



Synthesis: Simulating sound on GPUs

S. Bilbao, K. Kavoussanakis, N. Mc Donnell,
C.M. Maynard, S. Petrou and C. Webb

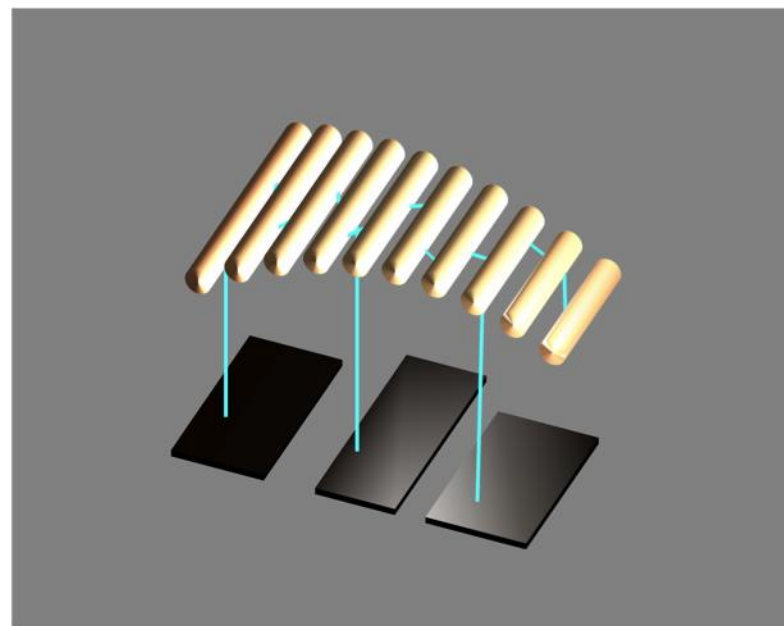
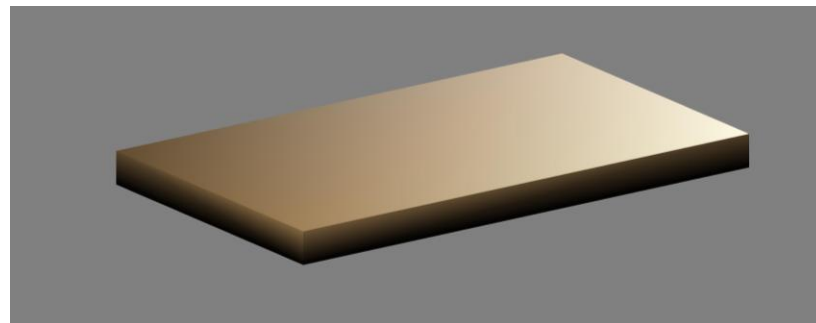
Department of Music

EPCC

University of Edinburgh

Dr Chris Maynard
Application Consultant, EPCC
c.maynard@ed.ac.uk
+44 131 650 5077

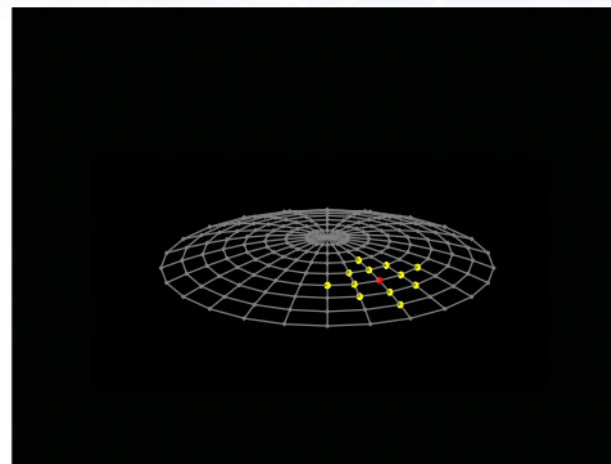
- Not a real gong! 🗣️
- Synthesised using a MATLAB code
- physical model of metal plate
- Time-domain method
- Ported example codes to GPUs
 - three month project!



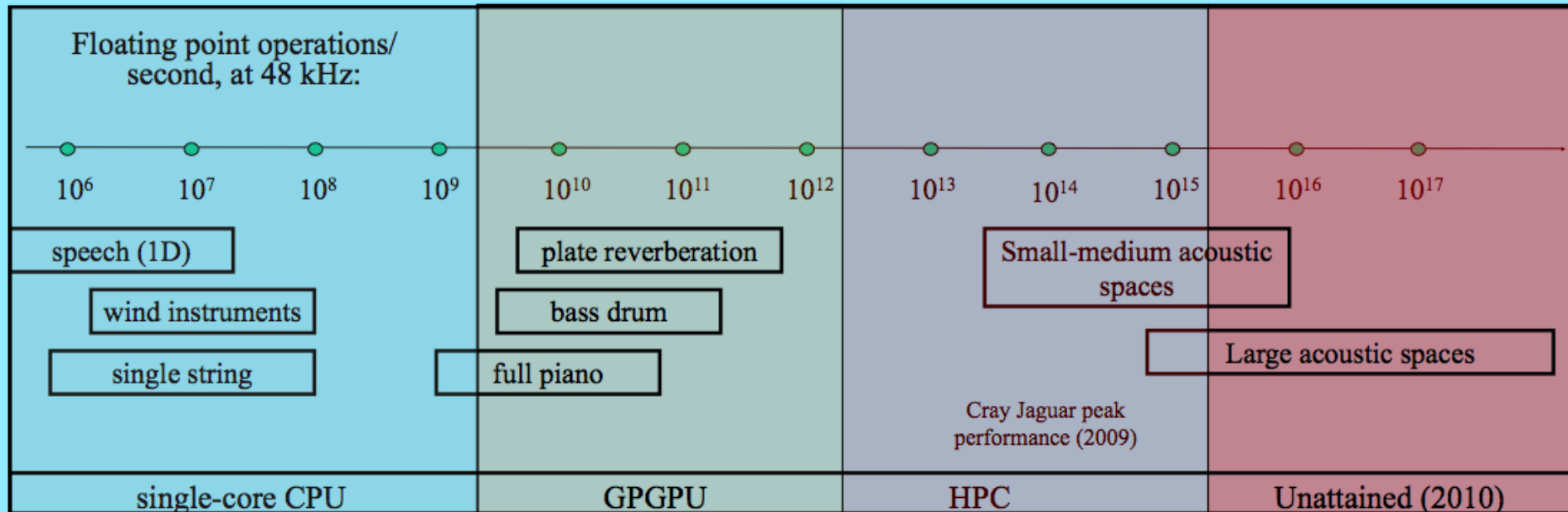
- An old story ...
 - isn't the one about old dogs and new tricks!
 - Plenty of new tricks for GPUs!
- To exploit highly parallel architecture
- necessary to exploit the data parallelism
 - requires application domain knowledge



- Time domain simulations
 - suitable for all musical acoustic
 - especially non-linear problems
- Choose mesh
 - FD, FEM, spectral etc



- Develop time domain recursion
 - audio sample rate, e.g., 44.1 kHz
- Time updates can be written
- Sparse Matrix – Vector multiplication (SpMV)



- Generally scales with nD volume, dimension, sample density
- General lower bounds on complexity follow from basic physics

- NVIDIA Tesla C1060
- 240 Streaming Processor cores
- Clock speed 1.3GHz
- Peak performance
 - Single Precision: 933 Gflops
 - Double Precision: 78 Gflops
 - >10 SP : DP
- Memory speed 800MHz
 - interface 512-bit
 - 102 GB/sec



- CUDA 2.3

- flops-per-word of memory access FpW
- Size of Word W
- Memory bandwidth Mb
- Peak flop rate per sec PF

$$FpW = \frac{PF}{Mb/W}$$

$$FpW_S = \frac{933GFs^{-1}}{102GBs^{-1}/4B} = 37$$

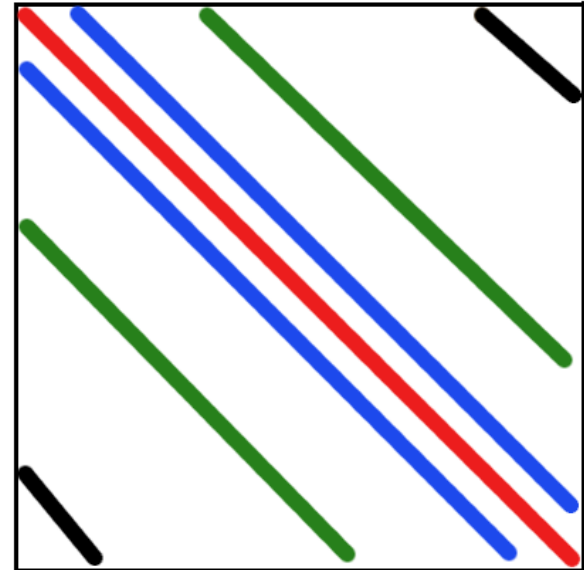
$$FpW_D = \frac{78GFs^{-1}}{102GBs^{-1}/8B} = 6$$

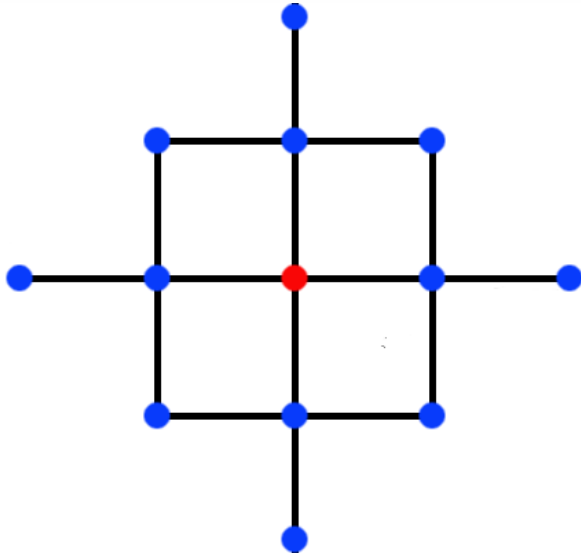
What does this mean?

- Typical scientific application
 - $\frac{1}{2}$ - 2 flops per word
- On Tesla
 - 37 flops per word SP
 - 6 flops per word DP
 - *c.f.* HECToR ~ 1
- **Flops are free**
 - Many more flops are available than can be used
- Data layout and memory management are critical
- NUMA



- The computational kernel
- Apply sparse matrix to vector each update
- Sparse matrix is a *Toeplitz* matrix
 - response to a system of a sudden impulse
 - e.g. Hitting a metal plate with a hammer
- Significant banded structure
- Only a few independent numbers





- 2-dimensional problem
- Finite difference 13-point stencil
- update red-point: depends on blue points of previous time-step

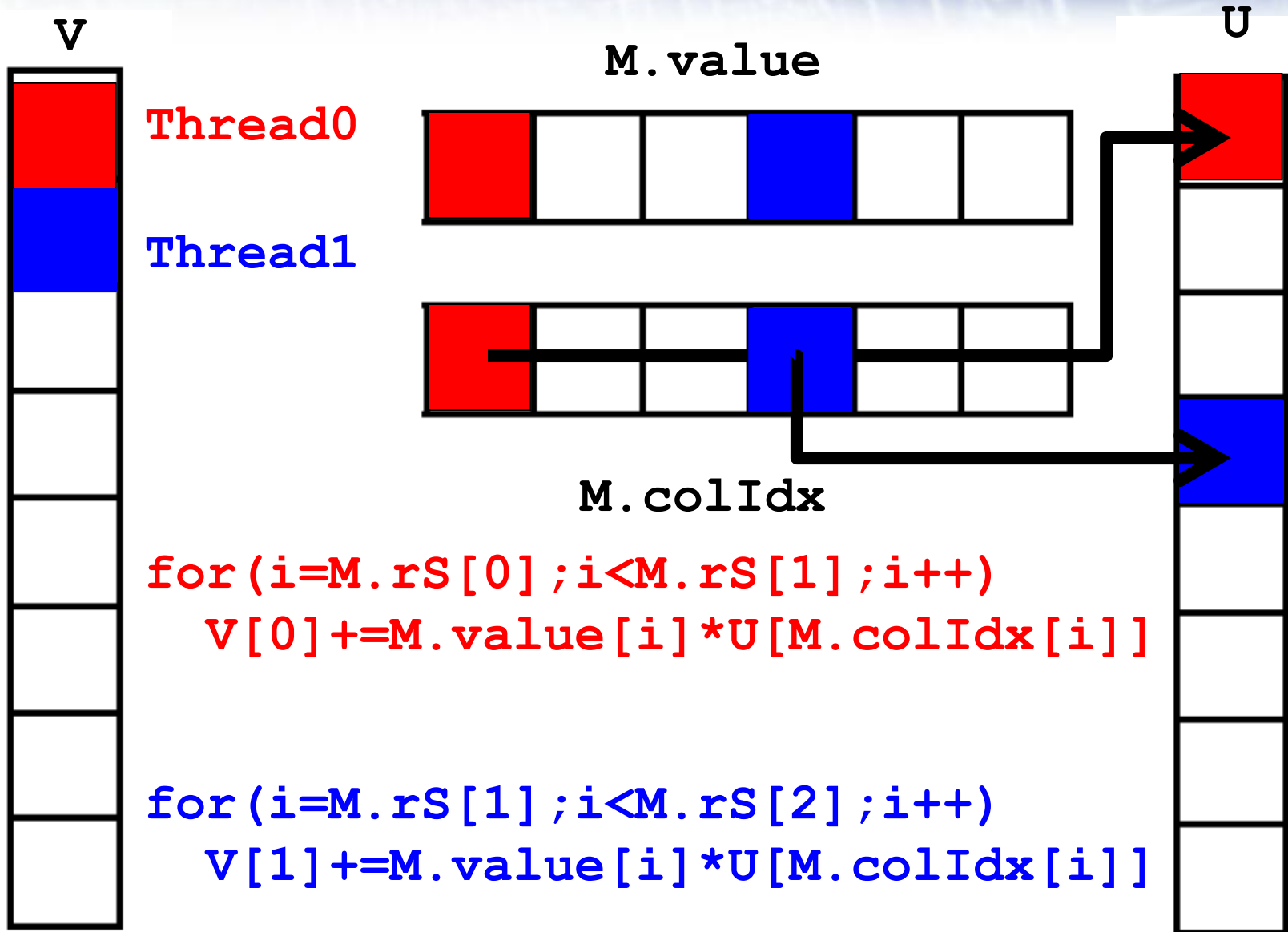
$$u_t(k) = \sum_j B_j * u_{t-1}(j)$$

- For GPUs \rightarrow need synchronise blocks *every time-step*

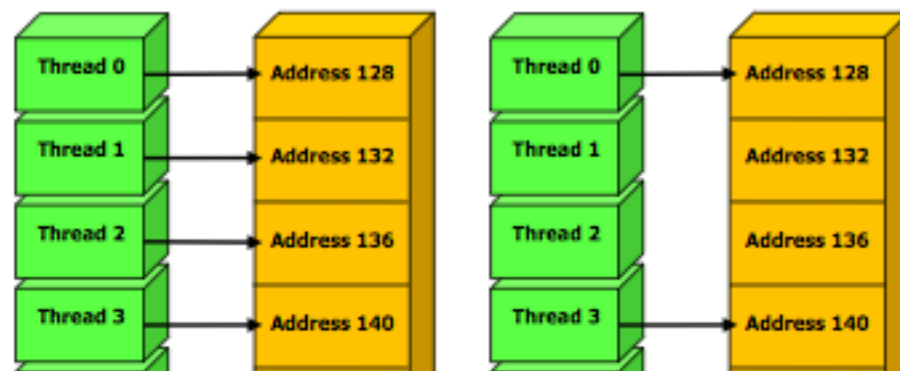
- There are several SpMV libraries implemented for GPUs
- Elected not to use them
- Performance of libraries depends on the matrix
 - e.g. Vázquez *et al* (2009), Proc. CMMSE, pp. 1081–1092
 - Speed up 5 – 80 depending on matrix
- These focus on generic data structures
 - Can't know the matrix structure in advance
- Domain knowledge
 - We know what the matrix is
 - Can design program accordingly



- Matlab has sparse matrix data format
- two-stage port via C
 - Check correctness of compiled code
- Originally used compressed row storage format
- 3 1-d arrays
 - float value(nnz) number of non-zeros
 - int colIdx(nnz) column of each non-zero element
 - int rowStart(nrow+1) 1st and last element in each row
- Is efficient for matrix-vector
 - double indexing → non-coalesced memory access in GPUs



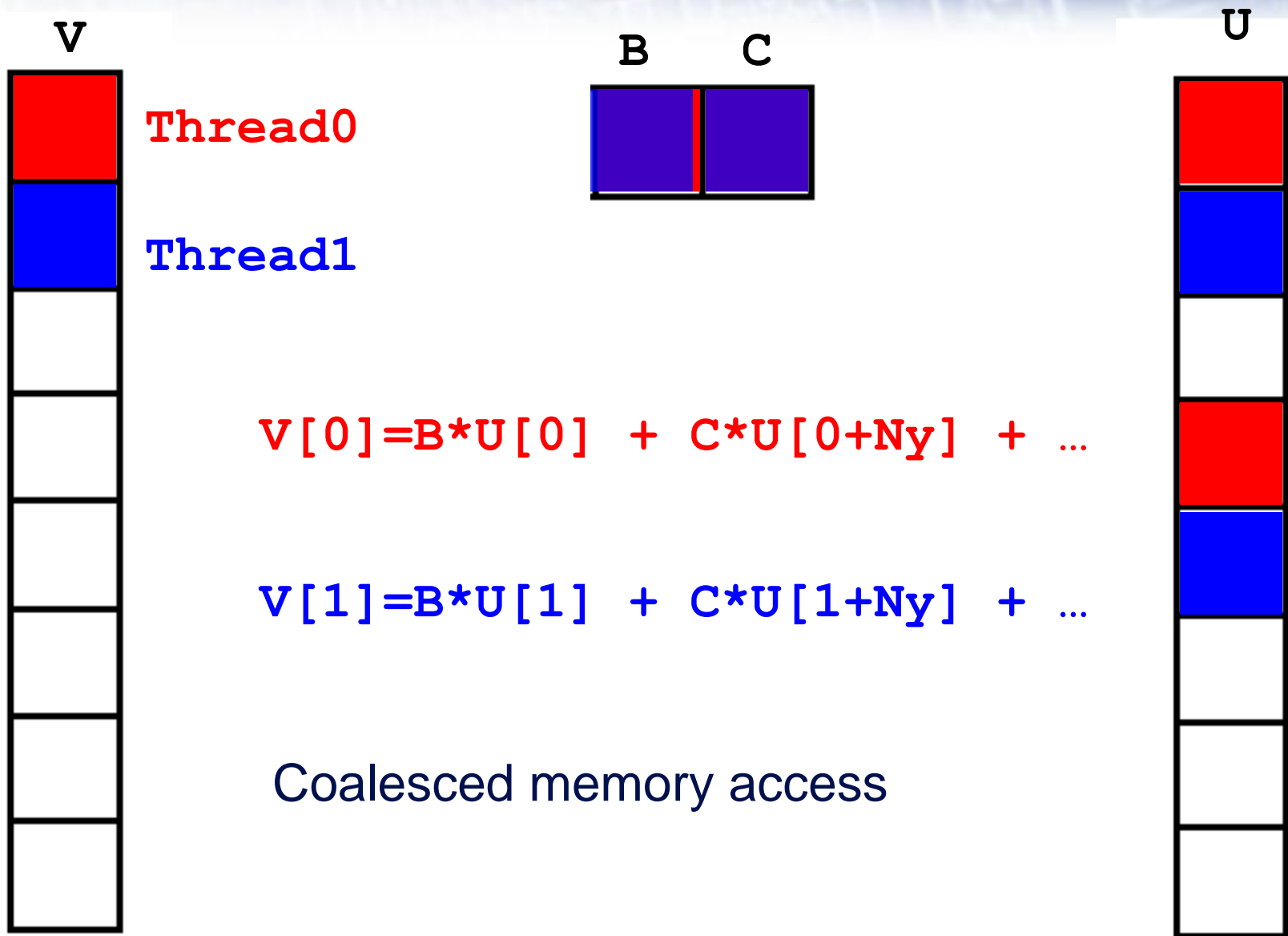
- Key to fast performance is fast memory access
 - copy the data into shared memory
 - 4MB per block/only within block
 - Use coalesced global memory access
 - re-write the algorithm for aligned access
 - Use texture memory
 - Read-only cache
 - Globally visible



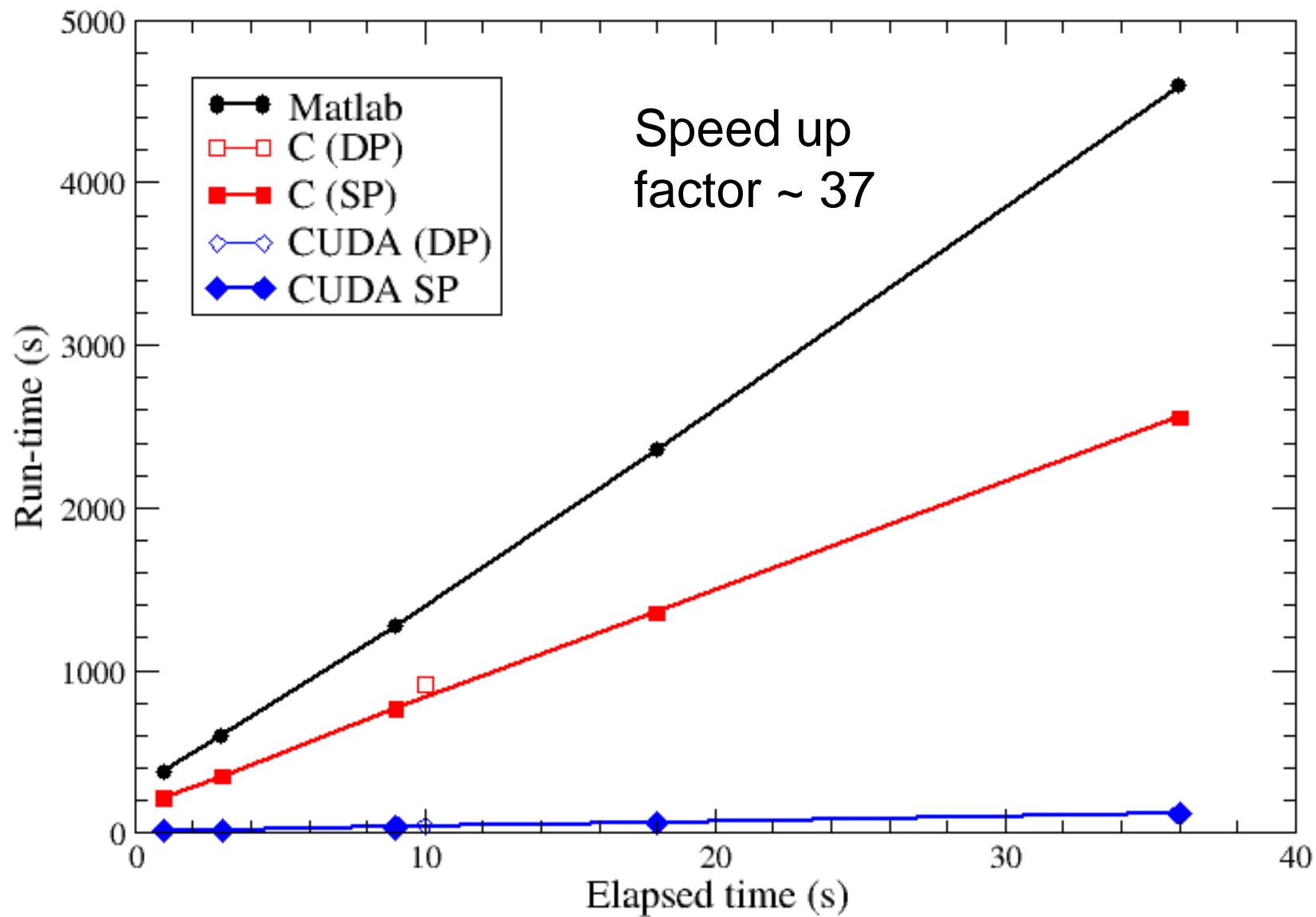
- Coalesced memory access for floats
 - (NVIDIA CUDA programming guide)
- Threads access contiguous memory
- Single memory transaction

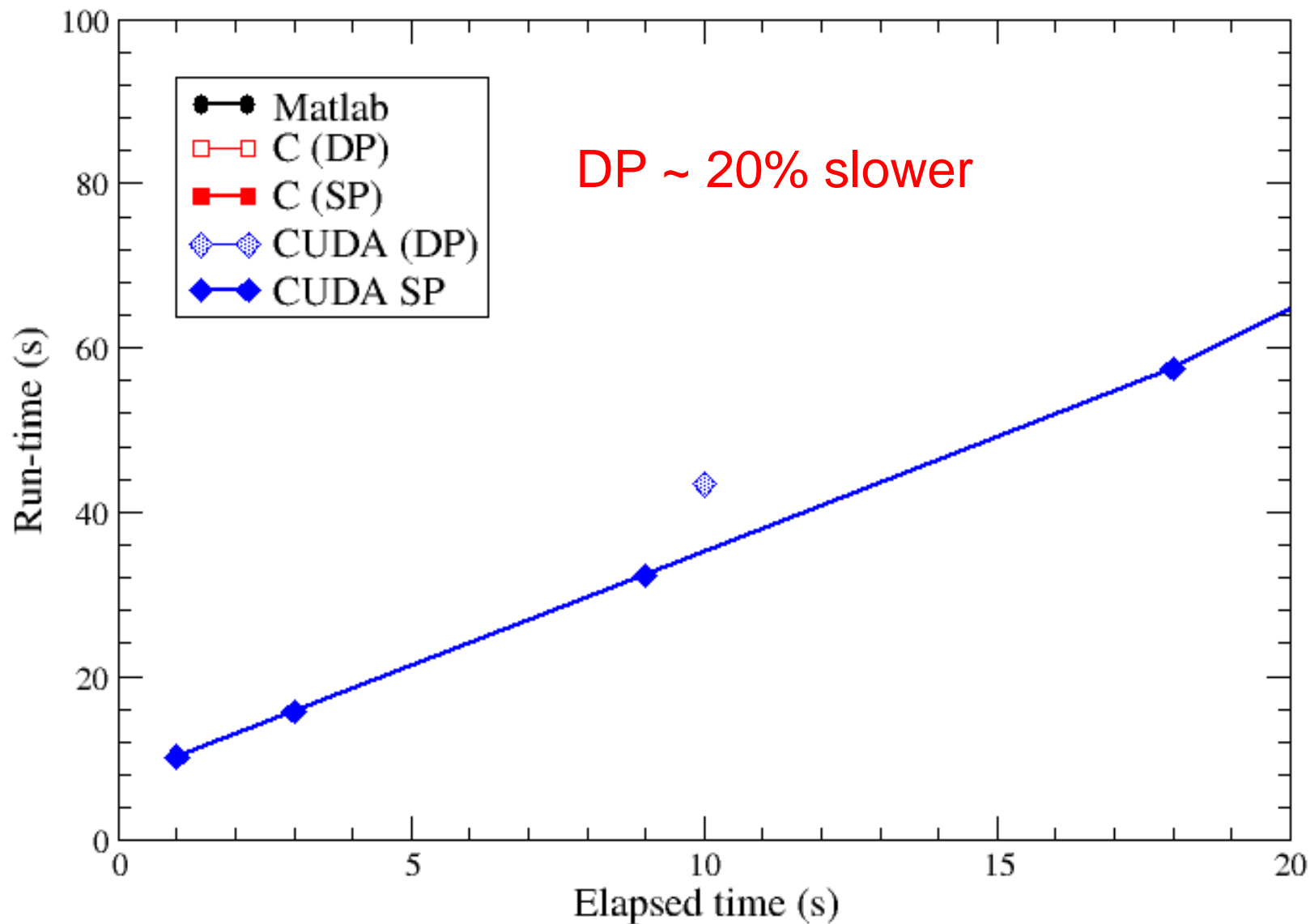
- Don't store the matrix
- Replace the matrix with a function
 - 5 parameters in this case
- Encode the stencil “by-hand”
- Less memory storage
 - use registers
 - matrix stored everywhere!
- Better memory access
 - coalesced access for the vectors





- Checking code produces correct answer
- Matlab to C was problematic in single precision
 - Sparse matrix data structure in matlab is double precision
 - Answer in C single precision dependent on rounding errors in parameters
 - Calculate parameters in double precision
- Used Fourier transform of signal output for comparison
 - Same in double
 - Equivalent in single
- C to CUDA was simpler
 - Single and double agreed





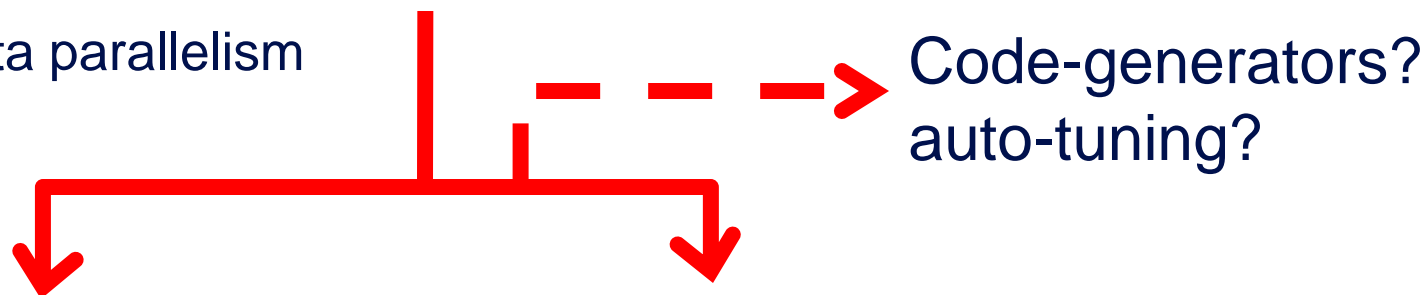
- Focussed on single precision
 - Peak performance **single:double** ~ 10
- Application required double precision
 - spent a lot of time checking and fixing up single precision
- Memory bounds
 - FpW ~ **37 single** FpW ~ **6 double**
- Unlikely we saturated double bound
 - Single precision was a red-herring
- Loading kernels onto GPU wasn't that expensive
- Loading kernel at each time step to synchronise blocks wasn't that expensive



- three month project porting codes to CUDA
 - achieved ~ 37 speed up SP ~ 30 DP
- Future work
 - 3 D problem + block Toeplitz structure
 - This is much harder

- To exploit highly parallel architecture

- exploit data parallelism



- Exploit domain knowledge
 - Know what the matrix is
 - Code by hand
 - not portable / not re-useable
- Use library
 - Data parallelism not explicit
 - easier to code
 - Re-usable