Imperial College London

Professor Paul H. J. Kelly, Professor of Software Technology

Inaugural lecture: Over and over again: the discipline of parallel software engineering

In the chair: Professor Jeff Magee, Head of Department, Department of Computing , Imperial College London

Vote of Thanks: Professor Christian Lengauer, University of Passau, Germany

- Making programs go faster
- Parallel programming
- Controlling complexity
- Some of my prior work and its connection to this agenda
- A manifesto for carrying this forward

Acknowledgements

All the hard work was done by other people

- Andrew Bennett, Frank Taylor, Sergio Almeida, Ariel Burton, Sarah Bennett, Olav Beckmann, Kwok Yeung, David Pearce, Jeyarajan Thiyagalingam, Junxian Liu, Ashley Saulsbury, Qian Wu
- Anton Lokhmotov, Lee Howes, Francis Russell, Jay Cornwall, Ashley Brown, Peter Collingbourne, Michael Mellor, Thanasis Konstantinidis
- Richard Jones, Alastair Houghton, Henry Falconer, Karen Osmond, Marc Hull, Thomas Hansen, Jacob Refstrup, Doug Brears, Thiebaut Weise
- Tony Field, Chris Hankin, Wayne Luk, David Bolton, Peter Osmon, John Darlington, Peter Harrison, Sebastian Hunt, Ross Paterson
- Bruno Nicoletti, Phil Parsonage, Robert Berry, Alastair Donaldson, Scott Baden, Gerard Gorman Paul Anderson, Tim Wilkinson, Phil Winterbottom, Tom Stiemerling, Kevin Murray
- Richard and Clarissa Stevenson

- Past PhD students and research group members
- Current research group members
- Many, many project and UROP students
- Fellow academics
- Collaborators

Imperial College London

Acknowledgements

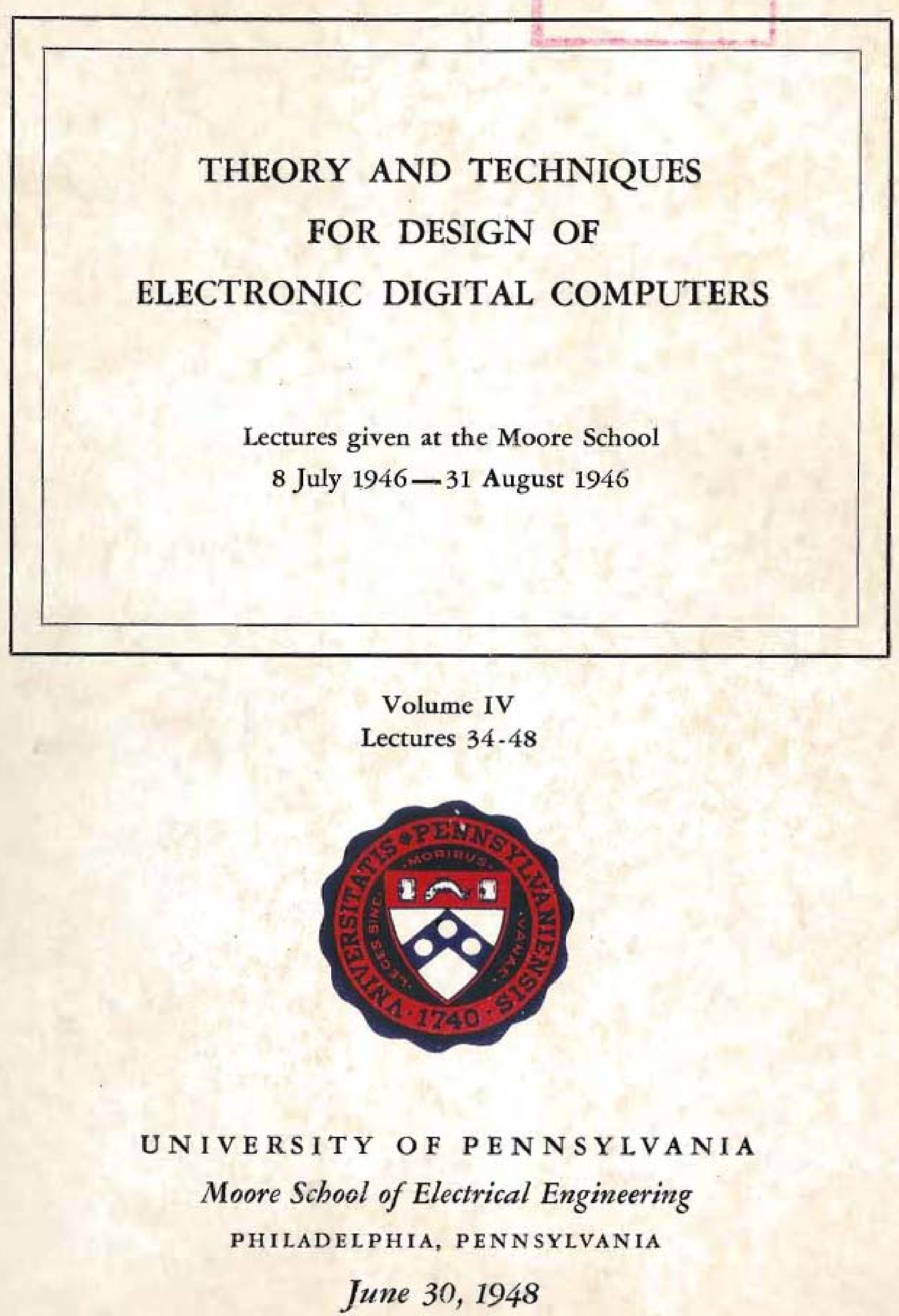
Research funding:

The research presented here has been, or is being, funded by:

- EPSRC
- IBM (Faculty Award, Industrial CASE)
- Microsoft
- The Foundry (Industrial CASE)
- Codeplay (Industrial CASE)
- Arup (Industrial CASE)

Thank you for your support!

The Moore School Lectures


THEORY AND TECHNIQUES
FOR DESIGN OF
ELECTRONIC DIGITAL COMPUTERS

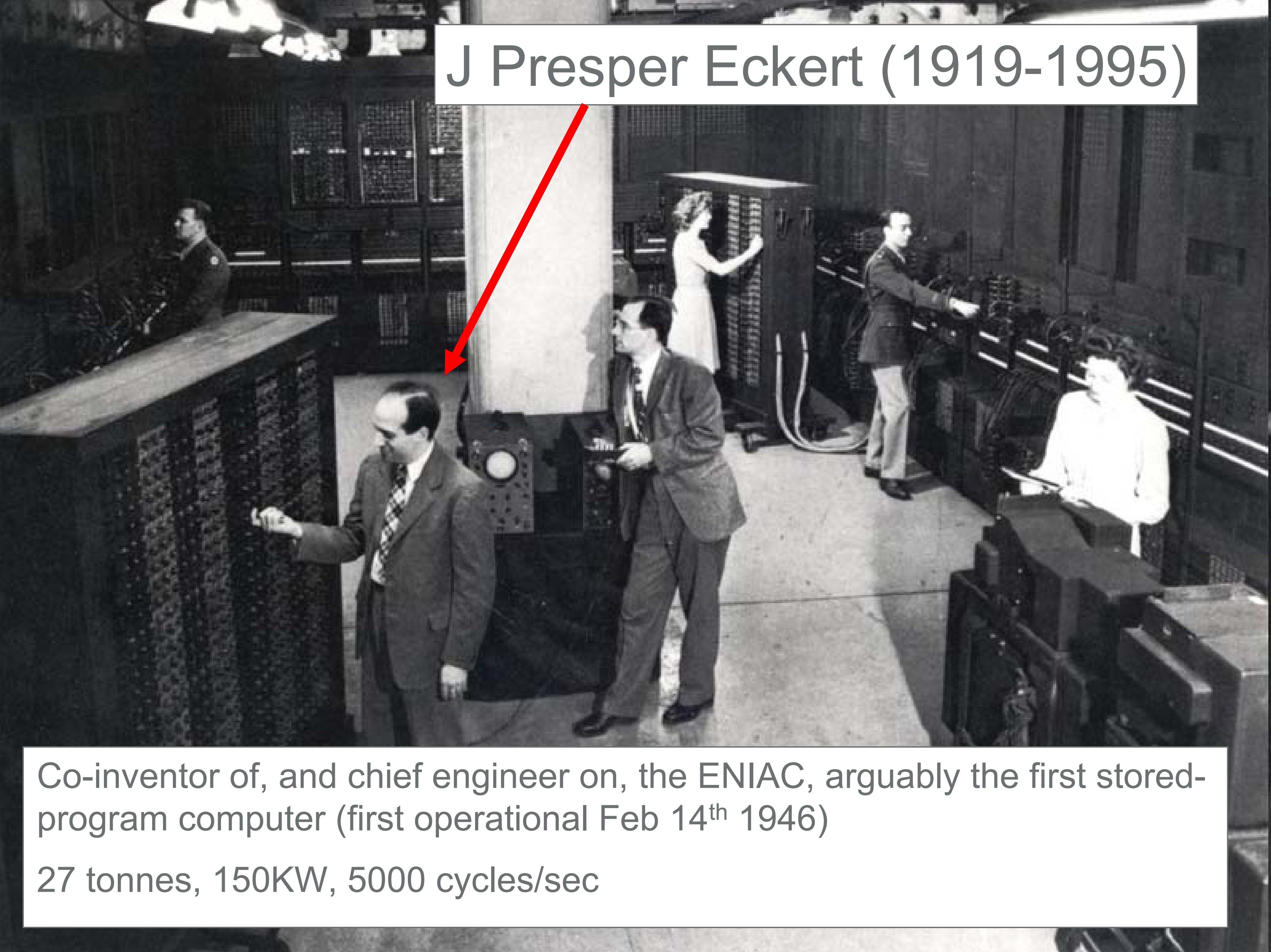
Lectures given at the Moore School
8 July 1946—31 August 1946

Volume IV
Lectures 34-48

UNIVERSITY OF PENNSYLVANIA
Moore School of Electrical Engineering
PHILADELPHIA, PENNSYLVANIA
June 30, 1948

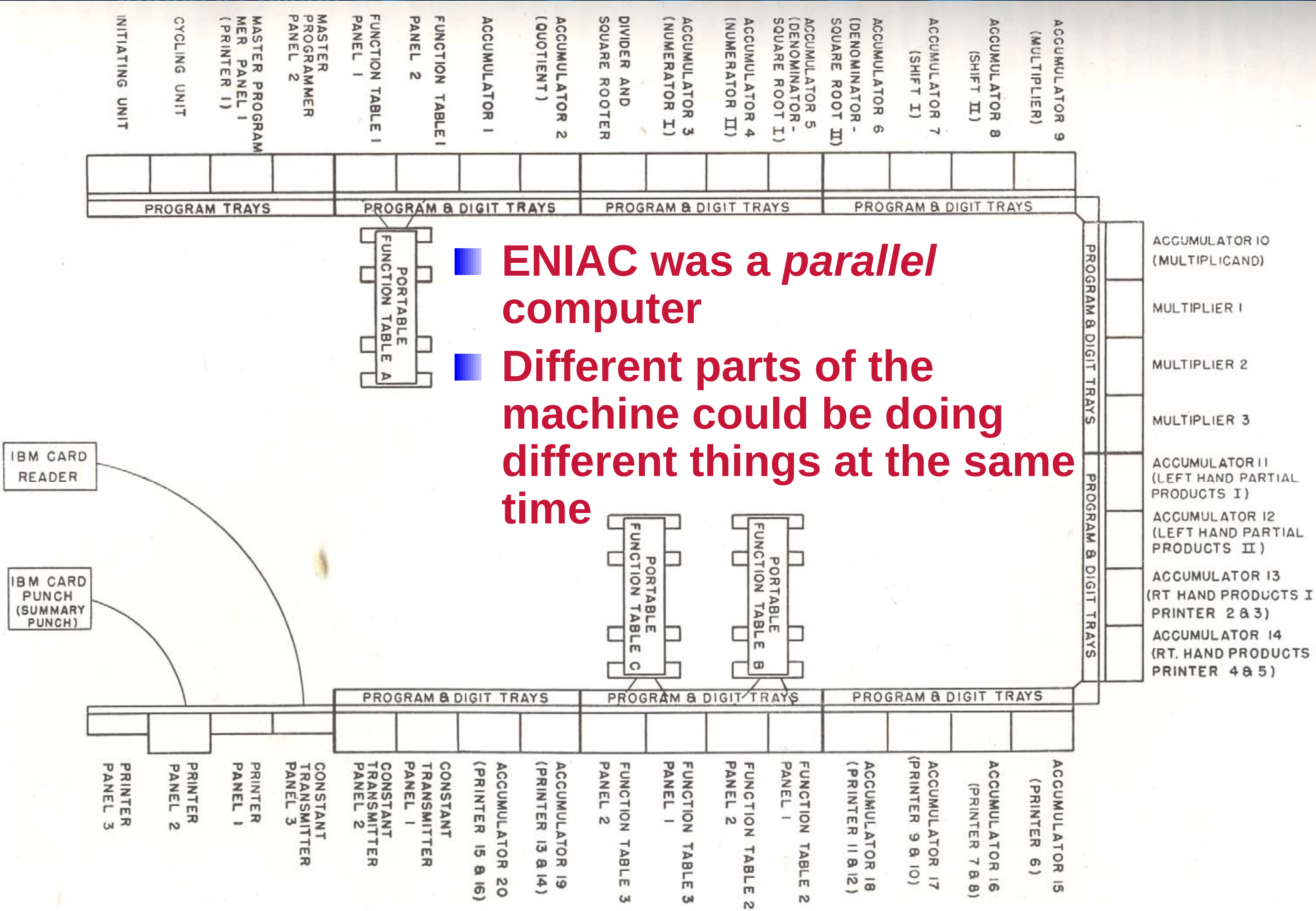
- The first ever computer architecture conference
- July 8th to August 31st 1946, at the Moore School of Electrical Engineering, University of Pennsylvania
- A defining moment in the history of computing
- *To have been there….*

J Presper Eckert (1919-1995)



Co-inventor of, and chief engineer on, the ENIAC, arguably the first stored-program computer (first operational Feb 14th 1946)

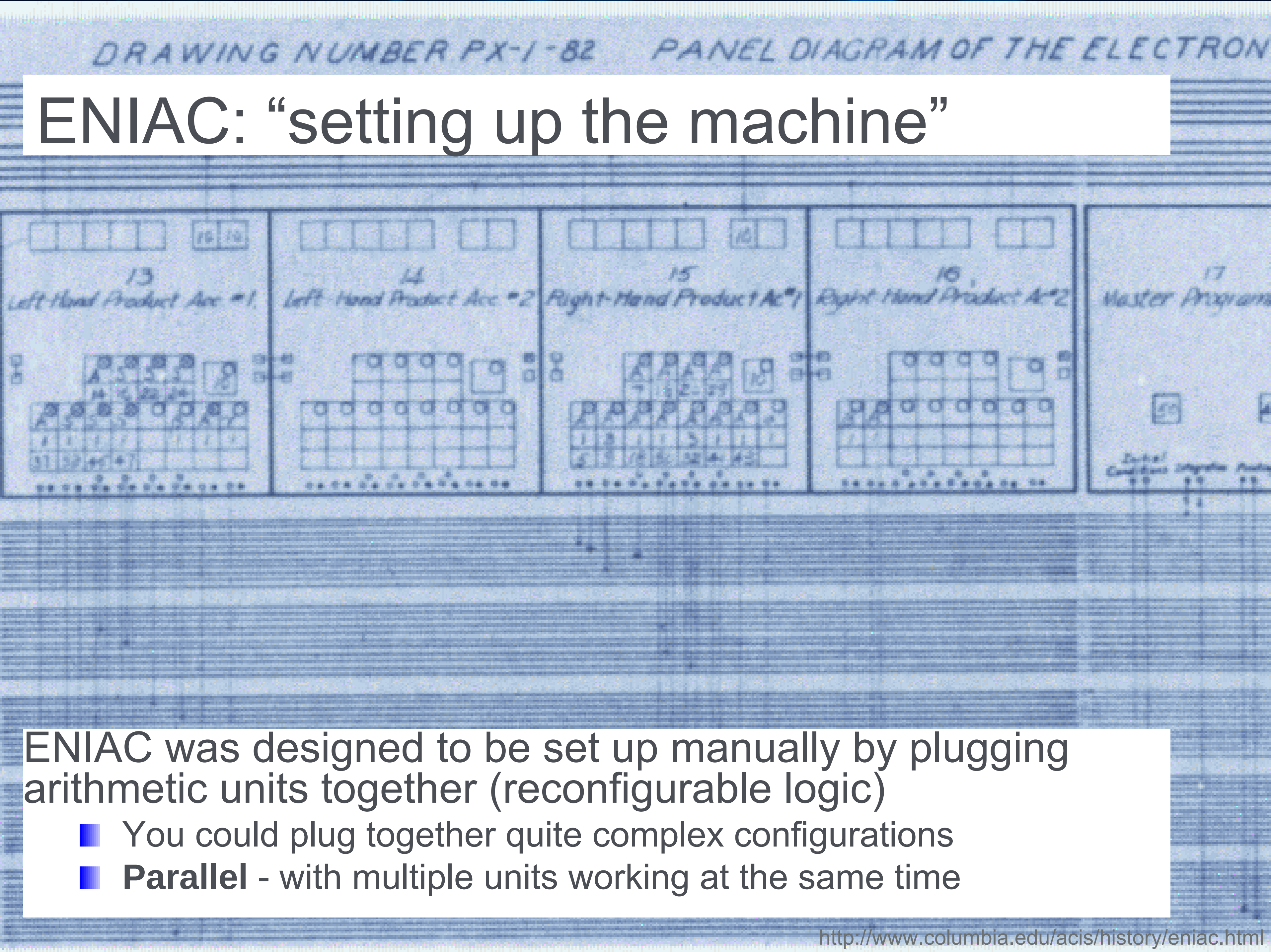
27 tonnes, 150KW, 5000 cycles/sec



- **ENIAC was a *parallel* computer**
- **Different parts of the machine could be doing different things at the same time**

J.G. Brainerd & T.K. Sharpless. "The ENIAC." pp 163-172 Electrical Engineering, Feb 1948.

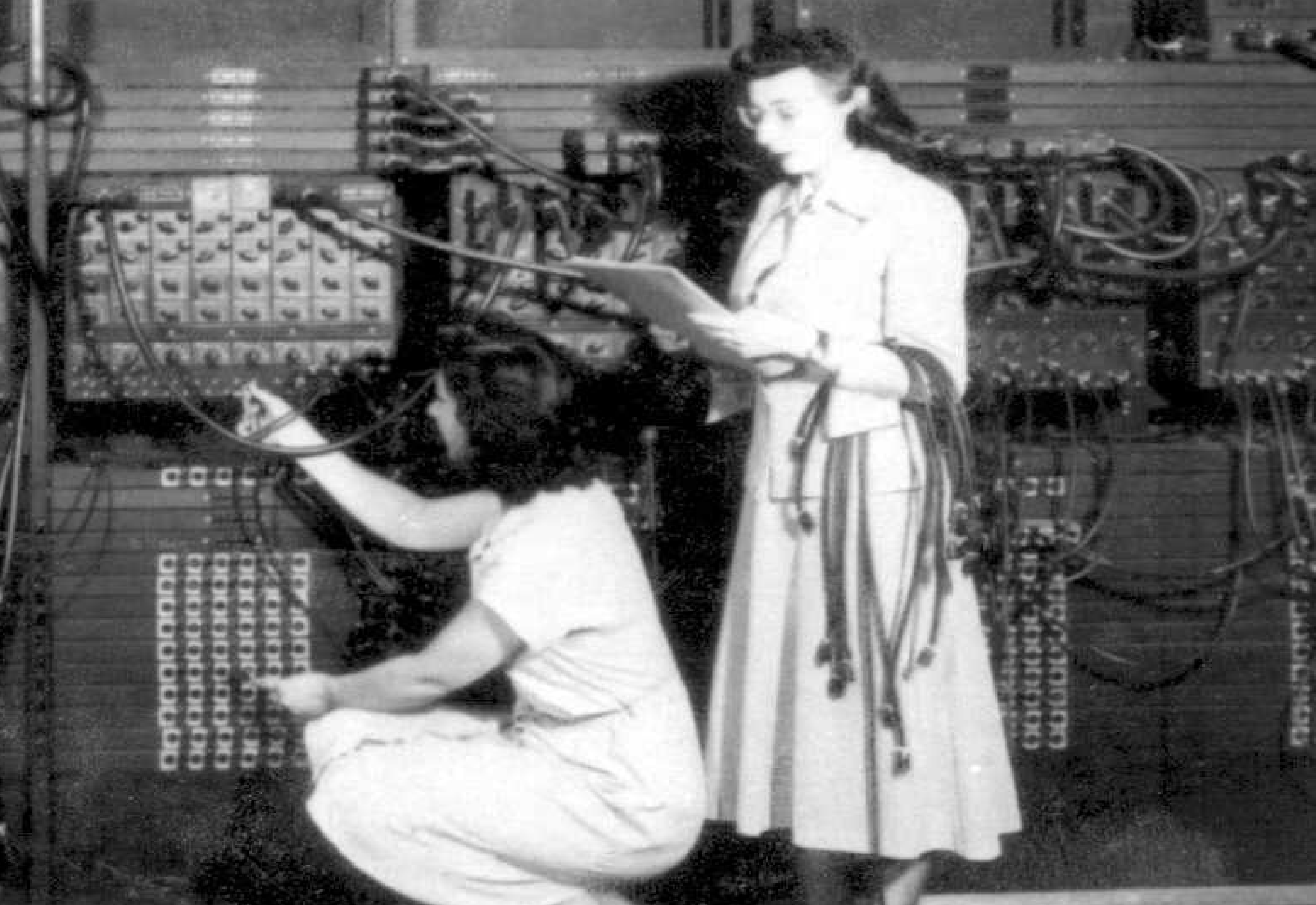
ENIAC: "setting up the machine"



ENIAC was designed to be set up manually by plugging arithmetic units together (reconfigurable logic)

- You could plug together quite complex configurations
- **Parallel** - with multiple units working at the same time

http://www.columbia.edu/acis/history/eniac.html



Gloria Gorden and Ester Gerston: programmers on ENIAC

http://www.columbia.edu/acis/history/eniac.html

LECTURE 45 26 AUGUST 1946

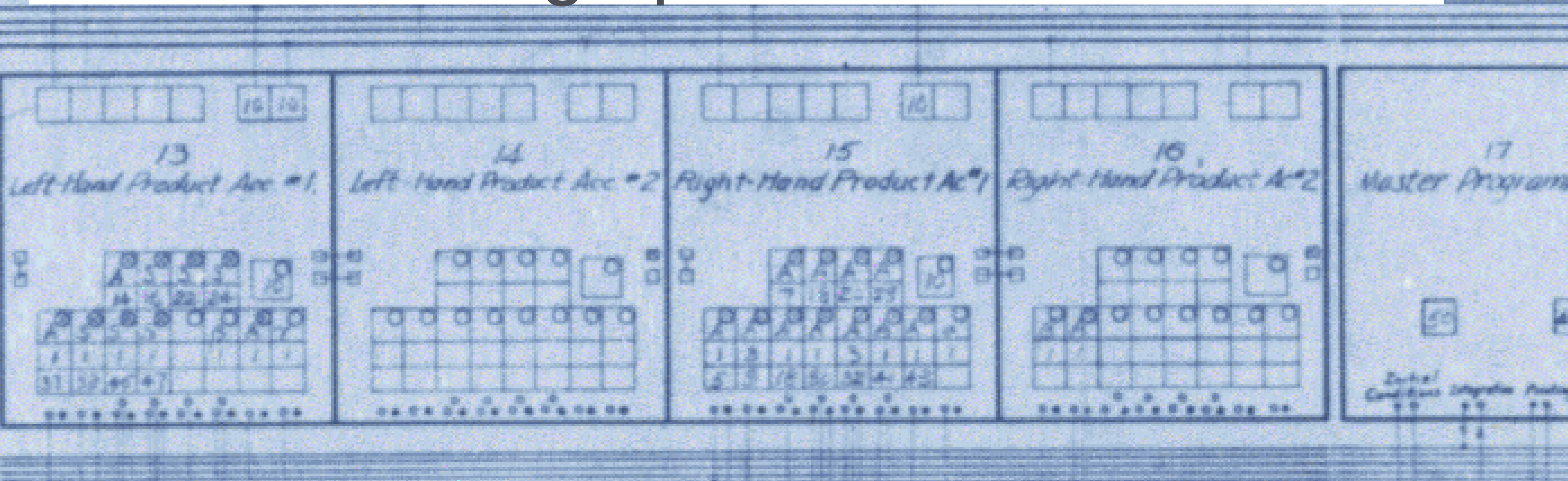
A PARALLEL CHANNEL COMPUTING MACHINE

Lecture by
J. P. Eckert, Jr.
Electronic Control Company

. . . Again I wish to reiterate the point that all the arguments for parallel operation are only valid provided one applies them to the steps which the built in or wired in programming of the machine operates. Any steps which are programmed by the operator, who sets up the machine, should be set up only in a serial fashion. It has been shown over and over again that any departure from this procedure results in a system which is much too complicated to use.

See also http://www.digital60.org/birth/themooreschool/lectures.html#l45

ENIAC: “setting up the machine”



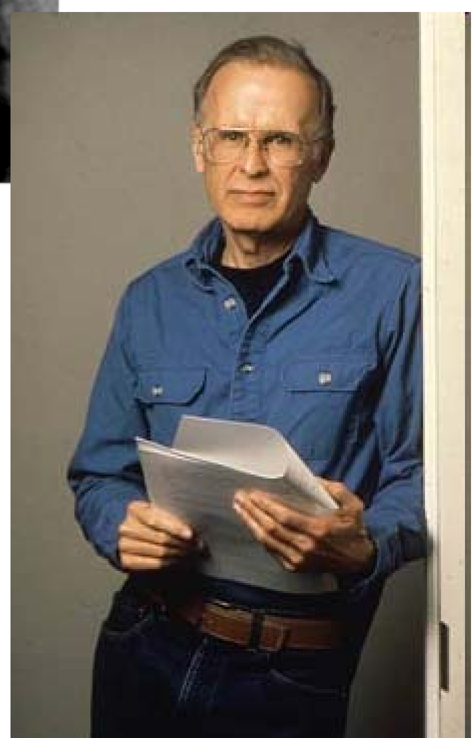
- The “big idea”: stored-program mode -
  - Plug the units together to build a machine that fetches instructions from memory - and executes them
  - So any calculation could be set up completely automatically – just choose the right sequence of instructions

http://www.columbia.edu/acis/history/eniac.html

The "von Neumann bottleneck"

John von Neumann

http://en.wikipedia.org/wiki/John_von_Neumann



John Backus

"Can Programming be Liberated from the von Neumann Style?" *(1979)*

www.post-gazette.com/pg/07080/771123-96.stm

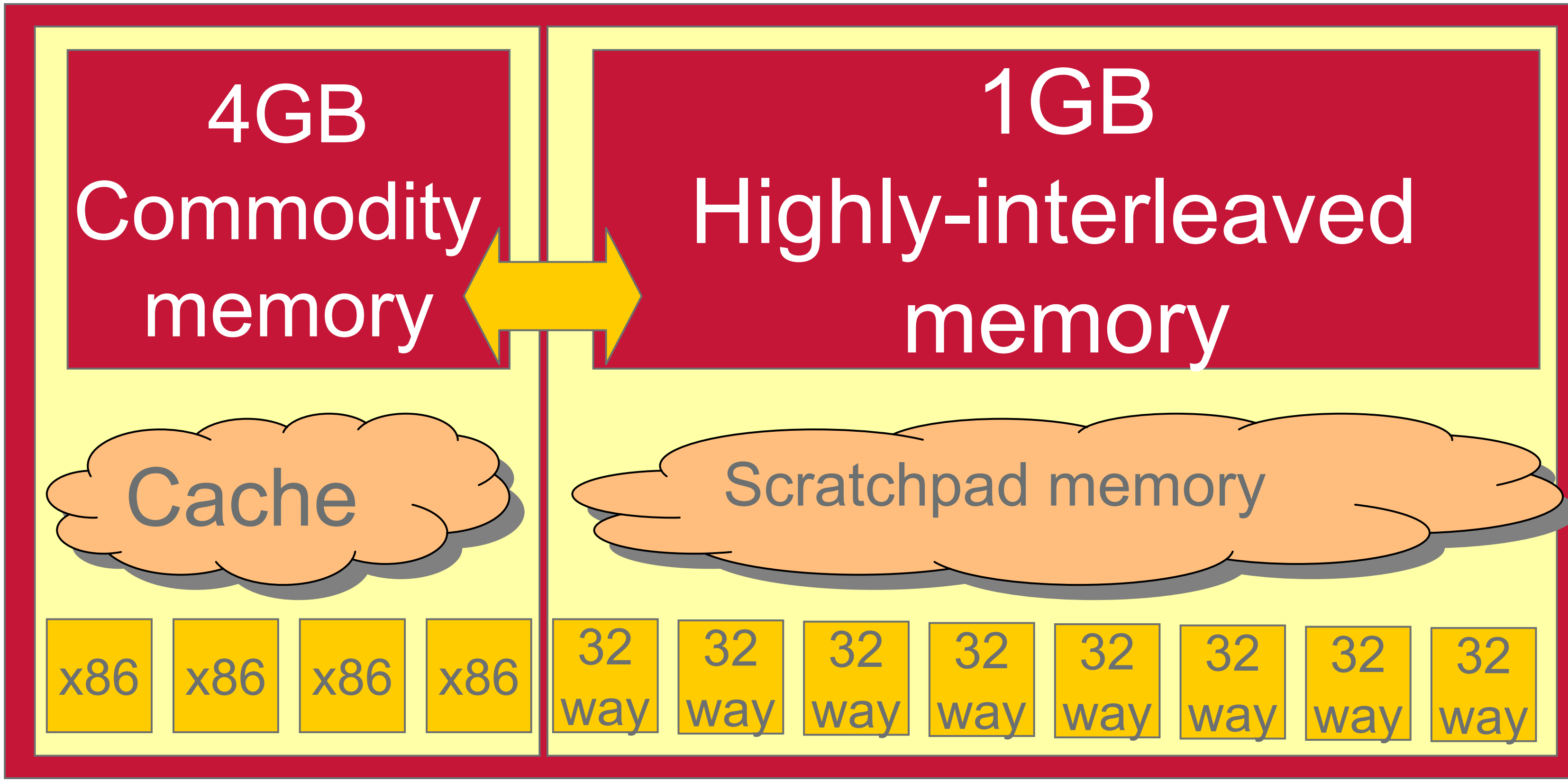
The price to pay:

- **Stored-program mode was serial – one instruction at a time**

- How can we have our cake - and eat it?

  - **Flexibility and ease of programming**
  - **Performance of parallelism**

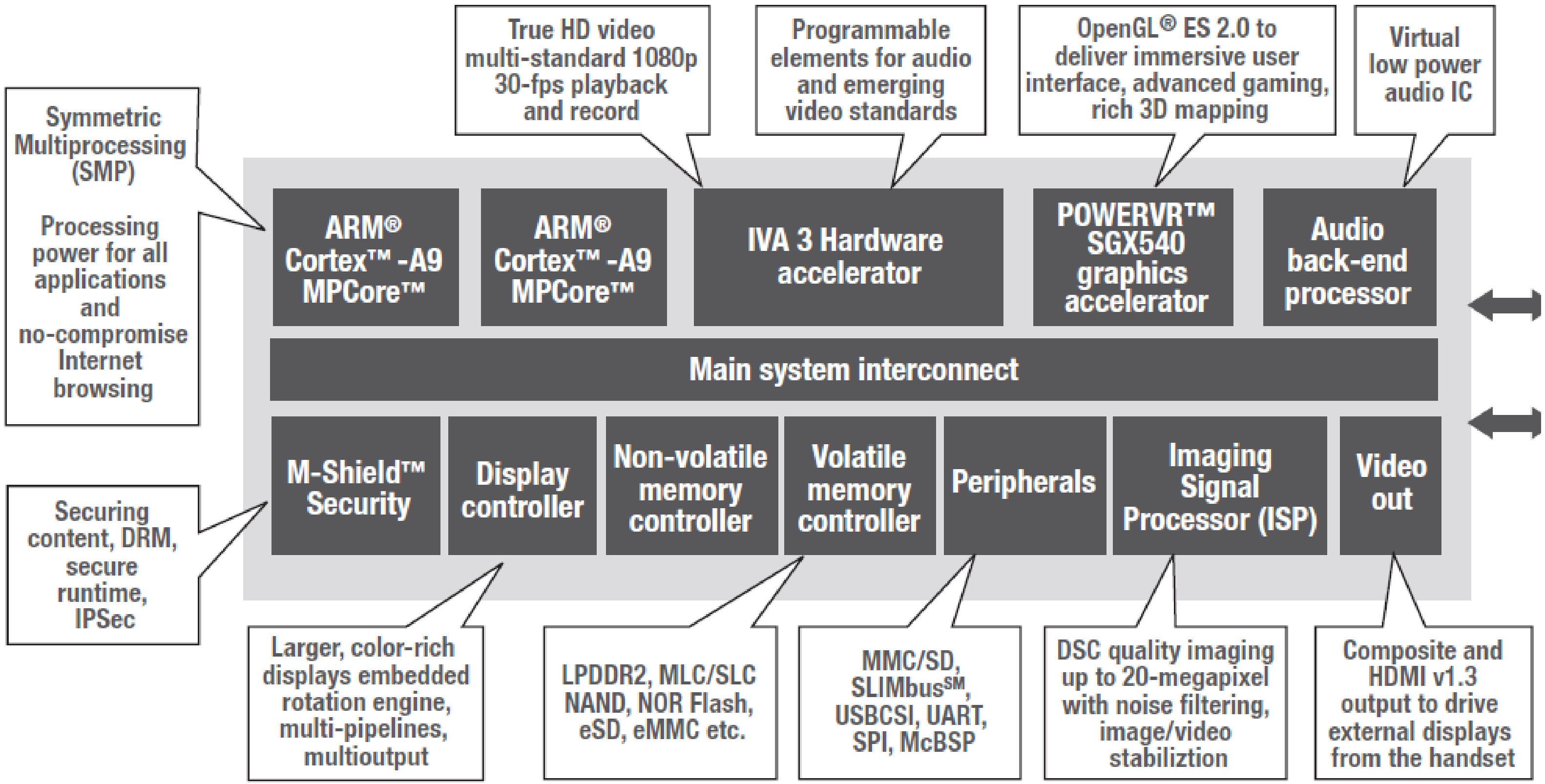
Does parallelism matter?



Typical 2009 personal computer

- 2- to 8-way multicore CPU:
  - Each core executes 2- to 4-wide parallel SSE instructions
- Attached programmable graphics processor is also highly parallel:
  - Typically 8 cores, each executing a 32-wide “warp” of instructions

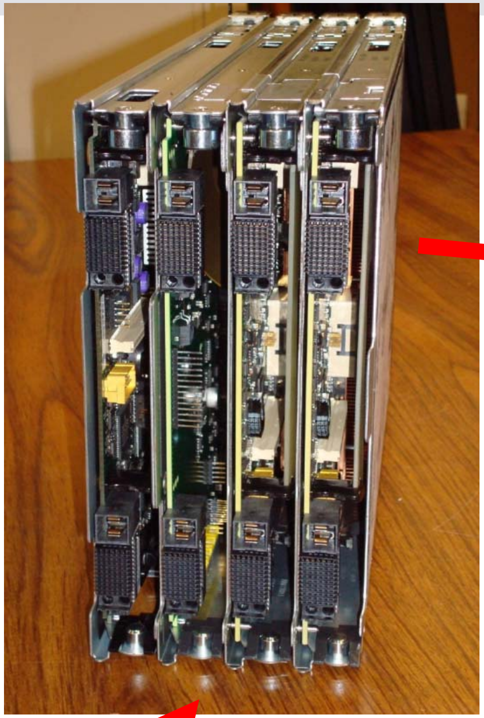
Parallelism is everywhere



- Texas Instruments OMAP4 Mobile Applications Platform
- Two ARM cores + programmable graphics processor + other more specialised accelerators
- To appear in 2010 smart phones and mobile internet devices

http://focus.ti.com/docs/solution/folders/print/501.html

Lots of parallelism…



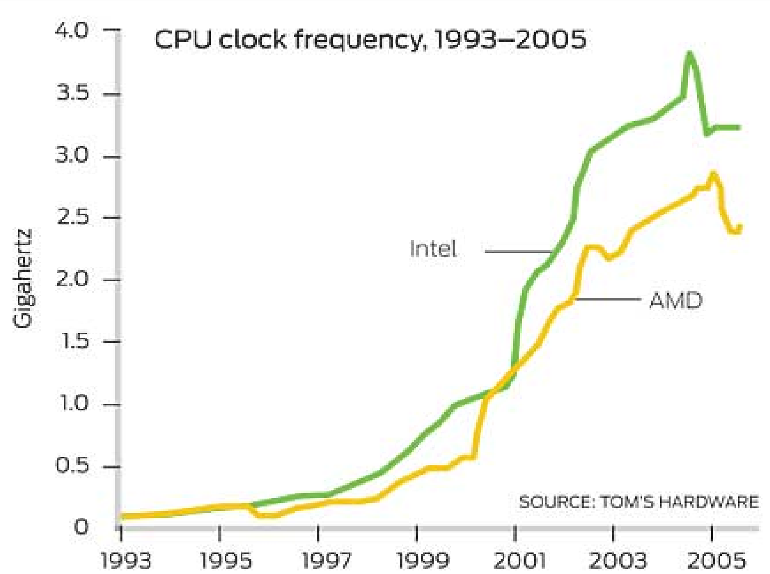
http://www.lanl.gov/orgs/hpc/roadrunner/index.shtml and http://www.infoworld.com

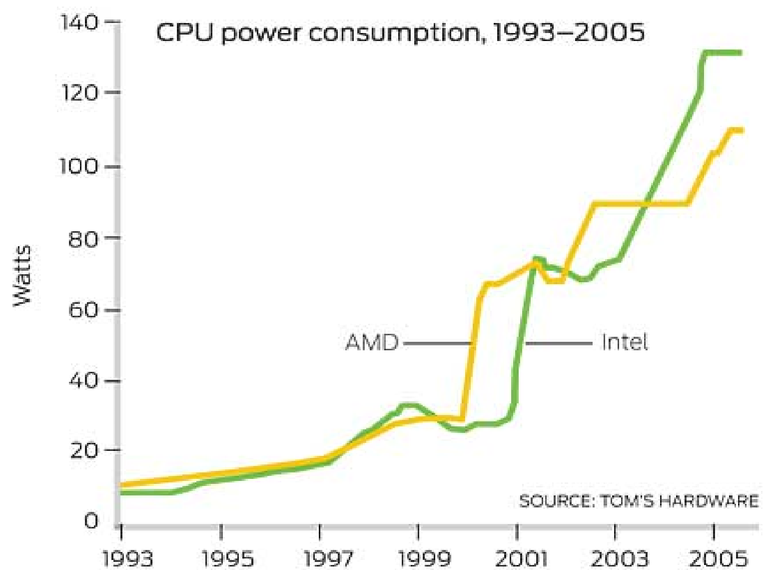
- RoadRunner being built by IBM for Los Alamos National Lab
- 3,456 TriBlades: Two dual-core Opterons + four IBM PowerXCell + interconnect
- 6,120 x86 + 12,240 PowerPC + 97,920 Cell SPEs: 122,400 total (2.35MWatts)
- Record-breaking 1 PetaFLOP (1000 TFLOPs, 10^{12} floating-point calculations per second) achieved in June 08

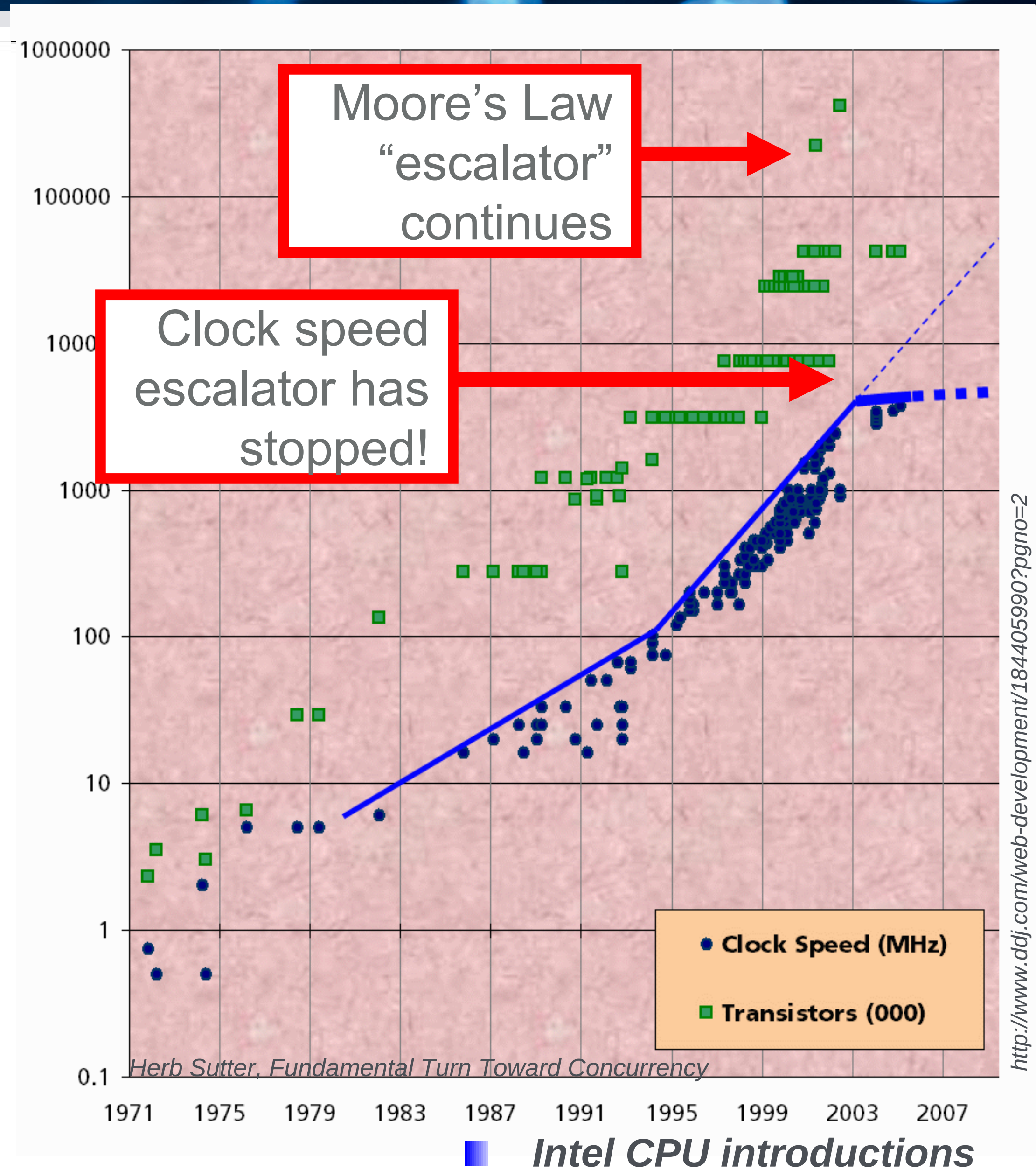
- Computational science simulations demand massive parallelism

Imperial College London

Why? The free lunch is over







Philip E Ross, Why CPU Frequency Stalled - http://www.spectrum.ieee.org/apr08/6106/CPU

Controlling complexity

- But "It has been shown *over and over again*…" that this results in a system too complicated to use

- How can we get the speed and efficiency without suffering the complexity?
- What have we learned since 1946?

Controlling complexity

- But "It has been shown *over and over again*…" that this results in a system too complicated to use
- How can we get the speed and efficiency without suffering the complexity?
- What have we learned since 1946?
  - Compilers and out-of-order processors can extract some instruction-level parallelism
  - Explicit parallel programming in MPI, OpenMP, VHDL are flourishing industries – they can be made to work
  - SQL, TBB, Cilk, Ct (all functional…), many more speculative proposals
  - *No attractive general-purpose solution*

Controlling complexity

- But "It has been shown *over and over again*…" that this results in a system too complicated to use

- How can we get the speed and efficiency without suffering the complexity?
- What have we learned since 1946?
  - Some discipline for controlling complexity
  - Program generation….
    - *Programs that generate programs*
    - *That are correct by construction*
    - *The generator encapsulates parallel programming expertise*

Controlling complexity

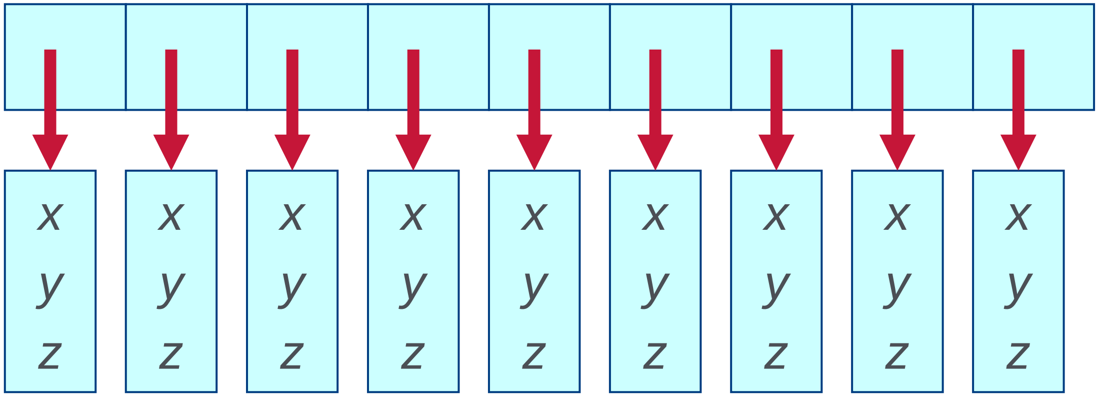
- But "It has been shown *over and over again*…" that this results in a system too complicated to use
- How can we get the speed and efficiency without suffering the complexity?
- What have we learned since 1946?
  - We *really* need parallelism

Easy parallelism

Example:

```
for (i=0; i<N; ++i) {
  points[i]->x += 1;
}
```

- Can the iterations of this loop be executed in parallel?



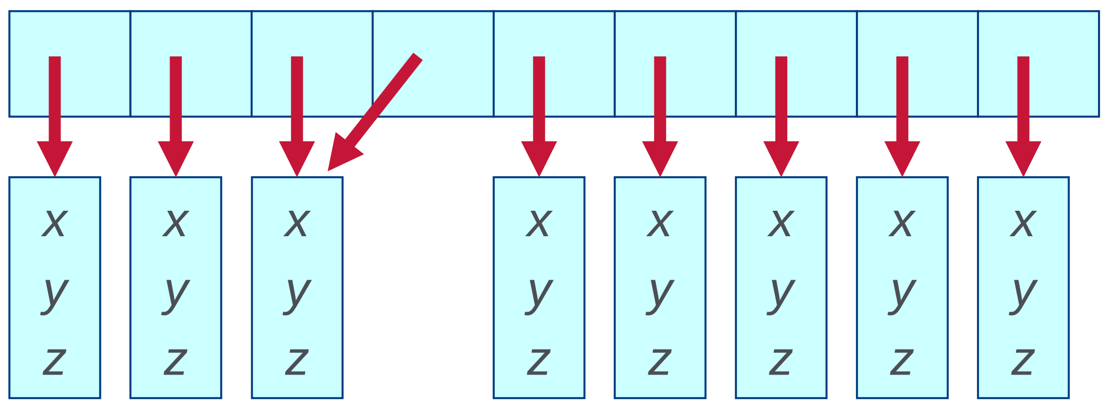
- No problem: each iteration is independent

Easy parallelism

Example:

```
for (i=0; i<N; ++i) {
  points[i]->x += 1;
}
```

- Can the iterations of this loop be executed in parallel?



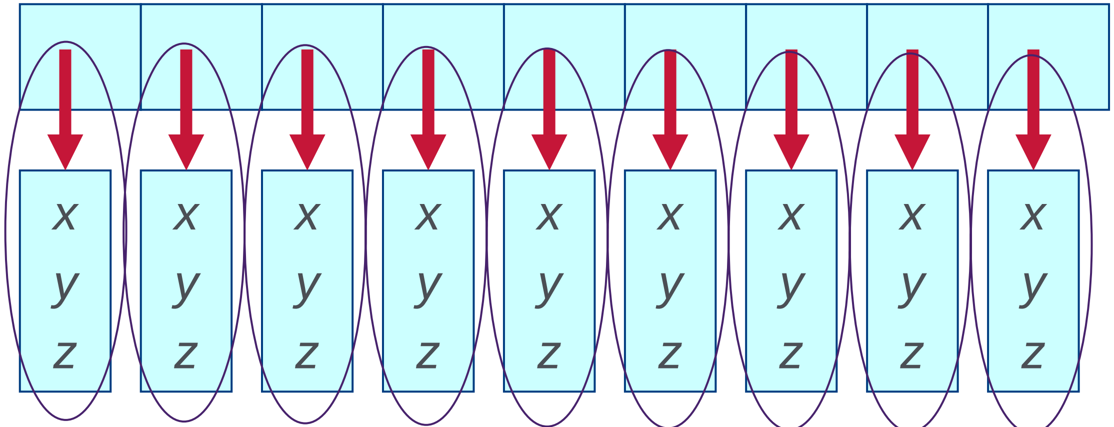
- Oh no: not all the iterations are independent!
  - You want to re-use piece of code in different contexts
  - Whether it's parallel depends on context!

Easy parallelism

Example:

```
for (i=0; i<N; ++i) {
  points[i]->x += 1;
}
```

- Can the iterations of this loop be executed in parallel?



Sergio Almeida's PhD thesis:

"Balloon types" ensure that each cell is reached only by it's owner pointer

Points-to analysis

Thesis work of David Pearce, now at Victoria University, New Zealand

```
int *f(int *p) {
  return p;
}
int g() {
  int x,y,*p,*q,**r,**s;
  s=&p;
  if(...) p=&x;
  else p=&y;
  r=s;
  q=f(*r);
}
```

(1) $f_* \supseteq f_p$

(2) $g_s \supseteq \{g_p\}$

(3) $g_p \supseteq \{g_x\}$

(4) $g_p \supseteq \{g_y\}$

(5) $g_r \supseteq g_s$

(6) $f_p \supseteq *g_r$

(7) $g_q \supseteq f_*$

Variable s of function g might point to variable p of function g

R might point to anything s might point to

f's p might point to anything r might point to

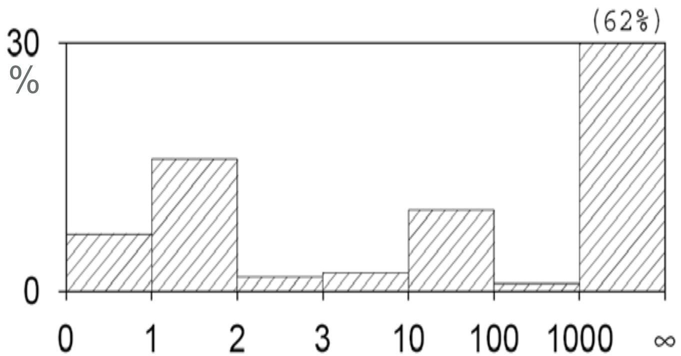
q might point to anything f returns

- Goal: for each pointer variable (p,q,r,s), find the set of objects it might point to at runtime

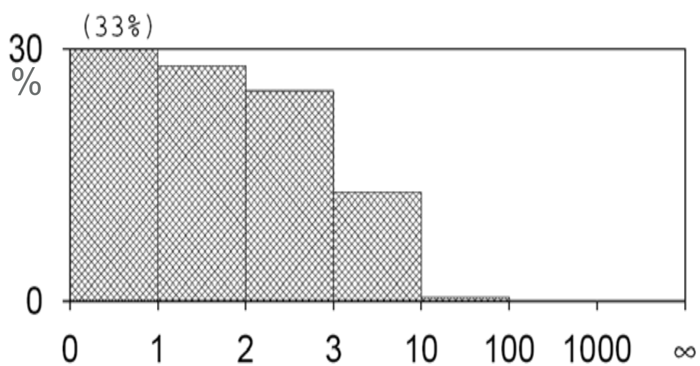
Field-sensitivity in pointer analysis

- We have quite a large constraint graph
  - Eg for 126.gcc from SPEC95:
    - 194K lines of code (132K excl comments)
    - 51K constraint variables (22K of them heap)
    - 7.4K “trivial” constraints
    - 39K “simple” constraints
    - 25K “complex” constraints (due to dereferencing)
- Need to bring together several tricky techniques to get sensible solution times
  - Difference-sets: propagate only changes so you can track what has changed
  - Topological sort: visit nodes in order that maximises solution propagation
  - Cycle detection: zero-weighted cycles can be collapsed
  - Dynamically: dereferencing pointers adds new edges
  - 0.61s for the whole program (900MHz Athlon)

- Histogram of points-to set size at dereference sites for 126.gcc:



- Field *insensitive*



- Field *sensitive*

Field-sensitivity in pointer analysis

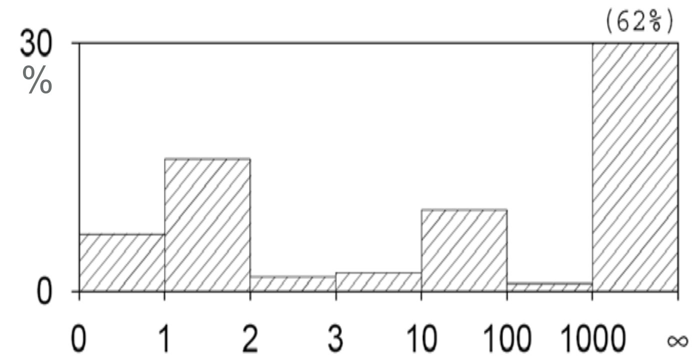
- We have quite a large constraint graph
  - Eg for 126.gcc from SPEC95:
    - 194KLOC (132K without comments etc)
    - 51K constraint variables (22K of them heap)
    - 7.4K “trivial” constraints
    - (due to
- N… cky techniques
  to …
  - changes so you
  - order that
  - cycles can be
  - nters adds new
  - 0.61s for the whole program (900MHz Athlon)

Reimplemented for GCC, the GNU Compiler Collection (by Dan Berlin, of IBM)

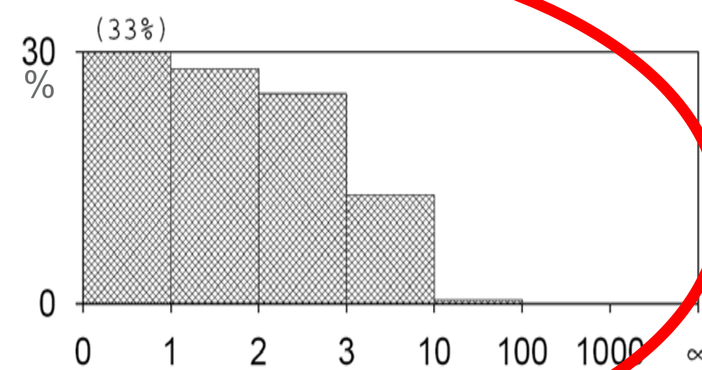
Released the week of David’s PhD defence

David’s paper is cited in the open-source code

- Histogram of points-to set size at dereference sites for 126.gcc:



- Field *insensitive*



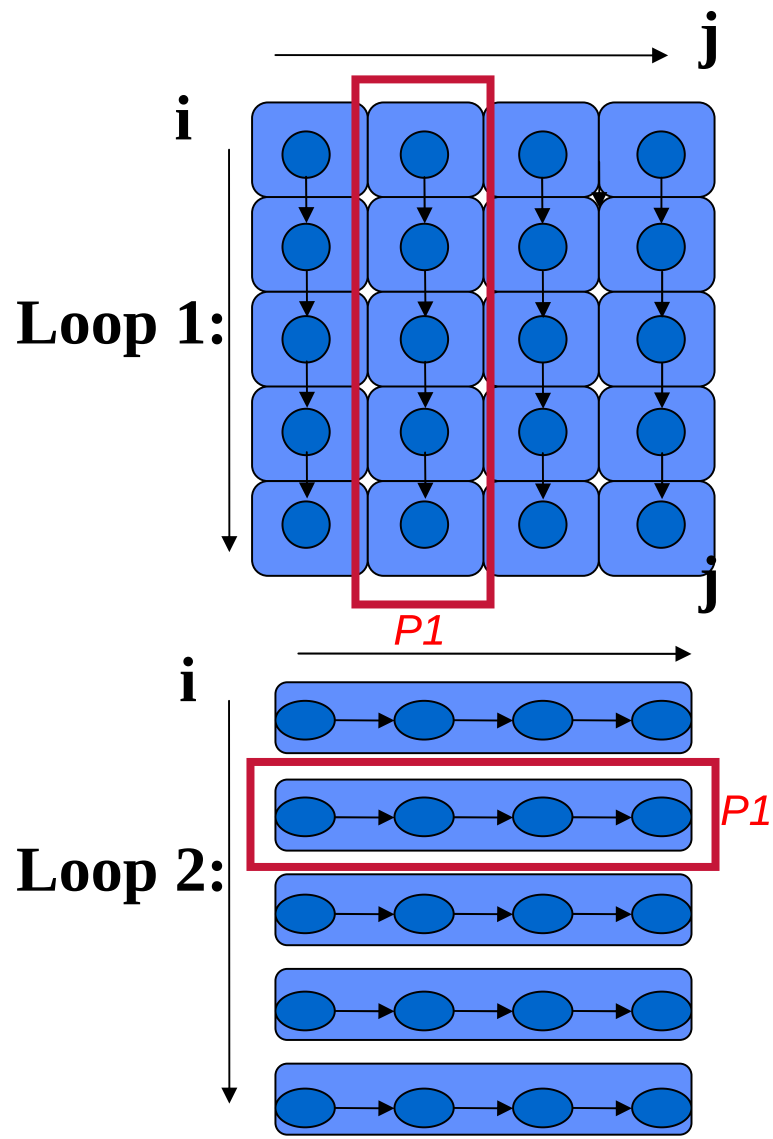
- Field *sensitive*

Imperial College London

Another loss of abstraction…

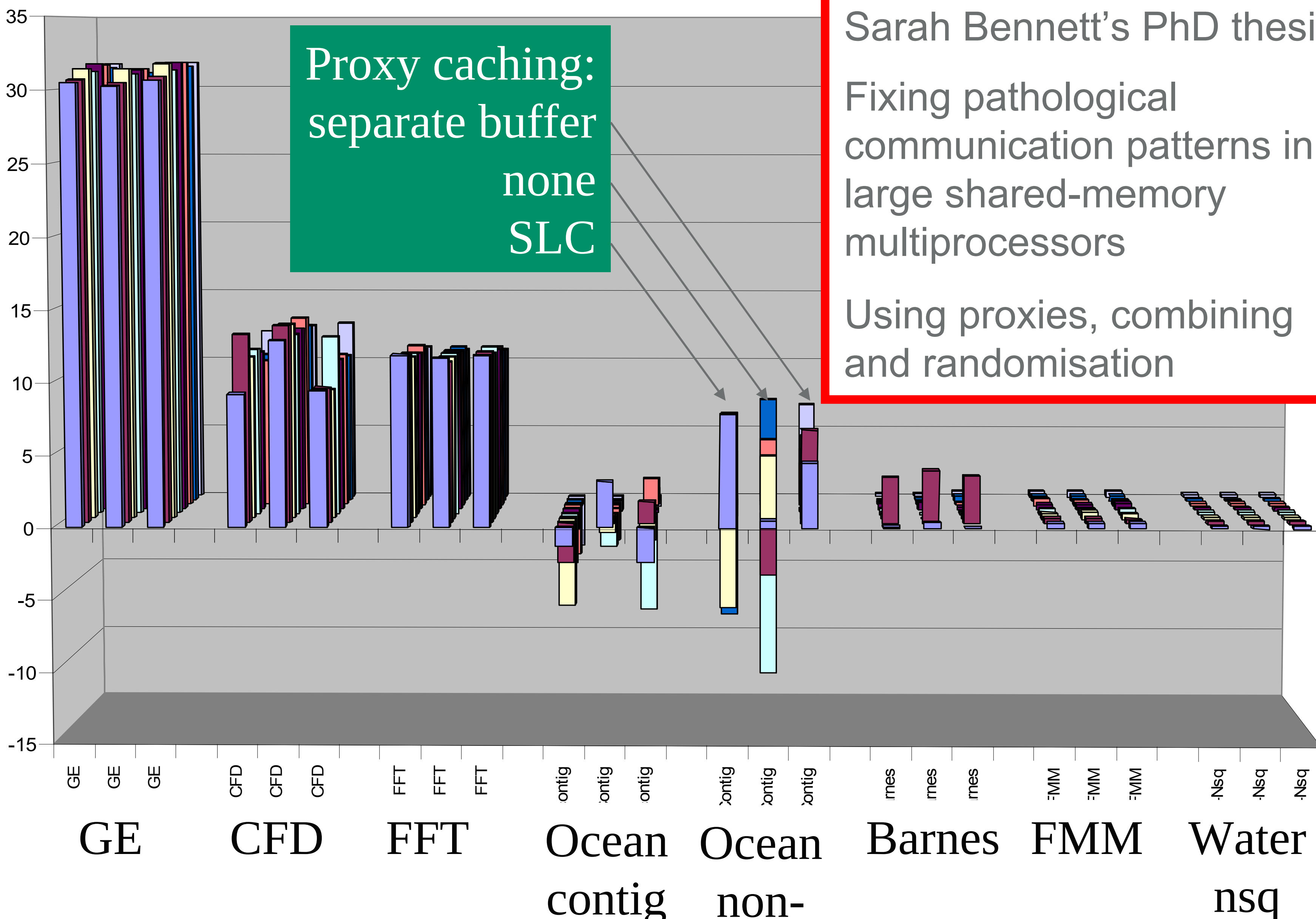
- **Shared memory makes parallel programming much easier:**

```
for(i=0; I<N; ++i)
  par_for(j=0; j<M; ++j)
    A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; I<N; ++i)
  for(j=0; j<M; ++j)
    A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```



- **First loop operates on rows in parallel**
- **Second loop operates on columns in parallel**
- **With distributed memory we would have to program message passing to transpose the array in between**
- **With shared memory… no problem!**

Randomisation & combining in cache-coherency protocols

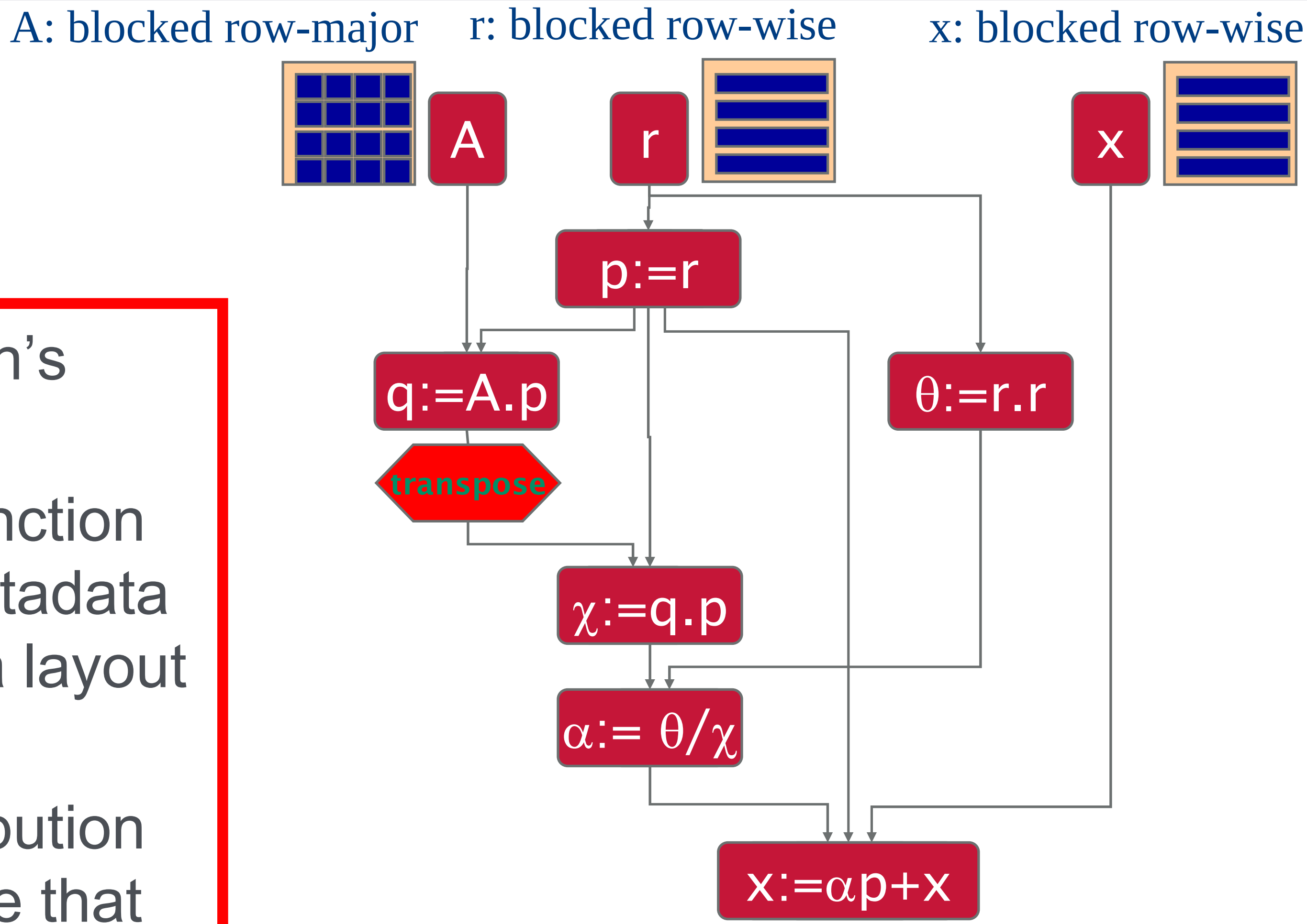


Sarah Bennett's PhD thesis:

Fixing pathological communication patterns in large shared-memory multiprocessors

Using proxies, combining and randomisation

Self-optimising linear algebra library



- Olav Beckmann's PhD thesis:
- Each library function comes with metadata describing data layout constraints
- Solve for distribution of each variable that minimises redistribution cost

Easy parallelism – tricky engineering

- Finding parallelism is usually easy
- Very few algorithms are inherently sequential
  - But if you want a large speedup you need to parallelise almost all of your program
- Parallelism breaks abstractions:
  - Whether code should run in parallel depends on context
  - How data and computation should be distributed across the machine depends on context
- "Best-effort", opportunistic parallelisation is almost useless:
  - Robust software must robustly, predictably, exploit large-scale parallelism

How can we build robustly-efficient multicore software

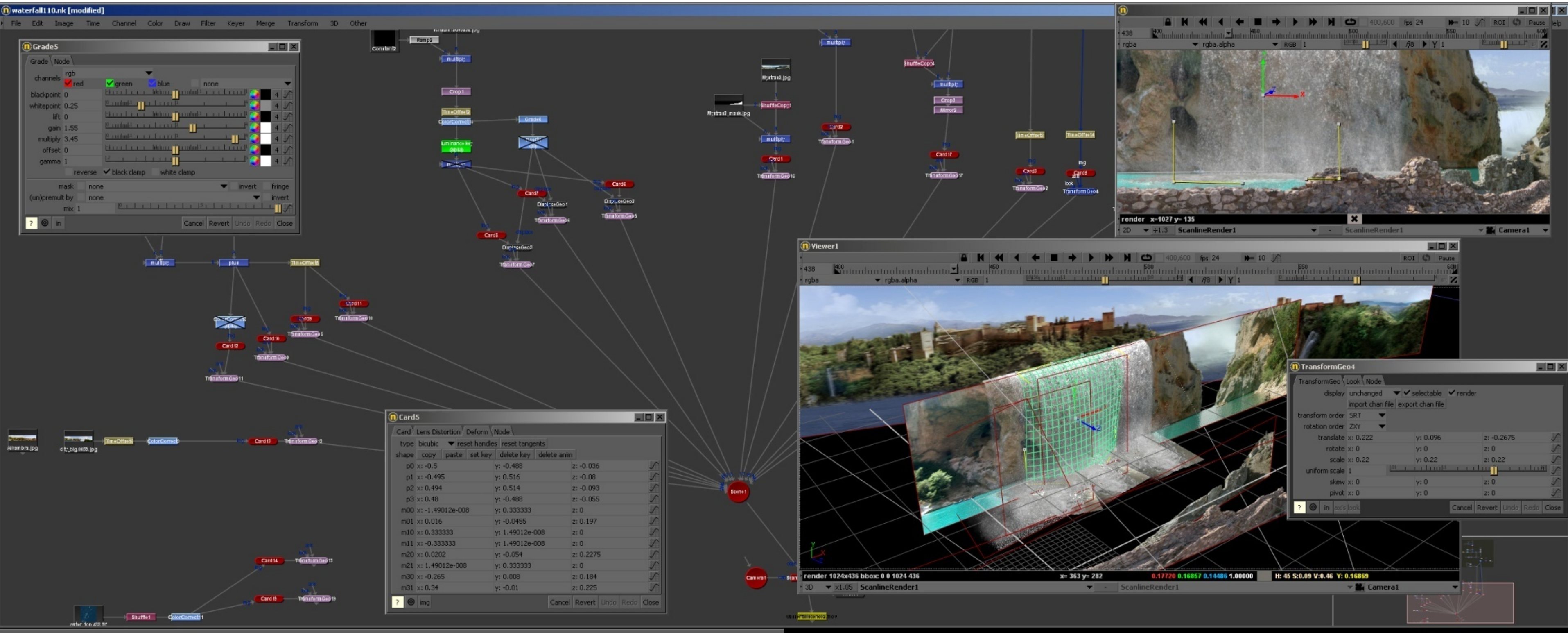
While maintaining the abstractions that keep code clean, reusable and of long-term value?

Case study: Visual Effects

- The Foundry is a London company building visual effects plug-ins for the movie/TV industry (http://www.thefoundry.co.uk/)
- Core competence: image processing algorithms
- Core value: large body of C++ code based on library of image-based primitives

- Opportunity 1:
  - Competitive advantage from exploitation of whatever platform the customer may have - SSE, multicore, vendor libraries, GPUs
- Opportunity 2:
  - Redesign of the Foundry's Image Processing Primitives Library
- Risk:
  - Premature optimisation delays delivery
  - Performance hacking reduces value of core codebase

Visual effects in movie post-production

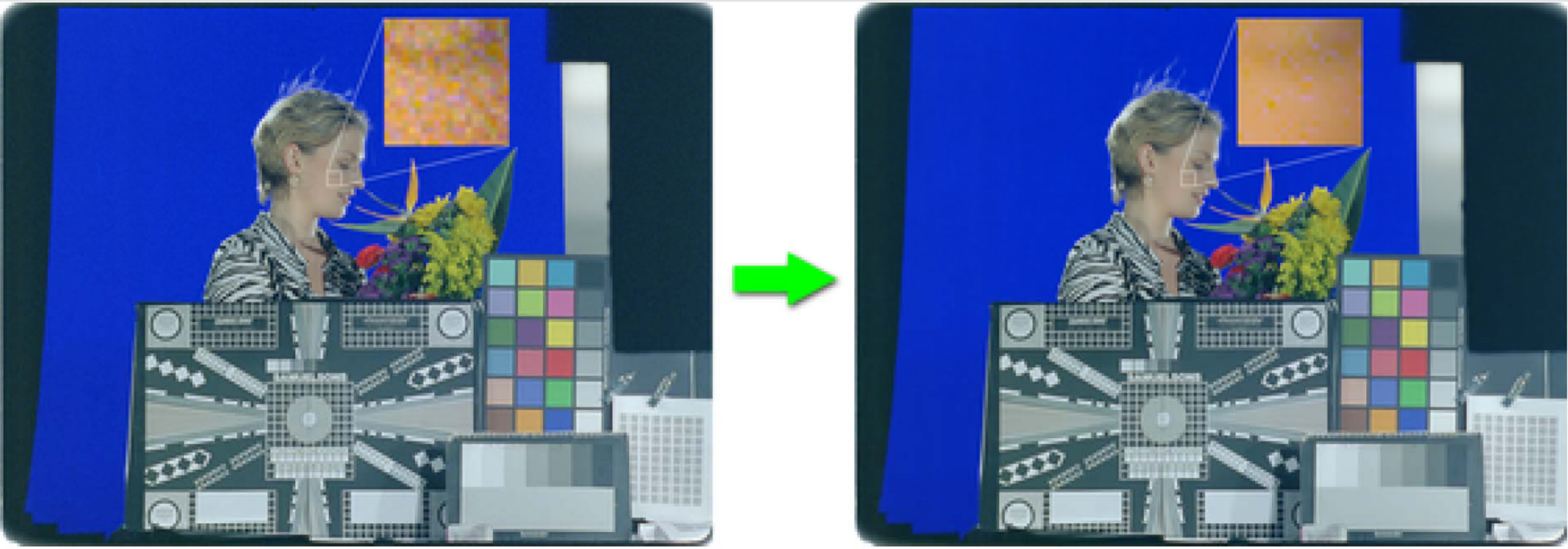


- Nuke compositing tool (http://www.thefoundry.co.uk)

- Visual effects plugins (Foundry and others) appear as nodes in the node graph
- We aim to optimise individual effects for multicore CPUs, GPUs etc
- In the future: tunnel optimisations across node boundaries at runtime.

(c) Heribert Raab, Softmachine. All rights reserved. Images courtesy of The Foundry

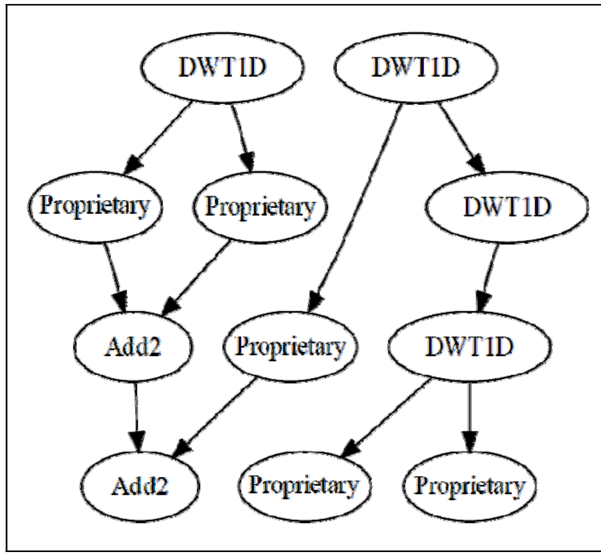
Visual effects: degrain example

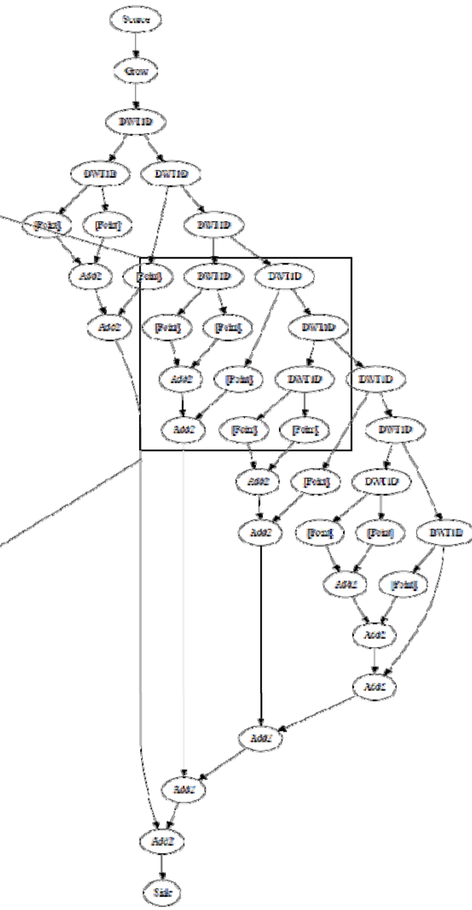


- Image degraining effect – a complete Foundry plug-in
- Random texturing noise introduced by photographic film is removed without compromising the clarity of the picture, either through analysis or by matching against a database of known film grain patterns
- Based on undecimated wavelet transform
- Up to several seconds per frame

Visual effects: degrain example

```
Image DeGrainRecurse(Image input, int level = 0) {
  Image HY,LY,HH,HL,LH,LL,HHP,HLP,LHP,LLP,pSum1,pSum2,out;

  DWT1D hDWT(eHorizontal, 1 << level);
  DWT1D vDWT(eVertical, 1 << level);
  hDWT(input, HY, LY);
  vDWT(HY, HH, LH);
  vDWT(LY, LH, LL);

  Proprietary prop;
  prop(HH, HHP);
  prop(LH, LHP);
  prop(HL, HLP);

  Sum sum;
  sum(HHP, LHP, pSum1);
  sum(HLP, pSum1, pSum2);

  /* Go to the next level of recursion. */
  LLP = (level < 3) ? DeGrainRecurse(LL, level+1) : LL;

  sum(pSum2, LLP, out);
  return out;
}
```



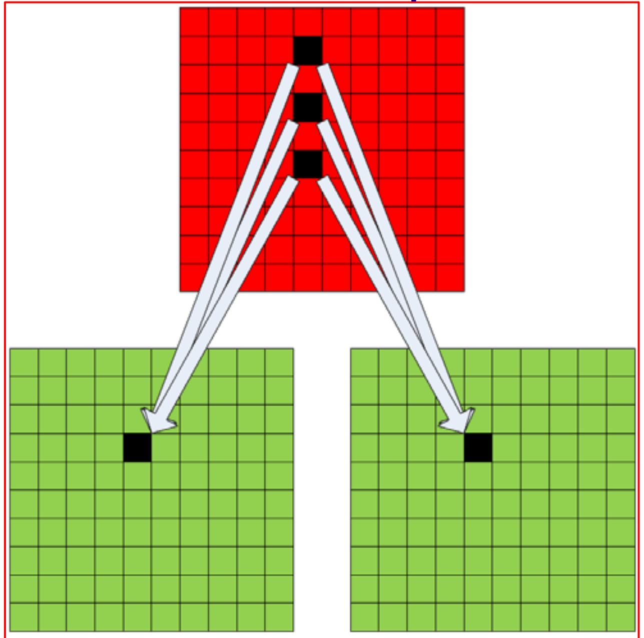


- The recursive wavelet-based degraining visual effect in C++
- Visual primitives are chained together via image temporaries to form a DAG
- DAG construction is captured through delayed evaluation.

Indexed functor

- Functor represents function over an image
- Kernel accesses image via *indexers*
- Indexers carry metadata that characterises kernel's data access pattern

```
class DWT1D : public Functor<DWT1D, eParallel> {
  Indexer<eInput,  eComponent, e1D> Input;
  Indexer<eOutput, eComponent, e0D> HighOutput;
  Indexer<eOutput, eComponent, e0D> LowOutput;
  mFunctorIndexers(Input, HighOutput, LowOutput);

  DWT1D(Axis axis, Radius radius) : Input(axis, radius) {}

  void Kernel() {
    float centre = Input();
    float high = (centre - (Input(-Input.Radius) +
                 Input(Input.Radius)) * 0.5f) * 0.5f;

    HighOutput() = high;
    LowOutput() = centre - high;
  }
};
```

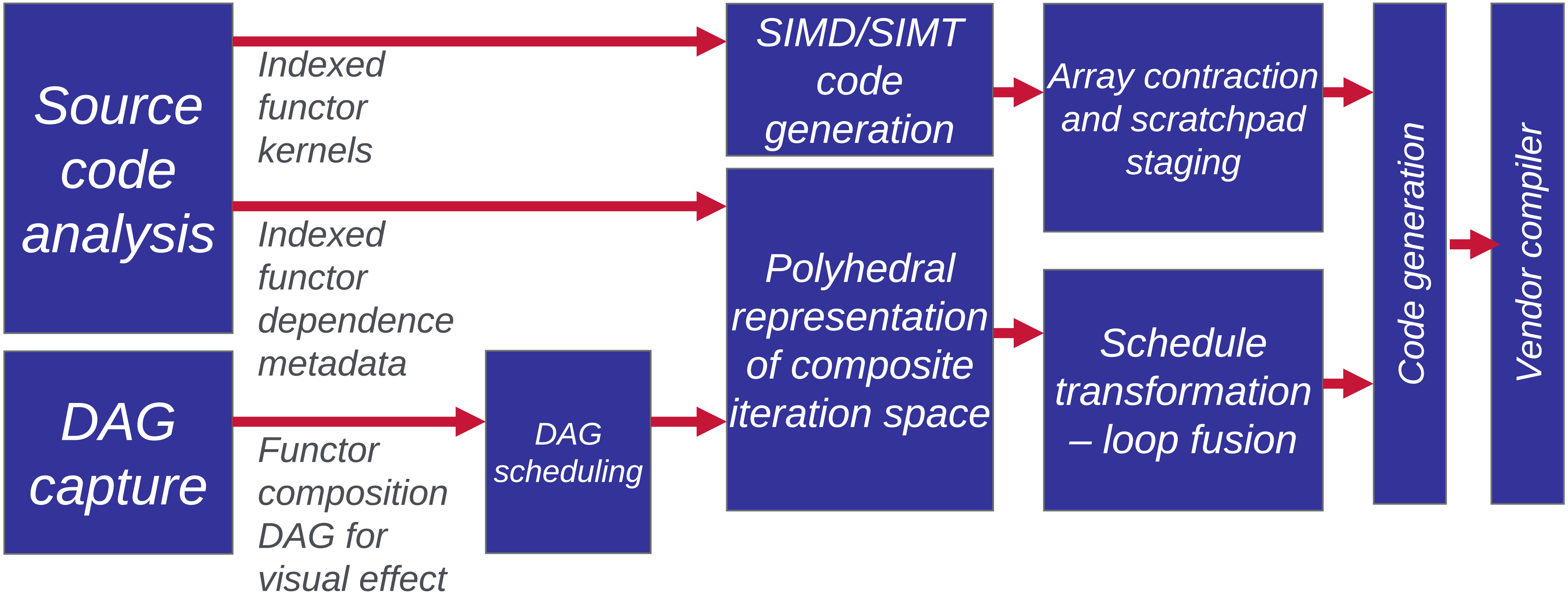


Vertical DWT
(1:2 / Filter Skeleton)

- One-dimensional discrete wavelet transform, as indexed functor
- Compilable with standard C++ compiler
- Operates in either the horizontal or vertical axis
  - Input indexer operates on RGB components separately
  - Input indexer accesses ±radius elements in one (the axis) dimension

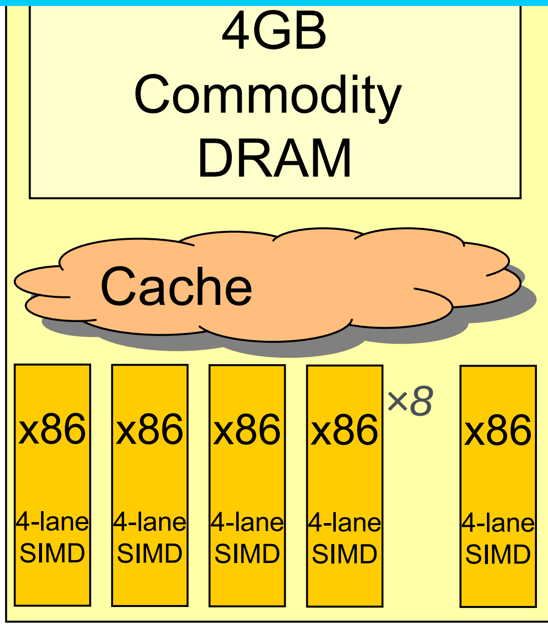
Software architecture

- Use of indexed functors is optimised using a source-to-source compiler (based on ROSE, www.rosecompiler.org)



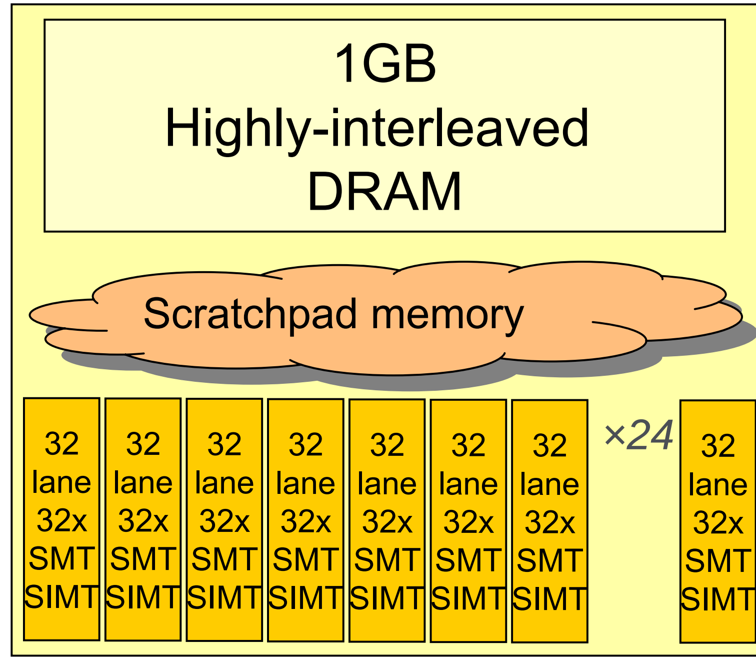
Two generic targets

- Goal:
  - single source code, high-performance code for multiple manycore architectures
- Proof-of-concept: two targets
  - Very different, need very different optimisations



- Lots of cache per thread
- Lower DRAM bandwidth

SIMD Multicore CPU

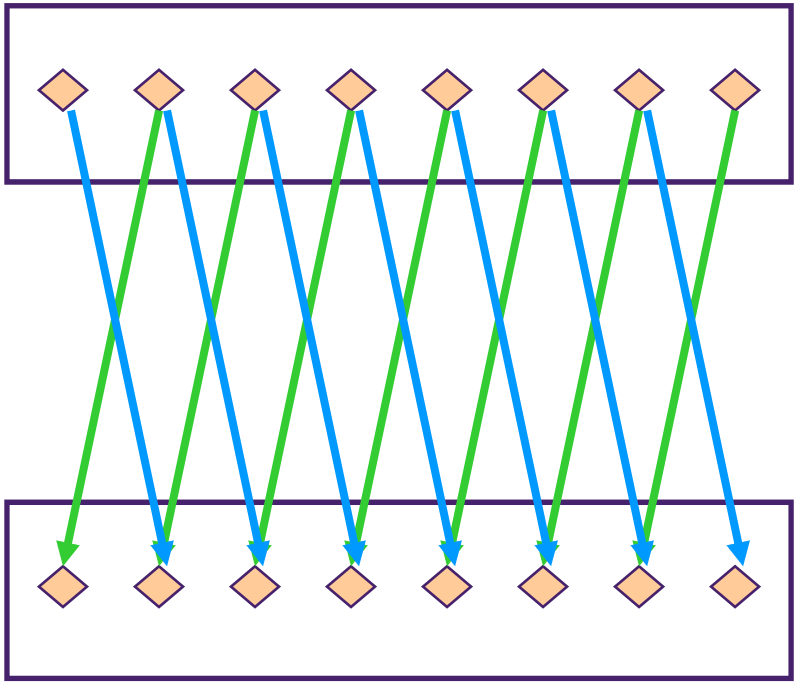


- Very, very little cache per thread
- Very small scratchpad RAM shared by blocks of threads
- Higher DRAM bandwidth

SIMT Manycore GPU

Fusing image filter loops

- Key optimisation is loop fusion
- A little tricky…for example:



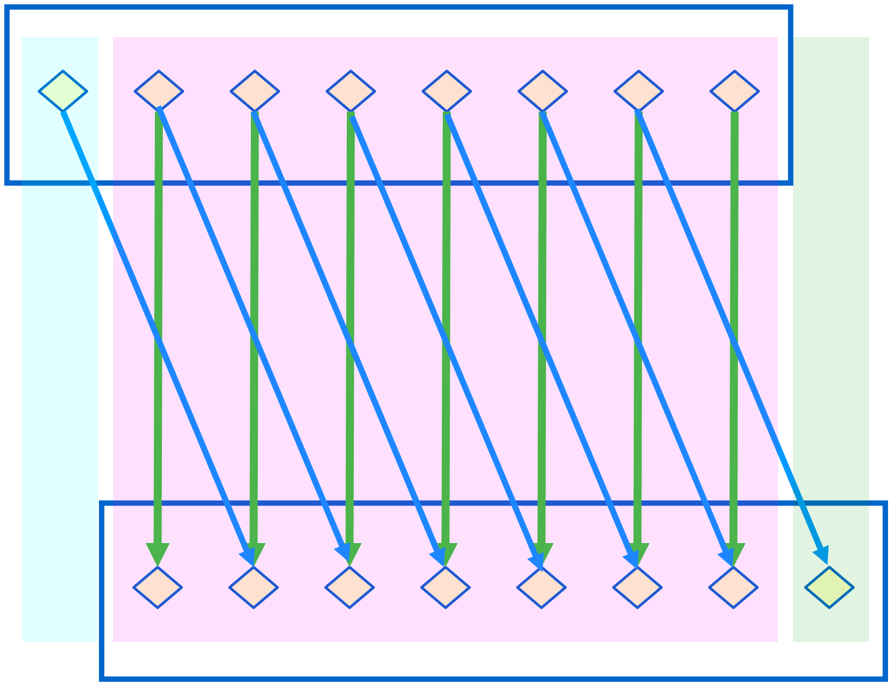
for (i=1; i<N; i++)
  V[i] = (U[i-1] + U[i+1])/2

for (i=1; i<N; i++)
  W[i] = (V[i-1] + V[i+1])/2

- "Stencil" loops are not directly fusable

Fusing image filter loops

- We make them fusable by shifting:

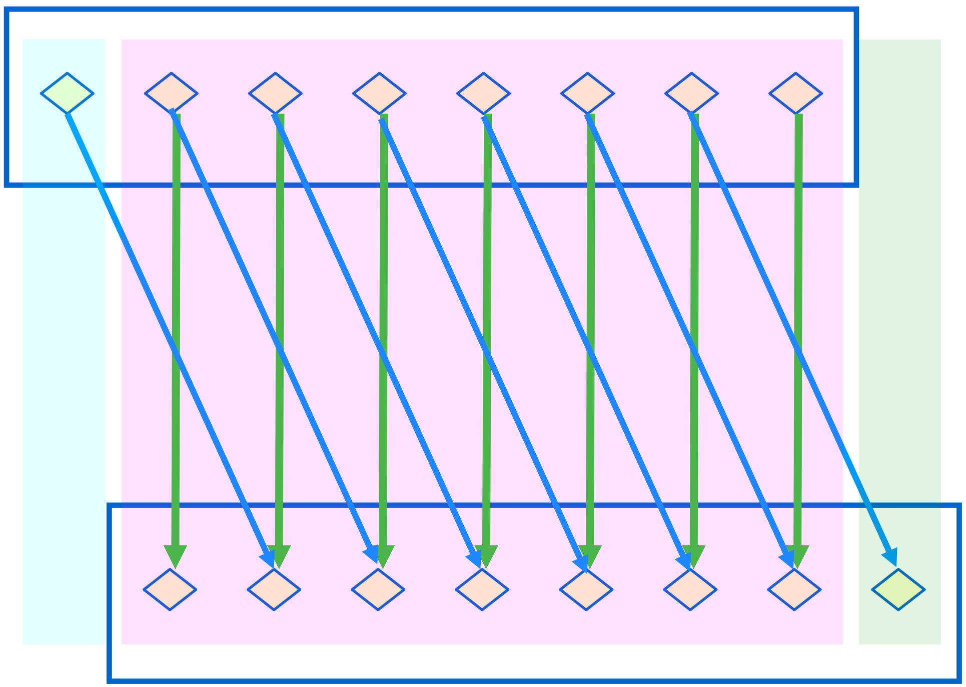


```
V[1] = (U[0] + U[2])/2
for (i=2; i<N; i++) {
  V[i] = (U[i-1] + U[i+1])/2
  W[i-1] = (V[i-2] + V[i])/2
}
W[N-1] = (V[N-2] + V[N])/2
```

- The middle loop is fusable
- We get lots of little edge bits

Array contraction

- The benefit of loop fusion comes from *array contraction* - eliminating intermediate arrays:



```
V[1] = (U[0] + U[2])/2
for (i=2; i<N; i++) {
  V[i%4] = (U[i-1] + U[i+1])/2
  W[i-1] = (V[(i-2)%4] + V[i%4])/2
}
W[N-1] = (V[(N-2)%4] + V[N%4])/2
```

- We need the last *two* Vs
- We need 3 V locations, quicker to round up to four
- Four-element contracted array, used as circular buffer
- Occupies small chunk of cache, avoids trashing rest of cache

The SIMD target...

Code generation for conventional PC with SSE ("SIMD") instructions:

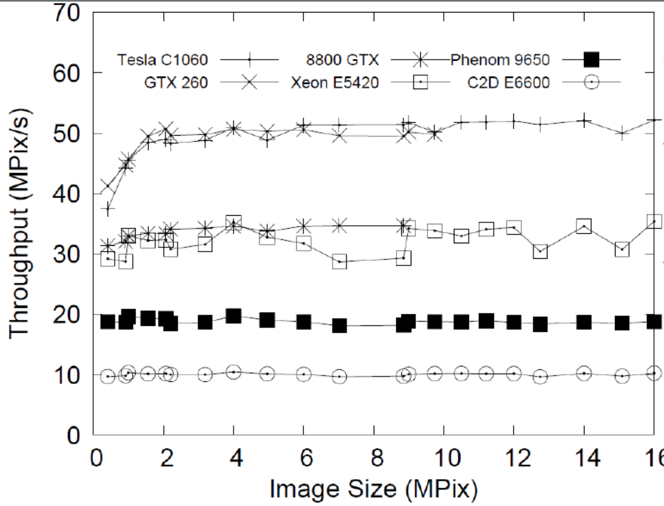
- **Aggressive loop fusion and array contraction**
  - Using the CLooG code generator to generate the loop fragments
- **Vectorisation and Scalar promotion**
  - Correctness guaranteed by dependence metadata
- **If-conversion**
  - Generate code to use masks to track conditionals
- **Memory access realignment**:
  - In SIMD architectures where contiguous, aligned loads/stores are faster, placement of intermediate data is guided by metadata to make this so
- **Contracted load/store rescheduling:**
  - Filters require mis-aligned SIMD loads
  - After contraction, these can straddle the end of the circular buffer – we need them to wrap-around
  - We use a double-buffer trick…

Imperial College London

SIMT – code generation for nVidia's CUDA

- **Constant/shared memory staging**
  - Where data needed by adjacent threads overlaps, we generate code to stage image sub-blocks in scratchpad memory
- **Maximising parallelism**
  - Moving-average filters are common in VFX, and involve a loop-carried dependence
  - We catch this case with a special "eMoving" index type
  - We create enough threads to fill the machine, while efficiently computing a moving average within each thread
- **Coordinated coalesced memory access**
  - We shift a kernel's iteration space, if necessary, to arrange an thread-to-data mapping that satisfies the alignment requirements for high-bandwidth, coalesced access to global memory
  - We introduce transposes to achieve coalescing in horizontal moving-average filters
- **Choosing optimal scheduling parameters**
  - Resource management and scheduling parameters are derived from indexed functor metadata, and used to select optimal mapping of threads onto processors.

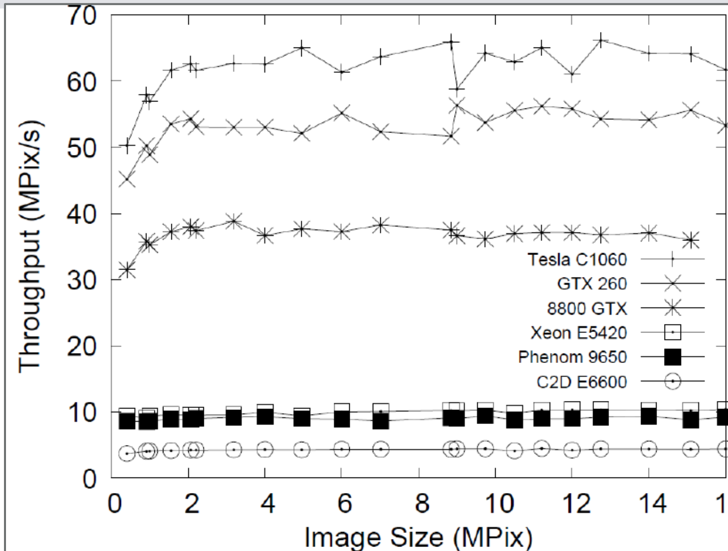
Performance results



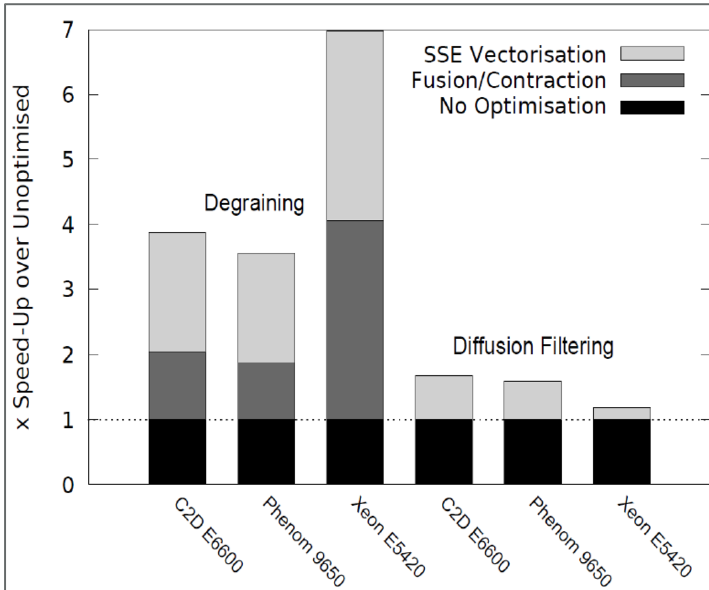
- In this example, CPU *can* beat a GPU
- Because loop fusion eliminates DRAM bottleneck
- Future work: loop fusion for the GPU!

- Tesla C1060 (nVidia)
  - 30-SM, CC 1.3
- GTX 260 (nVidia)
  - 24-SM, CC 1.3
- 8800 GTX (nVidia)
  - 16-SM, CC 1.0
- Phenom 9650 (AMD)
  - 4-core
- Xeon E5420 (Intel)
  - 8 cores, two sockets, two Core2Duos per socket
- C2D E6600 (Intel)
  - 2-Core Core2Duo

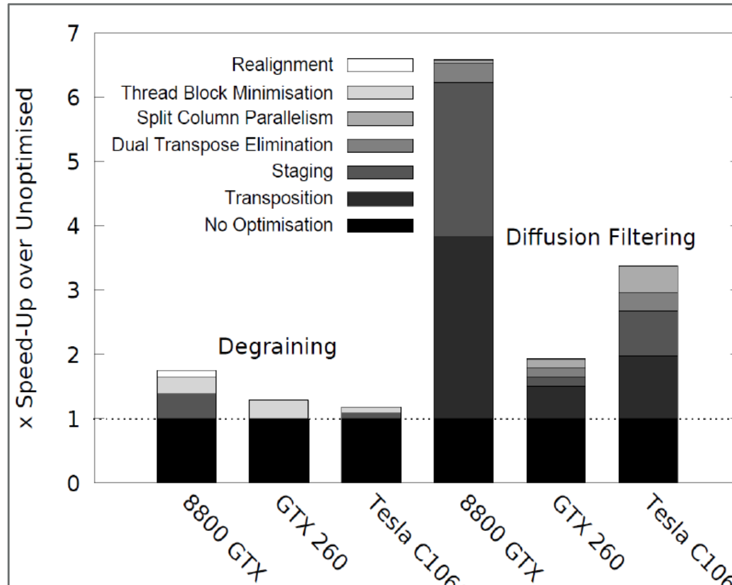
- All systems ran 64-bit Ubuntu Linux 8 with the Intel C/C++ Compiler 11.0, CUDA oolkit 2.1 and 180 series NVIDIA graphics drivers.We used ICC flags "-O3 –xHost no-prec-div -ansi-alias -restrict" and NVCC flag "-O3".
- GPU timings do not include host/device data transfers.
- images were stock photos cropped or repeated to a set of industry-standard rame sizes, powers-of-two and prime numbers



- In this example, GPUs *always* win
- Loop fusion is not possible
- So GPU DRAM bandwidth gives overwhelming advantage
- 8 cores are no better than 4 cores since bandwidth-limited

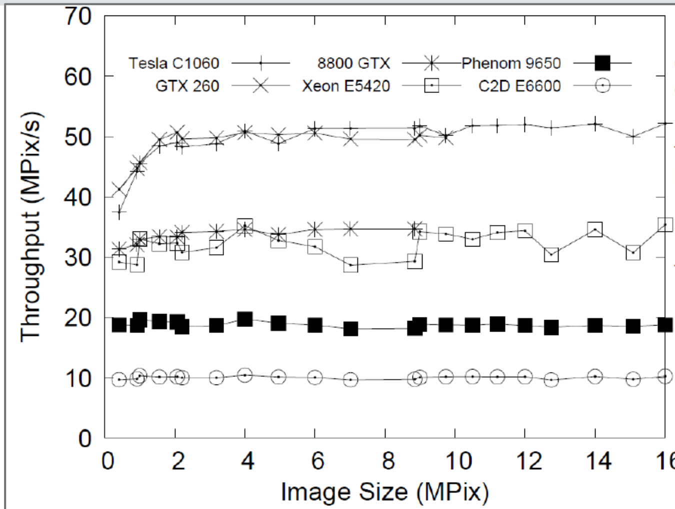


- Without loop fusion, SSE is of limited value – memory is bottleneck
- 8-core Intel Xeon has less DRAM and L2 bandwidth per core, so benefits more from fusion



- Older nVidia hardware was very sensitive to alignment of global memory accesses – not a problem with GTX260 and C1060
- Staging and transposition are crucial for diffusion filtering

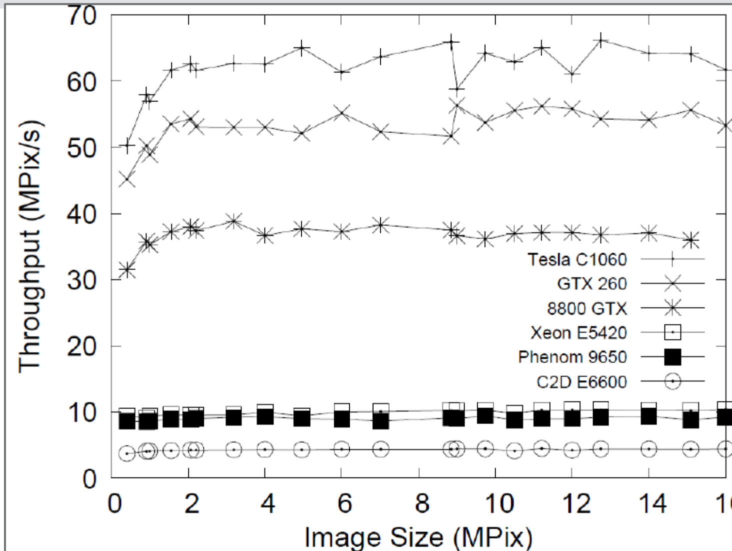
Performance results



- In this example, CPU *can* beat a GPU
- Because loop fusion eliminates DRAM bottleneck
- Future work: loop fusion for the GPU!

- Tesla C1060 (nVidia)
  - 30-SM, CC 1.3
- GTX 260 (nVidia)
  - 24-SM, CC 1.3
- 8800 GTX (nVidia)
  - 16-SM, CC 1.0
- Phenom 9650 (AMD)
  - 4-core
- Xeon E5420 (Intel)
  - 8 cores, two sockets, two
  - 2Duos per socket
- 600 (Intel)
- re Core2Duo

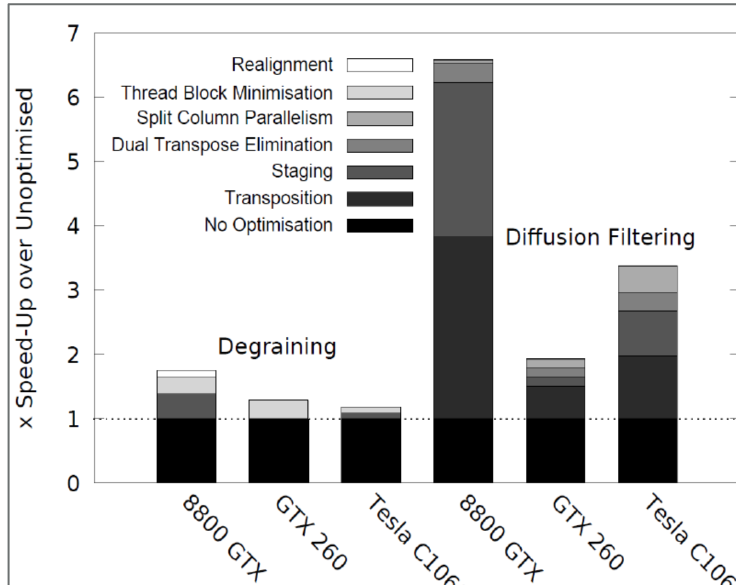
All systems ran 64-bit Ubuntu Linux 8 with the Intel C/C++ Compiler 11.0, CUDA



- In this example, GPUs *always* win
- Loop fusion is not possible
- So GPU DRAM bandwidth gives overwhelming advantage
- 8 cores are no better than 4 cores since bandwidth-limited

- Jay Cornwall's PhD thesis:
- Currently being delivered for use by The Foundry
- By Jay

- ithout loop usion, SSE is f limited alue – emory is ottleneck
- -core Intel eon has less RAM and L2 andwidth per ore, so enefits more om fusion

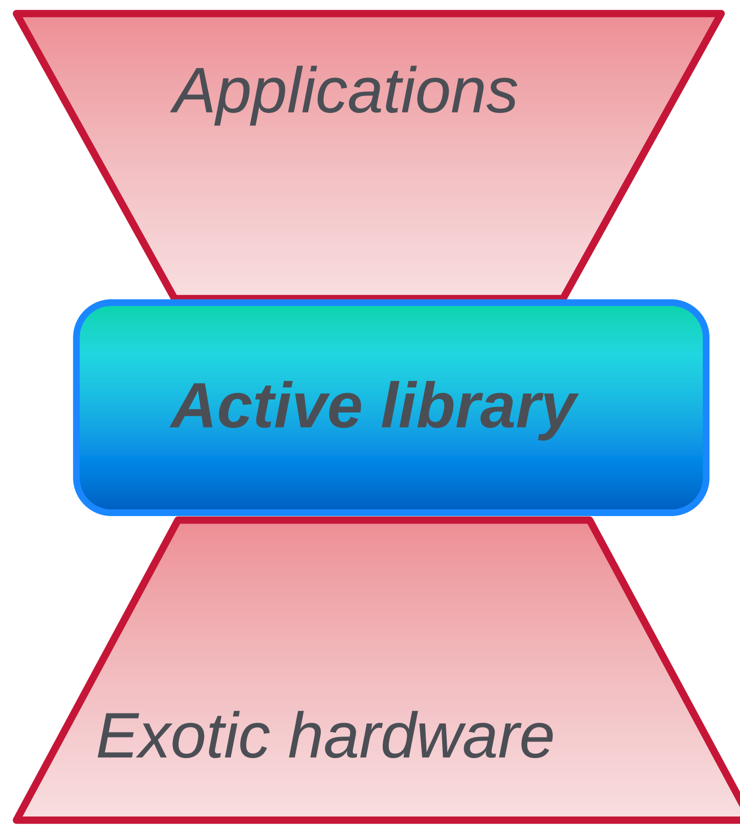


- Older nVidia hardware was very sensitive to alignment of global memory accesses – not a problem with GTX260 and C1060
- Staging and transposition are crucial for diffusion filtering

Imperial College London

Active libraries

- Domain-specific "active" library encapsulates specialist performance expertise
- Each new platform requires new performance tuning effort
- So domain-specialists will be doing the performance tuning
- Our challenge is to support them



Active libraries…

- A selection of active libraries we've developed
  - **DESOBLAS** (1998, Olav Beckmann)
    - Parallel dense matrix/vector library for clusters
    - Automatically selects array alignment to minimise redistribution
  - **DESOLA** (2006, Francis Russell, Mike Gist)
    - Dense matrix/vector linear algebra library for C++
    - Aggressive loop fusion
    - Fusion matches or exceeds hand-tuned ATLAS and IMKL
  - **MayaVi/DSI** (2005, Marc Hull, Karen Osmond, Olav Beckmann et al)
    - Large Python fluid dynamics visualisation tool based on VTK
    - Transparently parallelised for SMP and clusters (+ smart LoD, RoI)
  - **Aggregation of remote method invocations in Java and .Net**
    - (2003, Kwok Yeung, Michael Mellor)
    - Various run-time, static and hybrid implementations
  - **Visual Effects for The Foundry** (LCPC07)
    - Redesign of The Foundry's Fundamental Image Processing Library
    - For multicore: aggressive, skewed, loop fusion, array contraction, vectorisation
    - For GPU: staging, data-placement/alignment, partitioning, transposition
  - **Matrix assembly abstractions for finite element analysis**
    - (ongoing, Francis Russell)

Imperial College London

Specific technical challenges

- Generalise the indexed functors concept
  - **AEcute access-execute descriptors**
- Generic support for pluggable optimisations
  - **DeepWeaver static analysis query language**
- Automate and guide the search for optimal combinations of optimisations
  - **TaskGraph code generation and metaprogramming library**
- Robustness…
  - Static/dynamic checking of dependence metadata
  - Test generation for optimisations
  - We have a specification… can we verify the optimisations statically?
- What happens when you combine different active libraries?

Lee Howes' PhD

Michael Mellor's PhD

Imperial College London

So what of the future?

- Parallelism is everywhere
- Parallelism is essential
- Parallelism is disruptive – it breaks abstractions

- Eckert was right –
  - *Avoid* parallel programming!
  - Isolate ordinary software from parallelism

- Eckert was wrong – we just need the right…
  - Language
  - Machine
  - Discipline
  - Abstractions
  - Education

- *Tools to build really clever parallel implementations*
- *Tools to deliver them*
- *And protect us from what lurks below*

http://www.ralphclevenger.com