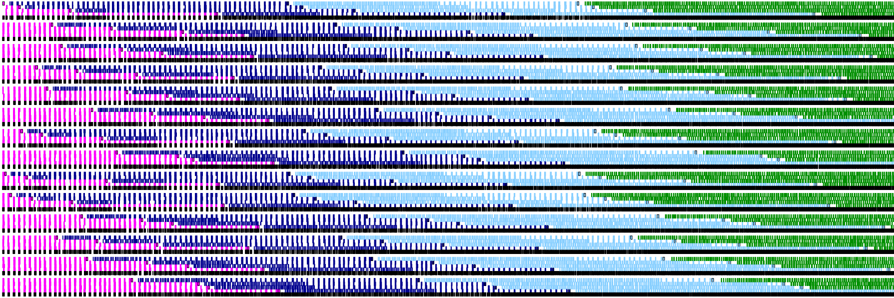
Thread blocks are not scheduled in order!

Dynamically pick row to avoid deadlock

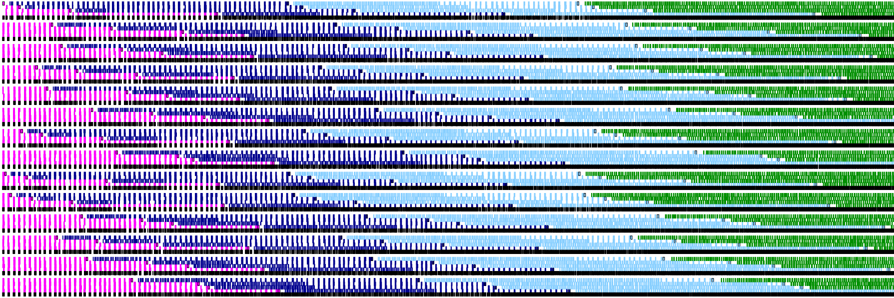
Only need two scalars for synchronization:

- Row for next thread block
- Latest column for which solution is available

Execution trace



Execution trace



Mode 1 Not waiting on data, constant computation.

Mode 2 Stops and starts as each column completes.

Performance bounds

4 blocks per SM with different behaviours:

Mode 1 **Not waiting on data, constant computation.**

Mode 2 Stops and starts as each column completes.

- Mode 1 is bandwidth bound.
- Only takes one thread block per SM to saturate.
- Competitive with CUBLAS _gemv.
- Little room for improvement.

Performance bounds

4 blocks per SM with different behaviours:

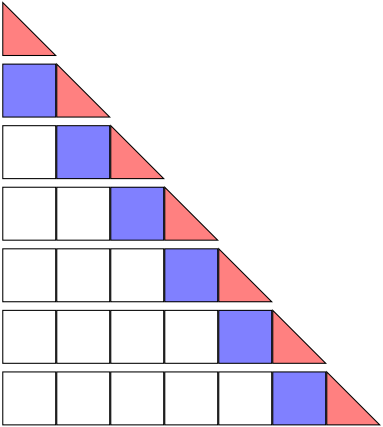
Mode 1 Not waiting on data, constant computation.

Mode 2 Stops and starts as each column completes.

- Mode 2 has short bursts of activity, but is mostly idle.
- Has to wait for data on the critical path.
- Significant at start of computation as affects all blocks.
- 14 SMs $\times$ 4 blocks each $\times$ 32 rows/block $\Rightarrow$
  $n = 1792$ before any Mode 1 occurs.

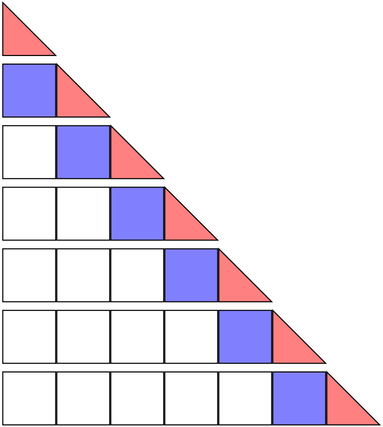
Science & Technology Facilities Council

Critical path



Critical path is coloured;
Executes serially

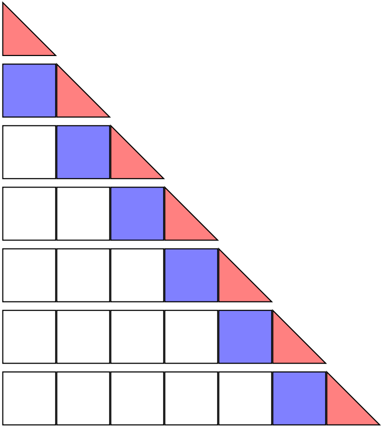
Critical path



Critical path is coloured;
Executes serially

Use tricks from before:
pre-cache values

Critical path



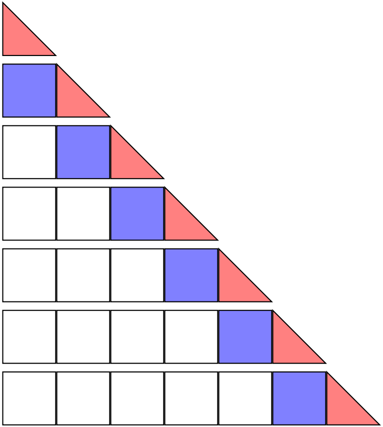
Critical path is coloured;
Executes serially

Use tricks from before:
pre-cache values

BUT:
Maintain high occupancy!

Science & Technology Facilities Council

Critical path



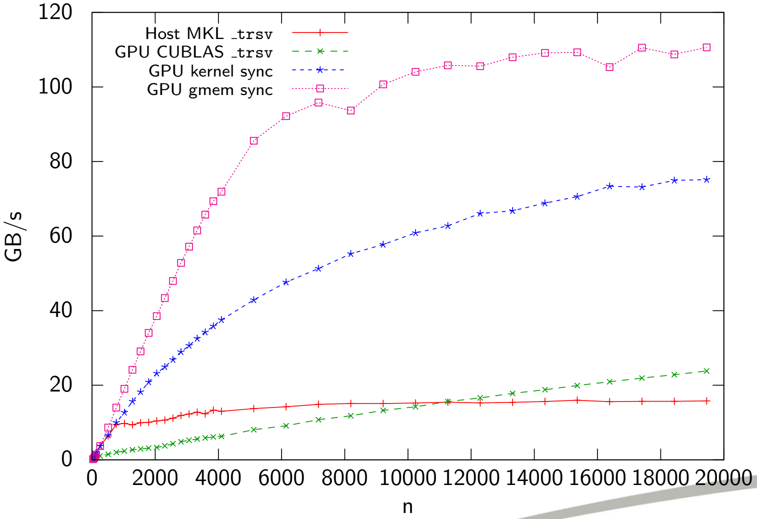
48k shmem $\Rightarrow$ At most 5 32×32 tiles

Want 4 thread blocks/SM!

- Use shared memory for diagonal tiles.
- Use registers for subdiagonal tiles.

Science & Technology Facilities Council

Global-memory synchronization results



Science & Technology Facilities Council

Better yet!

Memory-bound $\Rightarrow$ spare flops

Can we do redundant computation to speed the critical path?

Better yet!

Memory-bound $\Rightarrow$ spare flops

Can we do redundant computation to speed the critical path?

YES

Explicit inversion of diagonal blocks

- Diagonal solve $\rightarrow$ Matrix-vector multiply
- Same number of memory accesses, *less communication*!

Explicit inversion

$$\begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} X_{11} & \\ X_{21} & X_{22} \end{pmatrix} = \begin{pmatrix} L_{11}X_{11} & \\ L_{21}X_{11} + L_{22}X_{21} & L_{22}X_{22} \end{pmatrix}$$

Equate to identity.

$$\begin{array}{rcll} X_{11} & = & L_{11}^{-1} & \text{by recursion} \\ X_{22} & = & L_{22}^{-1} & \text{by recursion} \\ L_{22}X_{21} & = & -L_{21}X_{11} & \text{solve is stable - Higham 1995} \end{array}$$

Explicit inversion

$$\begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} X_{11} & \\ X_{21} & X_{22} \end{pmatrix} = \begin{pmatrix} L_{11}X_{11} & \\ L_{21}X_{11} + L_{22}X_{21} & L_{22}X_{22} \end{pmatrix}$$

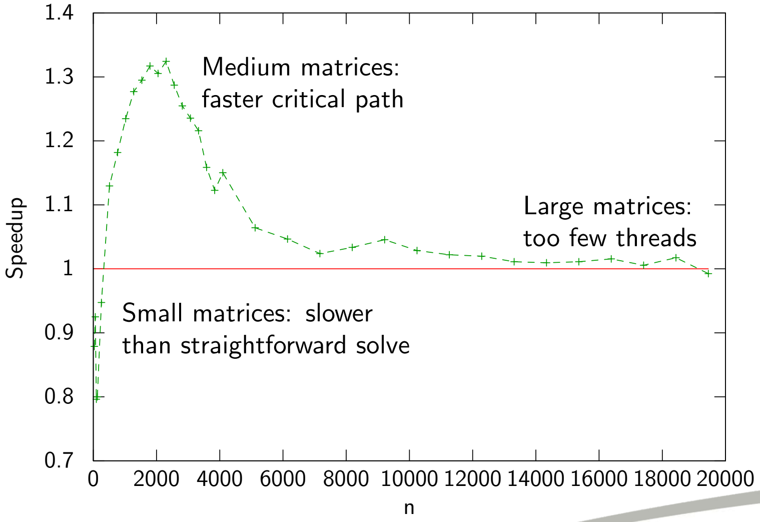
Equate to identity.

$$\begin{array}{rcll} X_{11} & = & L_{11}^{-1} & \text{by recursion} \\ X_{22} & = & L_{22}^{-1} & \text{by recursion} \\ L_{22}X_{21} & = & -L_{21}X_{11} & \text{solve is stable - Higham 1995} \end{array}$$

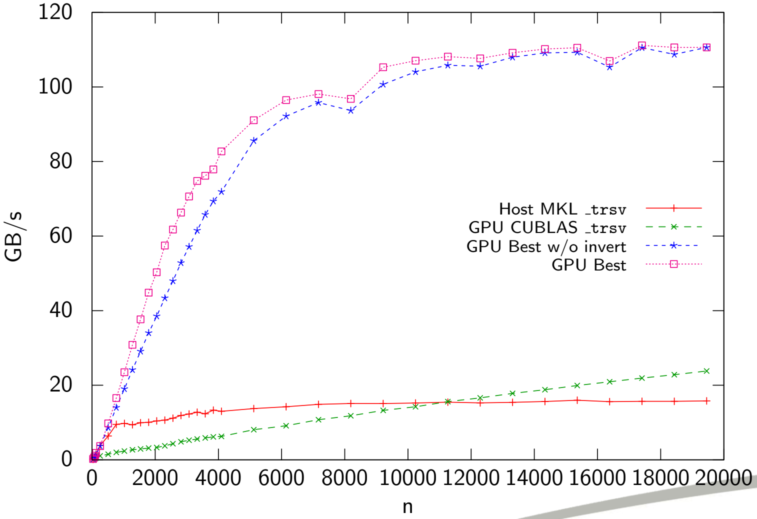
Doesn't require right-hand-side — can be done before needed

BUT: takes considerably longer than a solve: useless for small n.

Speedup over previous version

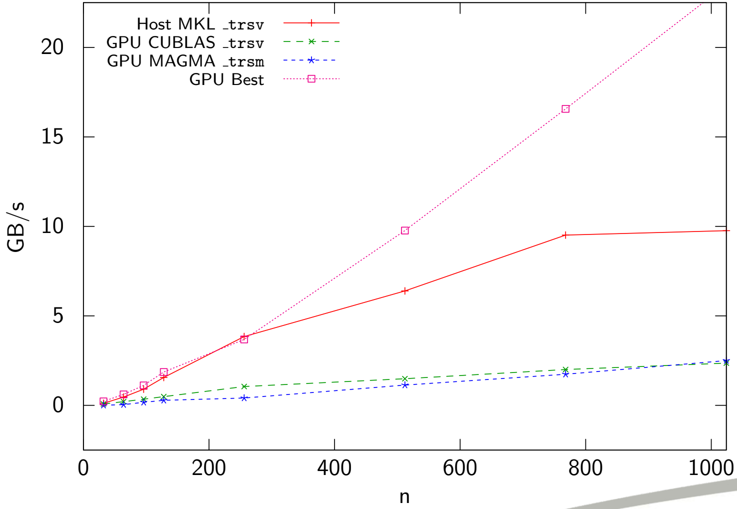


Overall best performance

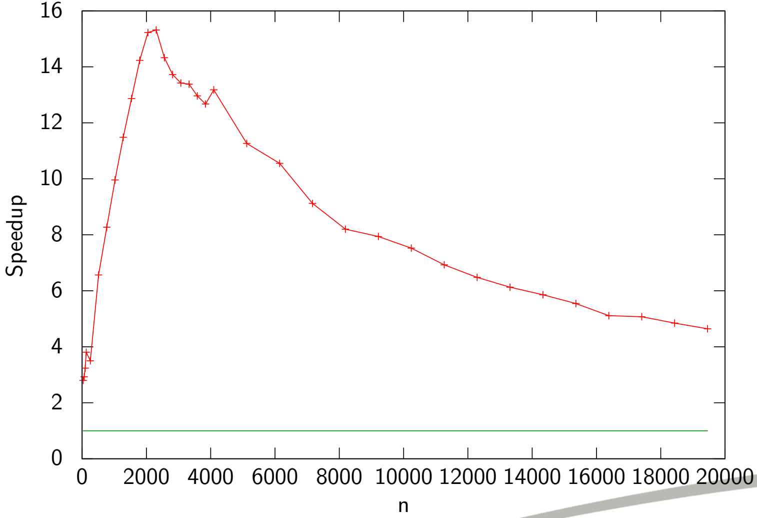


Science & Technology Facilities Council

Overall best performance (zoomed)



Speedup vs CUBLAS



Science & Technology Facilities Council

Conclusions

We've beaten CUBLAS soundly.
Achieved 75% of peak bandwidth.

- Can we do even better somehow?
- Could use tasks — but register pressure!

Next step is the sparse case

Code will be available under BSD licence

Science & Technology Facilities Council

Questions?