A fast triangular solve on GPUs

Jonathan Hogg

STFC Rutherford Appleton Laboratory

28th January 2012, Bath/RAL Numerical Analysis Day

GPUs and manycore programming

Nomenclature

Multicore Handful of big heavyweight cores. Most desktop machines.

Manycore Hundreds of lightweight cores. Many competing models.

Manycore architectures

- NVIDIA GPUs
- AMD GPUs
- Intel MIC (Xeon Phi/Knights Corner)

Lots of functional units that can be repeated ad infinitum.

What specs are we talking?

Chip	Cores	GB/ sec	TFLOP/ sec	GFLOPS/ Watt
NVIDIA K20X	13×64	250	1.31	5.6
AMD FirePro S10000	$2 \times 56 \times 32^*$	480	1.48	3.9
Intel Xeon Phi	60×8	320	1.00	4.5
Intel Desktop E5-2687W	16×4	50	0.20	1.3

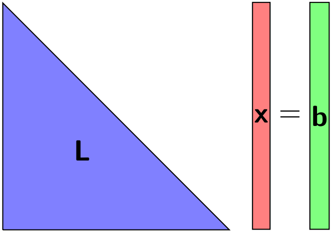
* single precision cores. double precision is 1/4.

$\Rightarrow$ **Definitely worth using.**

Note: GPU single precision performance much more than twice double.

Example: Triangular solve on a GPU

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



- Unusual GPU application: Memory bandwidth bound.
  Latency sensitive.

Usage

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

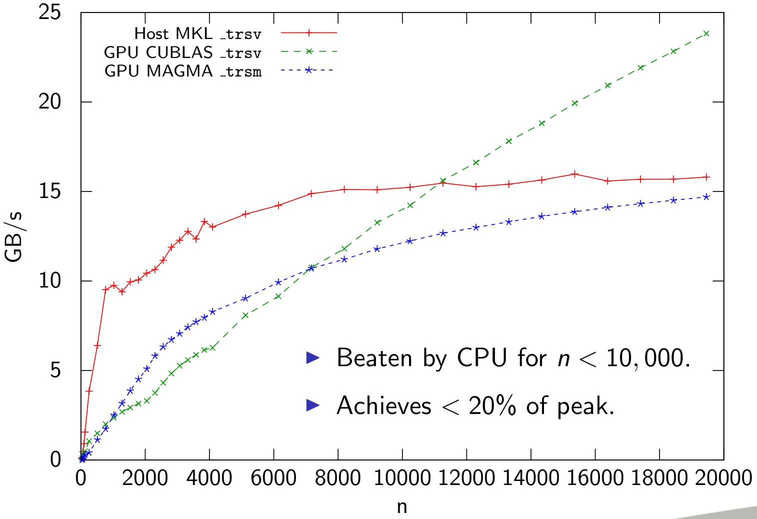
- Solve $Ax = b$ as $Ly = b$, $Ux = y$.
- Sparse solvers use many smaller matrices rather than one large dense one.

Often require 10s or 100s of solves per factorization

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

Science & Technology Facilities Council

Current libraries



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

Science & Technology Facilities Council

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.

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

Performance programming in one slide

All about chasing bottlenecks

Performance programming in one slide

All about chasing bottlenecks

Aim: Perform as many operations as we can.

Constraints:

- How many operations I can perform simultaneously (processor width $\times$ clock speed). *"Compute bound"*
- Whether the data is ready. *"Memory bound"*

Performance programming in one slide

All about chasing bottlenecks

Aim: Perform as many operations as we can.

Constraints:

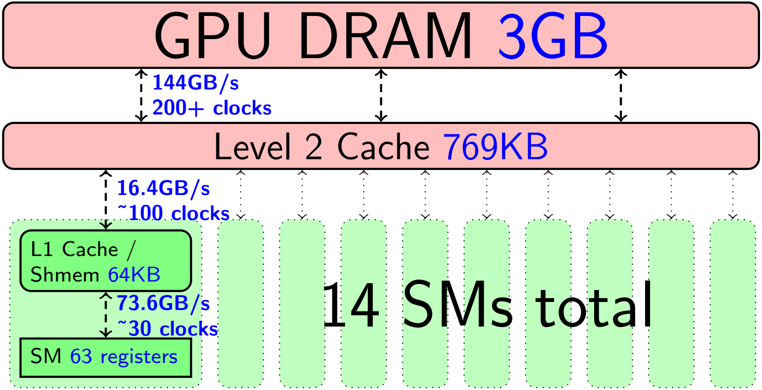
- How many operations I can perform simultaneously (processor width $\times$ clock speed). *"Compute bound"*
- Whether the data is ready. *"Memory bound"*

Data may not be ready because:

- Waiting for previous operation to finish (instruction latency)
- Data transfer rate from memory (memory bandwidth)
- Round-trip time following request (memory latency)

Complicated by multiple hierarchical levels of memory.

C2050 Memory layout (previous generation)



SM = Symmetric Multiprocessor

Theoretical bounds

- Number of entries is $\frac{1}{2}n(n+1)$
- Single SM: Main memory 2 doubles for every 32 ops.
- Entire GPU: Main Memory 16 doubles for every 448 ops.

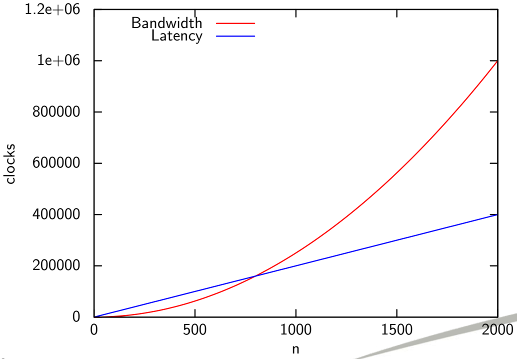
Theoretical bounds

- Number of entries is $\frac{1}{2}n(n+1)$
- Single SM: Main memory 2 doubles for every 32 ops.
- Entire GPU: Main Memory 16 doubles for every 448 ops.
- Incur latency n times. Can only treat one column at a time.
- Global memory latency: 200 cycles (optimistic?)

Science & Technology Facilities Council

Theoretical bounds

- Number of entries is $\frac{1}{2}n(n+1)$
- Single SM: Main memory 2 doubles for every 32 ops.
- Entire GPU: Main Memory 16 doubles for every 448 ops.
- Incur latency n times. Can only treat one column at a time.
- Global memory latency: 200 cycles (optimistic?)



Take highest curve.

Small matrices:

Latency bound

Large matrices:

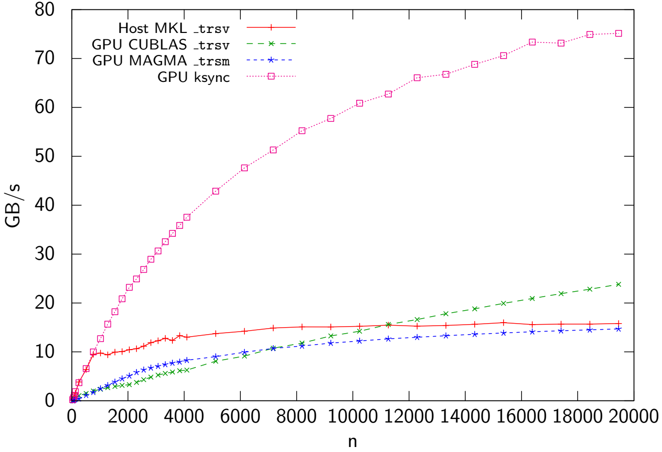
Bandwidth bound

2-kernel solution

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

- Apply our own tuned kernel to diagonal block.
- Apply CUBLAS `_gemv` kernel to off-diagonal blocks.
- Repeat for next block column.
- NVIDIA Driver enforces ordering for us.

Kernel-synchronized results



We can do better!

$n =$	512	1024	4096
`blkSolve()` (μs)	108	217	905
`dgemv()` (μs)	38	95	842
Execution time (μs)	171	371	2007
Launch overhead	**17%**	**19%**	**15%**
Work in `blkSolve()`	**18%**	**9%**	**2%**

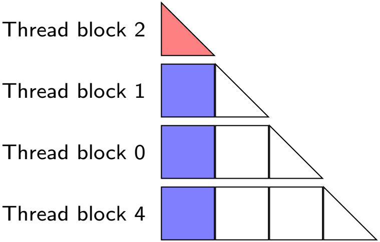
- Substantial overheads from using kernel launches for synchronization
- Amount of time in blkSolve() — Amdahl strikes again!

Global-memory synchronized

Aim: Single kernel-launch

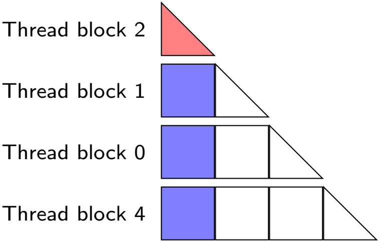
- Use global memory for synchronization — much cheaper than using the NVIDIA driver
- Fine grained synchronization...
- ...hence matrix-vector product runs concurrently with solve.

Thread block ⇒ block row



CAUTION
Thread blocks are not scheduled in order!

Thread block ⇒ block row



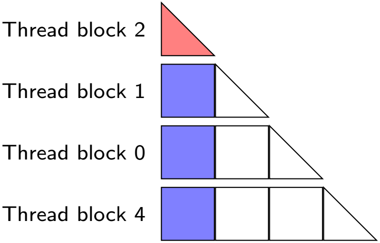
CAUTION

Thread blocks are not scheduled in order!

Dynamically pick row to avoid deadlock

Science & Technology Facilities Council

Thread block $\Rightarrow$ block row



CAUTION

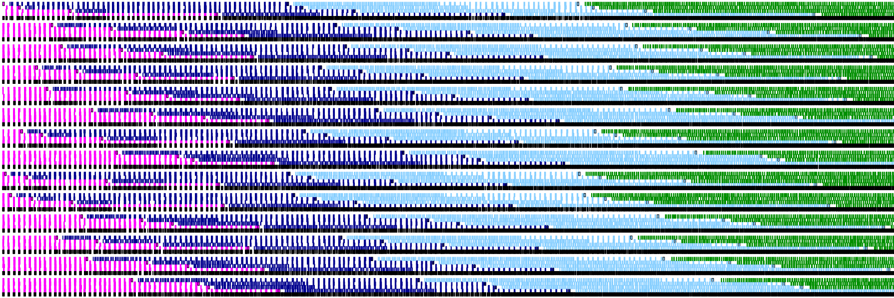
Thread blocks are not scheduled in order!

Dynamically pick row to avoid deadlock

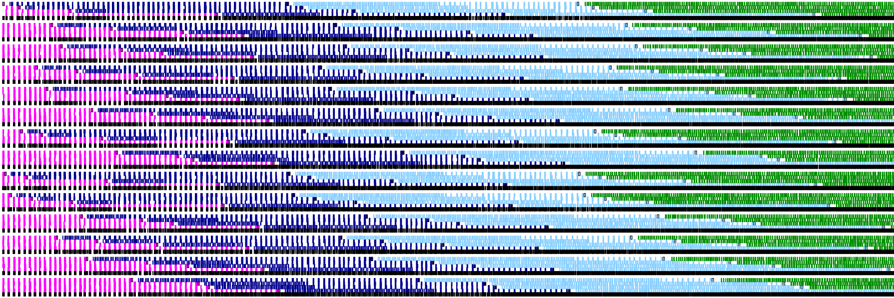
Only need two scalars for synchronization:

- 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.

Science & Technology Facilities Council

Performance model

Only really interested in when it finishes.

- Each SM has 4 'slots'.
- Look at slot that executes the final block row k.
- Same slot executes $k, k-56, k-2*56, \ldots$.
- Calculate execution time for each of these blocks and add them together.

First few blocks are latency bound

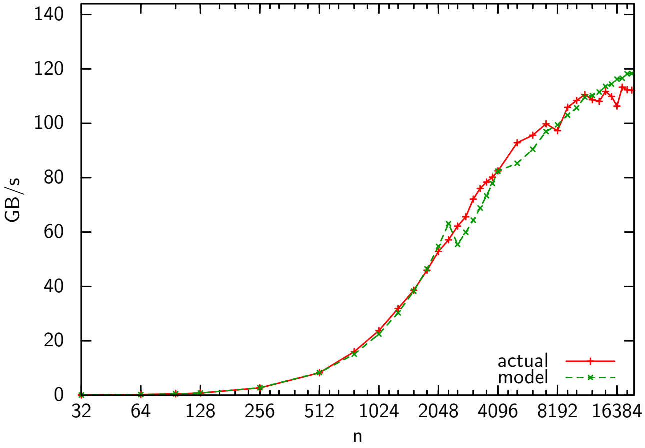
- Model as $t_{init} + nblk \times t_{latency}$.

Subsequent blocks are bandwidth bound

- Model as $t_{init} + nblk \times t_{bandwidth}$.

Science & Technology Facilities Council

Performance model (cont.)



Performance model (cont.)

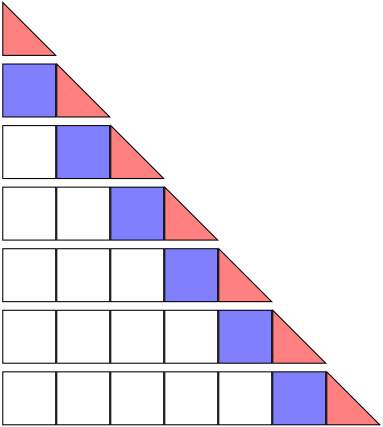
Performance model

$$t = t_{setup} + nrow \times t_{init} + nblk_{latency} \times t_{latency} + nblk_{bandwidth} \times t_{bandwidth}$$

- Can't improve $t_{bandwidth}$: physical limitation.
- Aim to reduce $t_{latency}$.

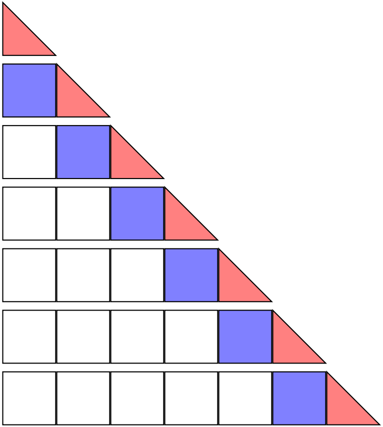
Science & Technology Facilities Council

Latency Critical path



Critical path is coloured;
Executes serially

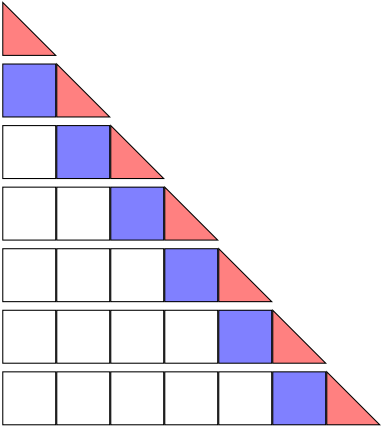
Latency Critical path



Critical path is coloured;
Executes serially

Use standard tricks:
pre-cache values

Latency Critical path

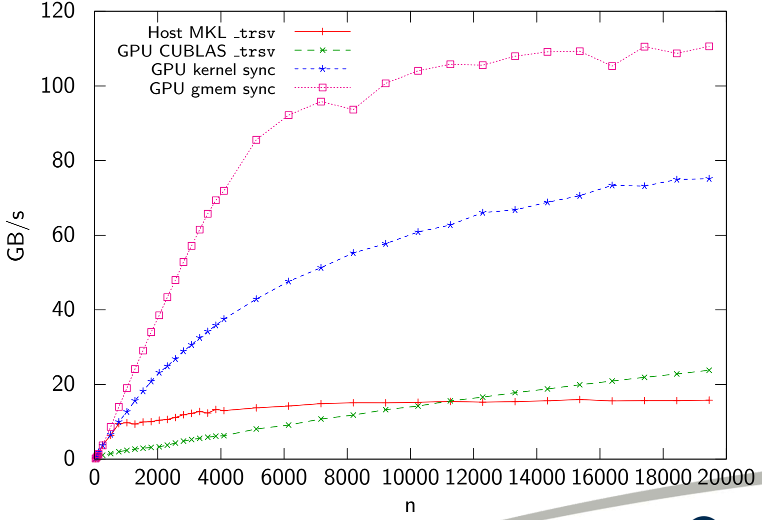


48k shmem $\Rightarrow$ At most 5 32×32 tiles

Want 4 thread blocks/SM!

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

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*!

Science & Technology Facilities Council

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

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

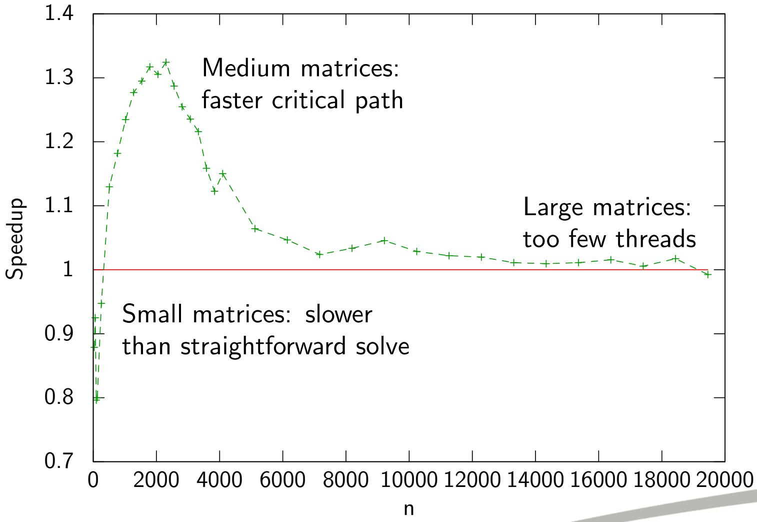
Equate to identity.

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

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

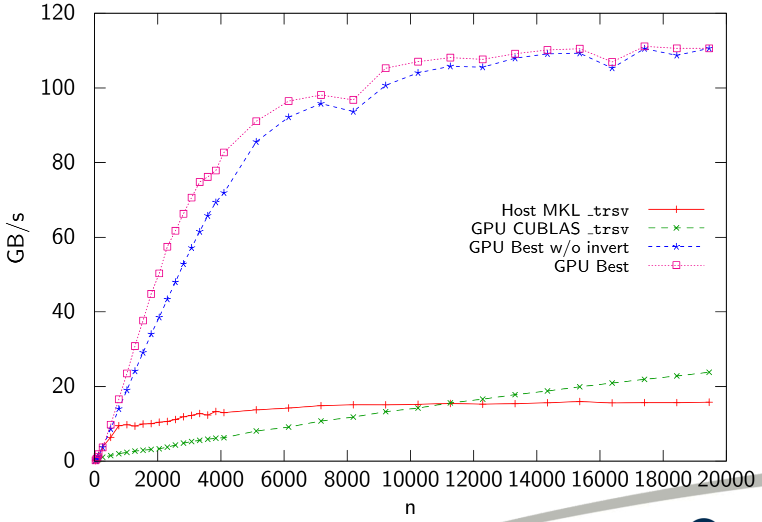
BUT: takes considerably longer than a solve: useless for small n.

Speedup over previous version

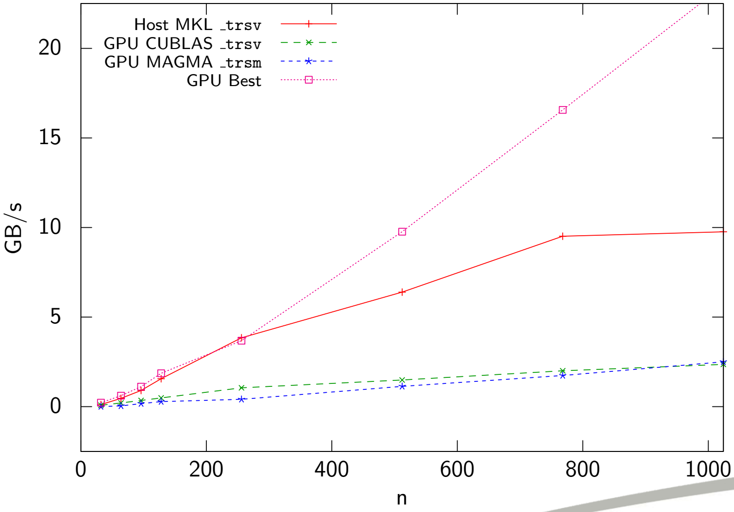


Science & Technology Facilities Council

Overall best performance



Overall best performance (zoomed)



Science & Technology Facilities Council

Conclusions and Lessons

We've beaten CUBLAS soundly.
Achieved 75% of peak bandwidth.
Code will be in next version of CUBLAS.

Lessons

- Its all about the memory.
- Spending extra ops to reduce memory latency or bandwidth can be worthwhile.
- CUDA is nice: we get explicit control over memory movements.
- (CUDA is horrible: we need to explicitly control memory movements).

Science & Technology Facilities Council