Performance Transparency and Performance Portability

Geoff Lowney

Intel Senior Fellow

DHPC++ 2019 Conference
May 13, 2019

Performance transparency: A programmer can write a performance-oriented program for a platform and predict how it will perform.

Performance portability: The program requires minimal tuning to run with acceptable performance on a new target.

Outline

Programmer's performance model

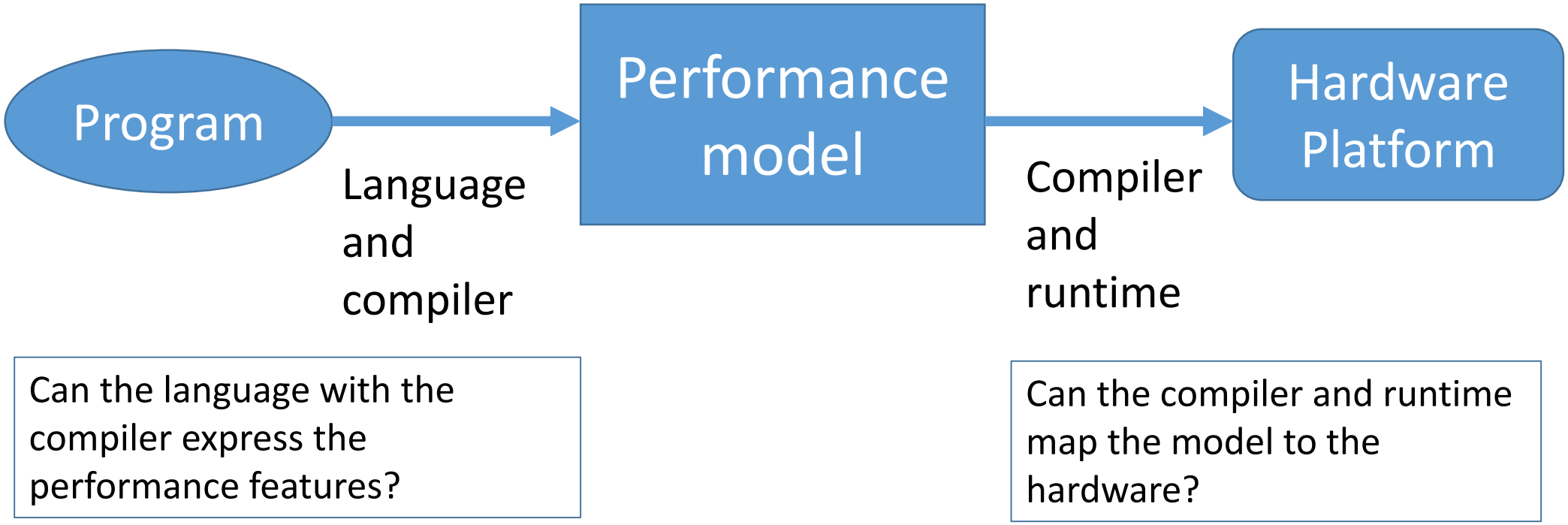
Vector processors and auto-vectorization

GPUs and fine-grained SPMD (Single program, multiple data)

Multi-core CPUs and work-stealing

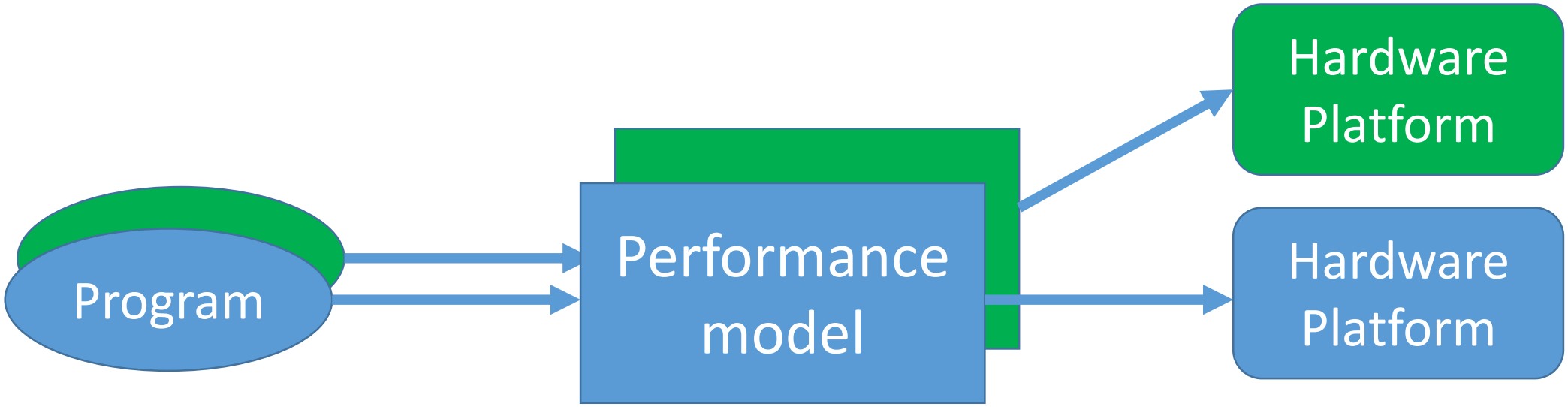
FPGAs and systolic algorithms

Programmer's performance model



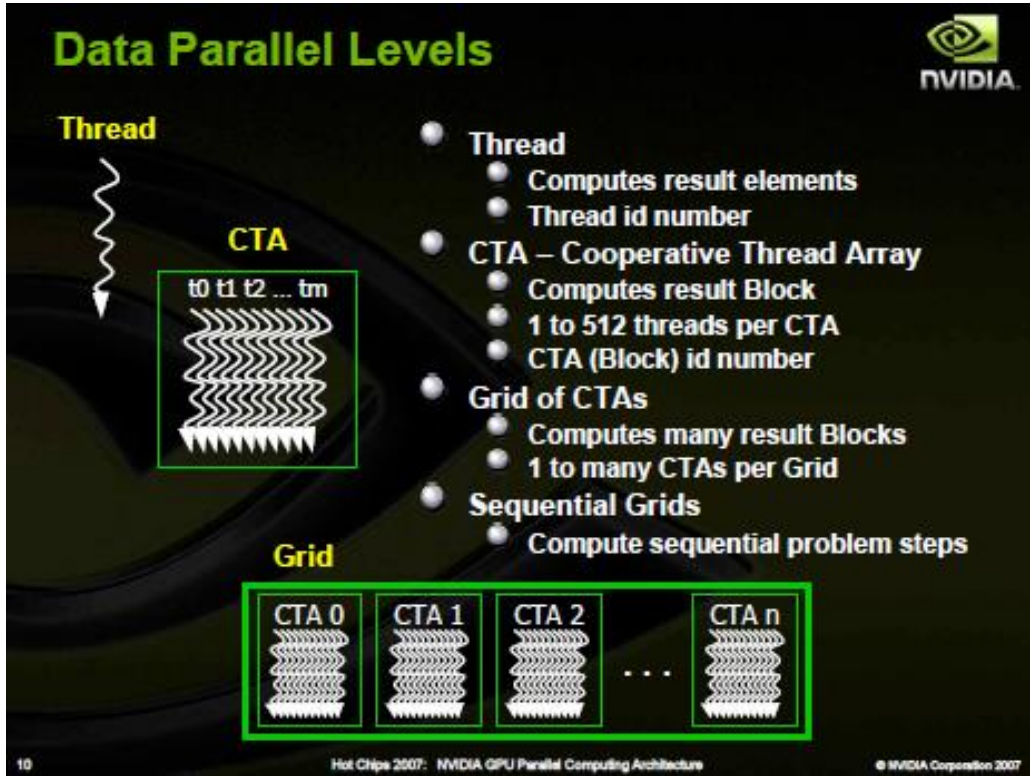
Programmer reasons about the performance model to achieve performance transparency

Performance portability

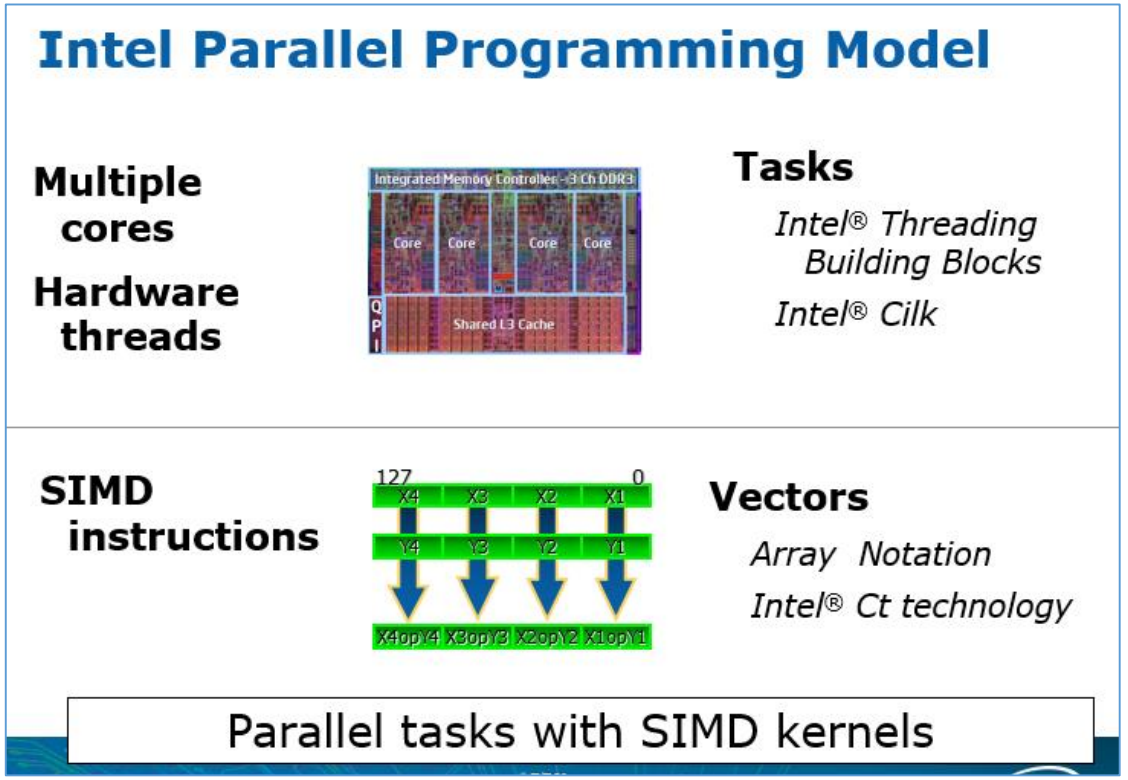


How many changes required to run well on a new platform?

Look back 12 years

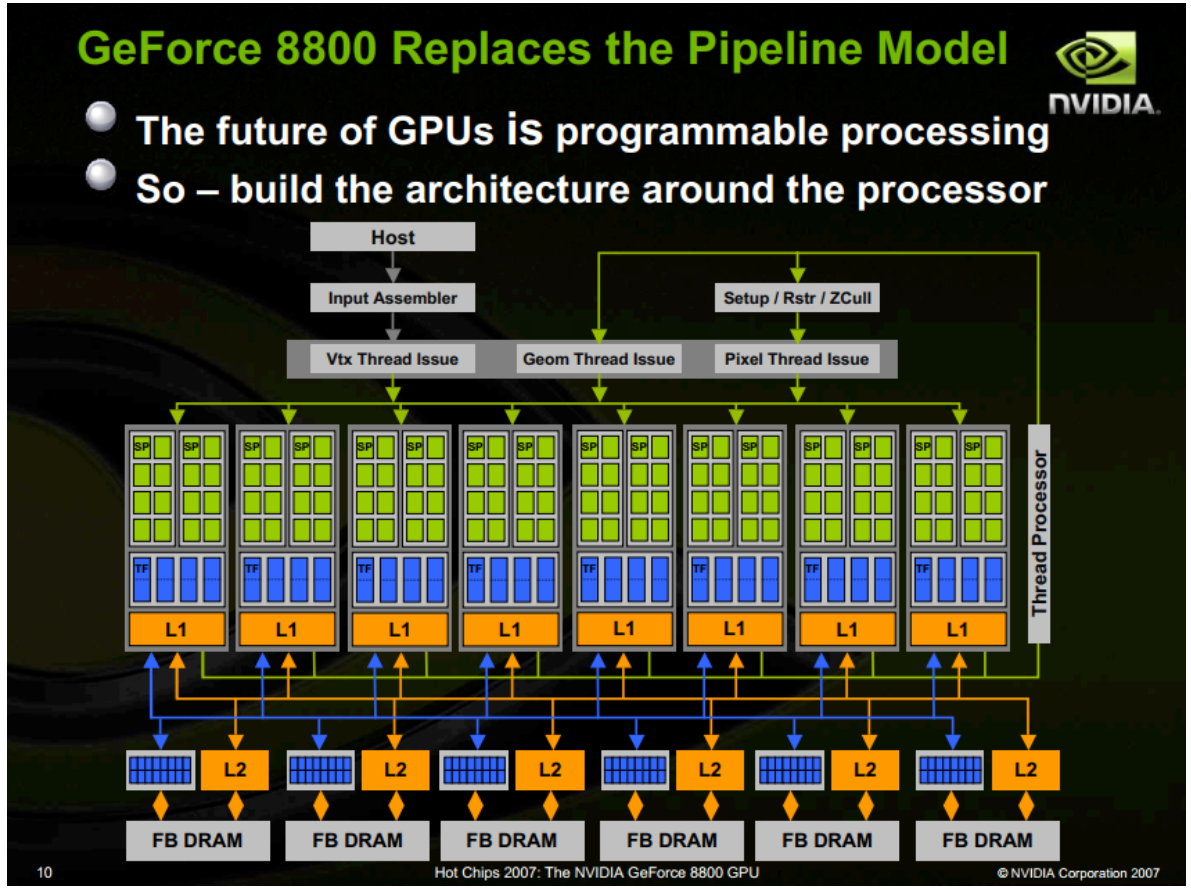


Hot Chips 2007

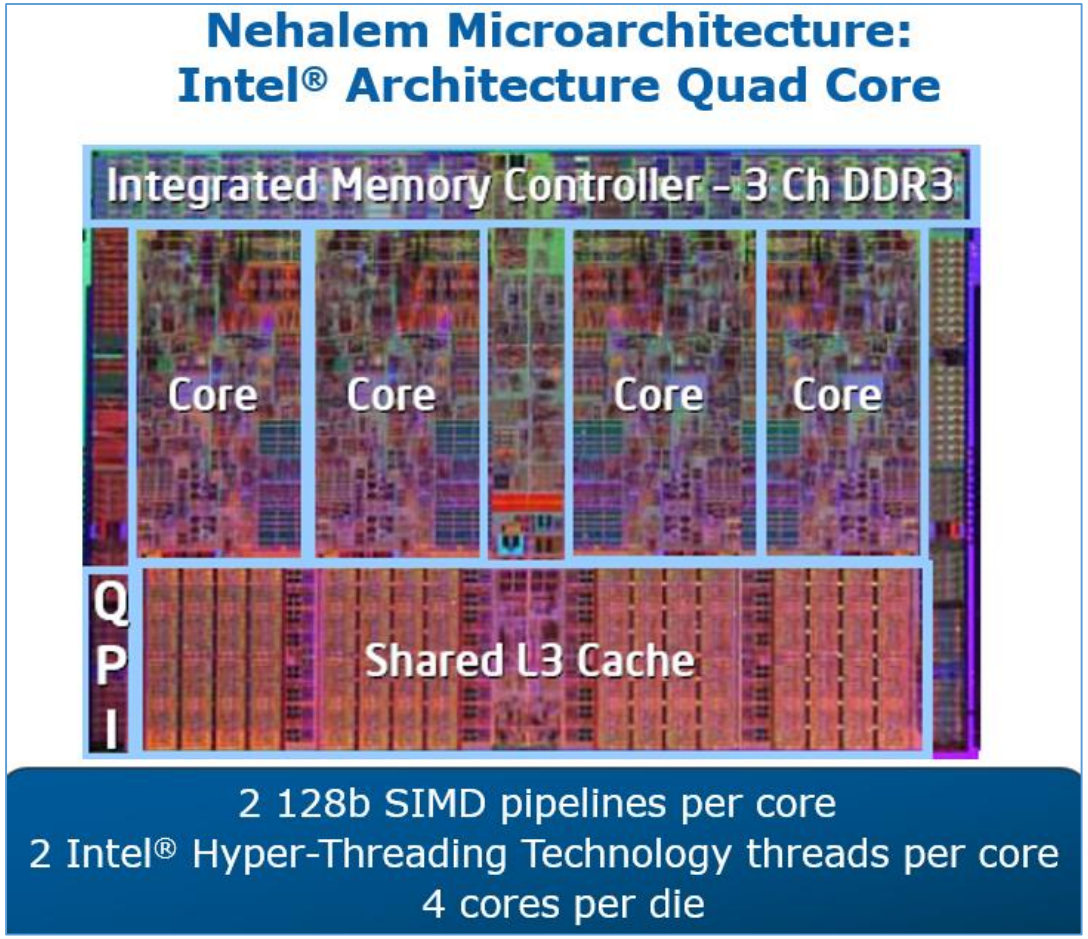


Learn from GP-GPU. Fine-grained SPMD is the better model for data parallel programming

Two multi-core, multi-threaded, SIMD architectures



Hot Chips 2007



Remember the 100x GPU vs CPU Myth

In the summer of 2007, a visiting student, Su Xiaoke, worked with me to investigate the performance of NVIDIA GPUs on a LIBOR market model Monte Carlo application. **Using an NVIDIA 8800 GTX graphics card with 128 cores, we achieved a speedup of over 100 relative to a single Xeon core.**

Professor Mike Giles

http://people.maths.ox.ac.uk/~gilesm/cuda_old.html

LIBOR loop nest

```
for (path=0; path<npath; path++) {
   ...
  for(n=0; n<Nmat; n++) {
    ...
    for (i=n+1; i<N; i++) {
      lam  = lambda[i-n-1];
      con1 = delta*lam;
      v    += (con1*L[i])/(1.0+delta*L[i]);
      vrat = exp(con1*v - lam*(sqez-0.5*con1));
      L[i] = L[i]*vrat;
    }
    ...
  }
}
```

Outer loop parallel

Inner loop sequential

Simple mapping to CUDA

Replace outer loop with kernel invocation from the host

```
Pathcalc_Portfolio_KernelGPU2<<<dimGrid, dimBlock>>>(d_v);


_global__ void Pathcalc_Portfolio_KernelGPU2(float *d_v) {
   ...
  for(n=0; n<Nmat; n++) {
    ...
    for (i=n+1; i<N; i++) {
      lam  = lambda[i-n-1];
      con1 = delta*lam;
      v   += (con1*L[i])/(1.0+delta*L[i]);
      vrat = exp(con1*v + lam*(sqez-0.5*con1));
      L[i] = L[i]*vrat;
    }
    ...
  }
}
```

Kernel essentially unchanged

Similar solutions today in OpenMP and SYCL

```
#pragma omp parallel for simd
for (path=0; path<npath; path++) {
   ...
  for(n=0; n<Nmat; n++) {
    ...
    for (i=n+1; i<N; i++) {
      lam  = lambda[i-n-1];
      con1 = delta*lam;
      v   += (con1*L[i])/(1.0+delta*L[i]);
      vrat = exp(con1*v + lam*(sqez-0.5*con1));
      L[i] = L[i]*vrat;
    }
    ...
  }
}
```

```
Q.parallel_for<class m>(
      range<1>(npath),
      [=](id<1> path) {
  ...
  for(n=0; n<Nmat; n++) {
    ...
    for (i=n+1; i<N; i++) {
      lam  = lambda[i-n-1];
      con1 = delta*lam;
      v   += (con1*L[i])/(1.0+delta*L[i]);
      vrat = exp(con1*v + lam*(sqez-0.5*con1));
      L[i] = L[i]*vrat;
    }
    ...
  }
}
```

Performance portability across platforms with SPMD

Outline

Programmer's performance model

Vector processors and auto-vectorization

GPUs and fine-grained SPMD (Single program, multiple data)

Multi-core CPUs and work-stealing

FPGAs and systolic algorithms

Cray vector machines

Cray 1 1976
Cray XMP 1982

Instruction set with vectors

A vector register holds 64 64b floating point numbers

Pipelined, not SIMD

Gaussian elimination

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & | & b_1 \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} & | & b_2 \\ 0 & 0 & a_{33} & \cdots & a_{3n} & | & b_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & | & \vdots \\ 0 & 0 & a_{n3} & \cdots & a_{mn} & | & b_n \end{bmatrix}$$

```
      DO 10 I = 1,N
10    Y(I) = Y(I) + A*X(I)
```

```
VLD  v1, X(I)    ; load 64 elements
VLD  v2, Y(I)    ; load 64 elements
VMUL v1,r1,v1    ; A*X(I:I+63)
VADD v3, v2, v1  ; Y(I:I+63) + V1
VST  V3, Y(I)    ; store 64 elements
ADD  I,I,64      ; increment
CMP
Br
```

Simple performance model

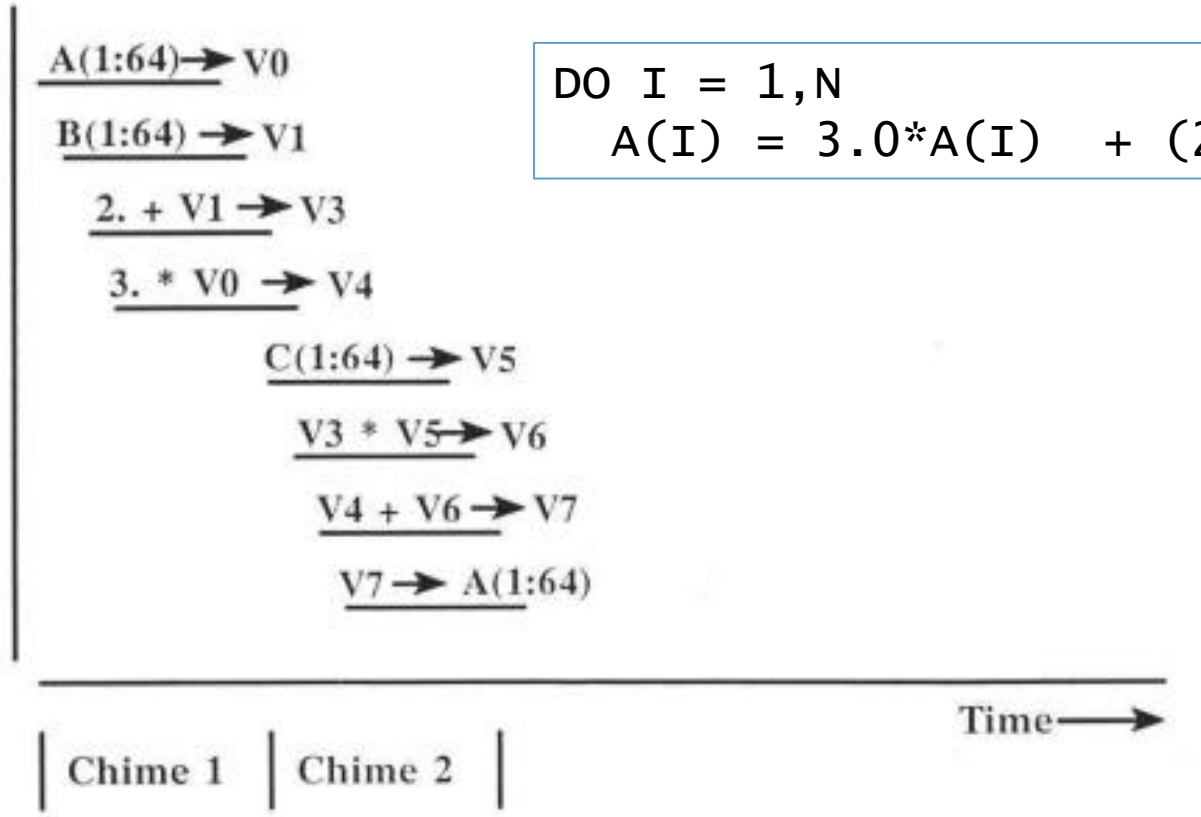


FIGURE 2.10.
Cray X-MP Chime Diagram

```
DO I = 1,N
  A(I) = 3.0*A(I)  + (2.0+B(I)) * C(I)
```

Levesque, A Guidebook to Fortran on Supercomputers

Simple performance model

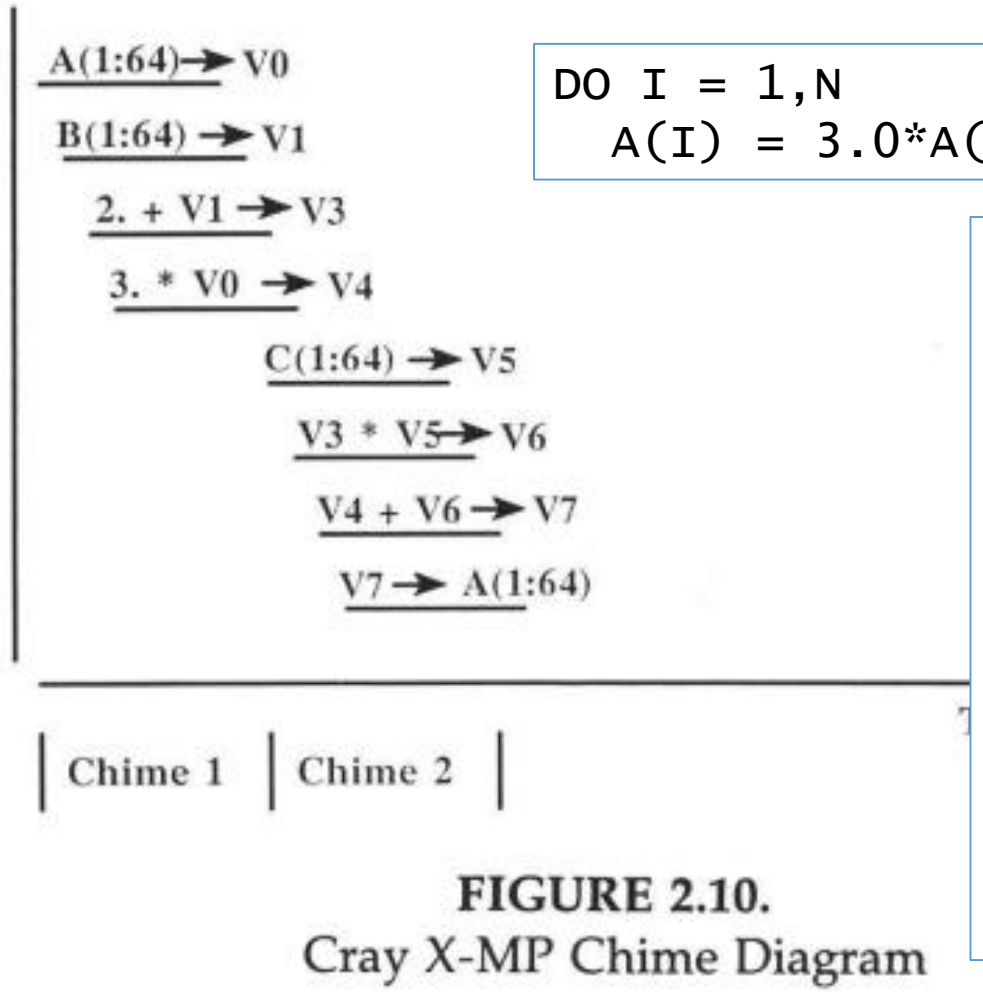


FIGURE 2.10.
Cray X-MP Chime Diagram

```
DO I = 1,N
  A(I) = 3.0*A(I)  + (2.0+B(I)) * C(I)
```

Programmers model:

Inner-loop auto-vectorization, performance meaured in chimes.

Programmers learn to write loops that can be vectorized by the compiler.

Levesque, A Guidebook to Fortran on Supercomputers

Similar to RISC pipeline

```
LD A(1)->r1
LD A(2)→r2
   LD B(1)->r3
   LD B(2)->r4
      FADD 2.0, r1->r5
      FADD 2.0, r2->r7
         FMUL 3.0,r3->r8
         FMUL 3.0,r4->r9
            LD C(1)->R10
            LD C(2)->R11
```

Dual issue RISC pipeline

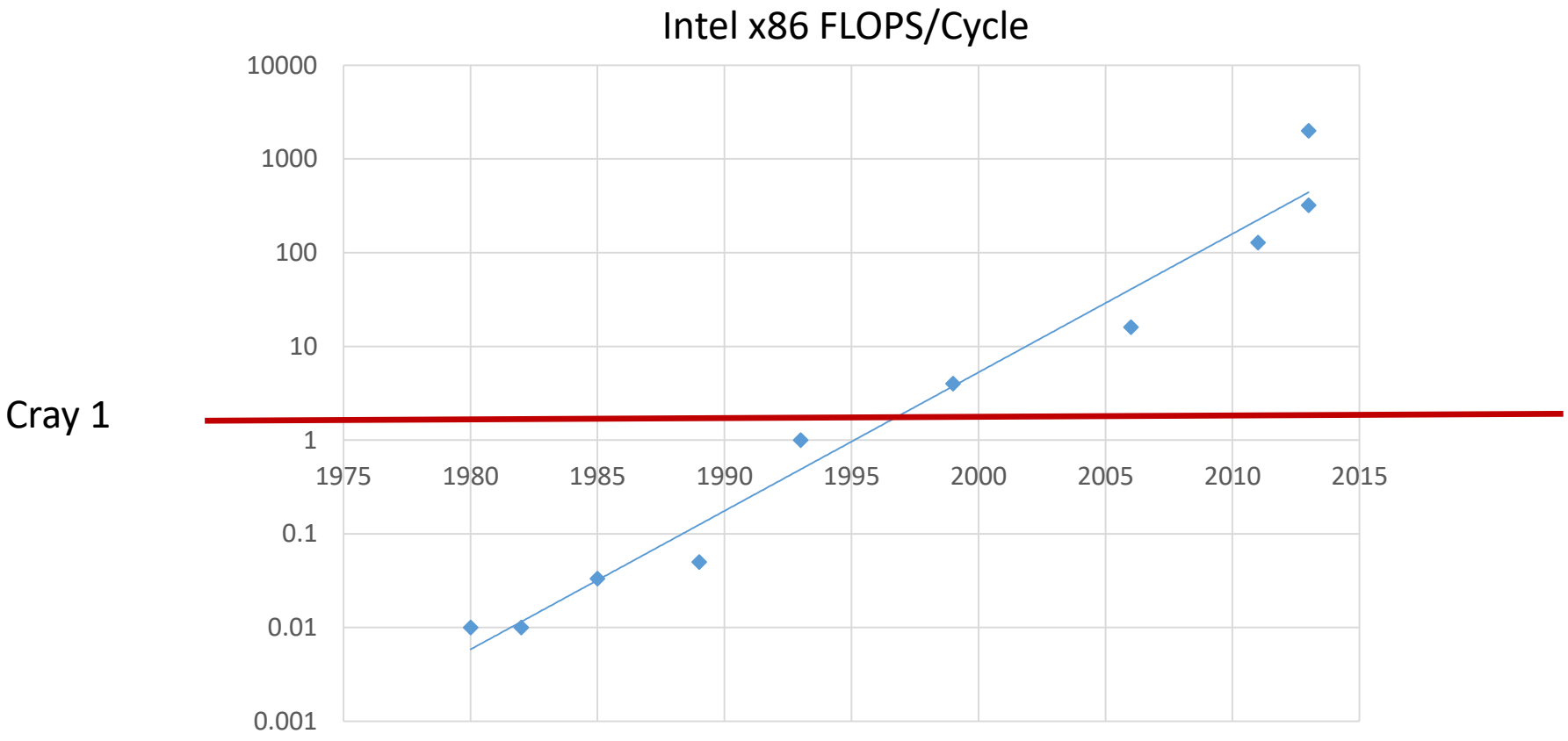
```
DO I = 1,N
  A(I) = 3.0*A(I)  + (2.0+B(I)) * C(I)
```

auto-vectorization becomes software pipelining

Programmers model is essentially the same.

Performance portability with compiler optimization of inner loops

Inner loops are not enough for SIMD



```
C
C*************************************************************************
C***  KERNEL 1       HYDRO FRAGMENT
C*************************************************************************
C
cdir$ ivdep
 1001    DO 1 k = 1,n
    1       X(k)= Q + Y(k) * (R * ZX(k+10) + T * ZX(k+11))
C
```

Livermore loop #1

http://www.netlib.org/benchmark/livermore

1000 line kernel, not shown

A more modernrnel

Outline

Programmer's performance model

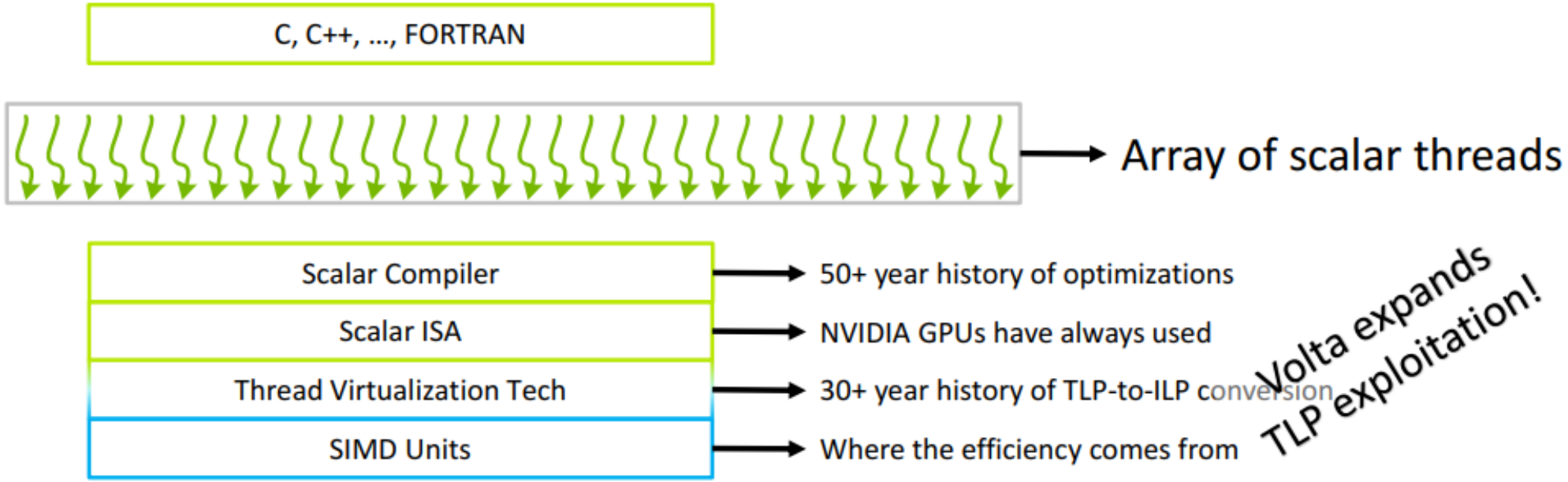
Vector processors and auto-vectorization

GPUs and fine-grained SPMD (Single program, multiple data)

Multi-core CPUs and work-stealing

FPGAs and systolic algorithms

NVIDIA SIMT GPUS: SCALAR THREADS ON SIMD UNITS



Hot Chips 2017

Matt Pharr: Bring GPU model to CPU

SPMD On A GPU SIMD Unit

```
a = b + c;
if (a < 0)
    ++b;
else
    ++c;
```

~PTX	ALU	ALU	ALU	ALU	...	ALU
fadd	+	+	+	+		+
cmp, jge l_a	T	F	F	T		T
fadd, jmp l_b	++			++		++
l_a:						
fadd		++	++			
l_b:						

(Based on http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf)

SPMD On A CPU SIMD Unit

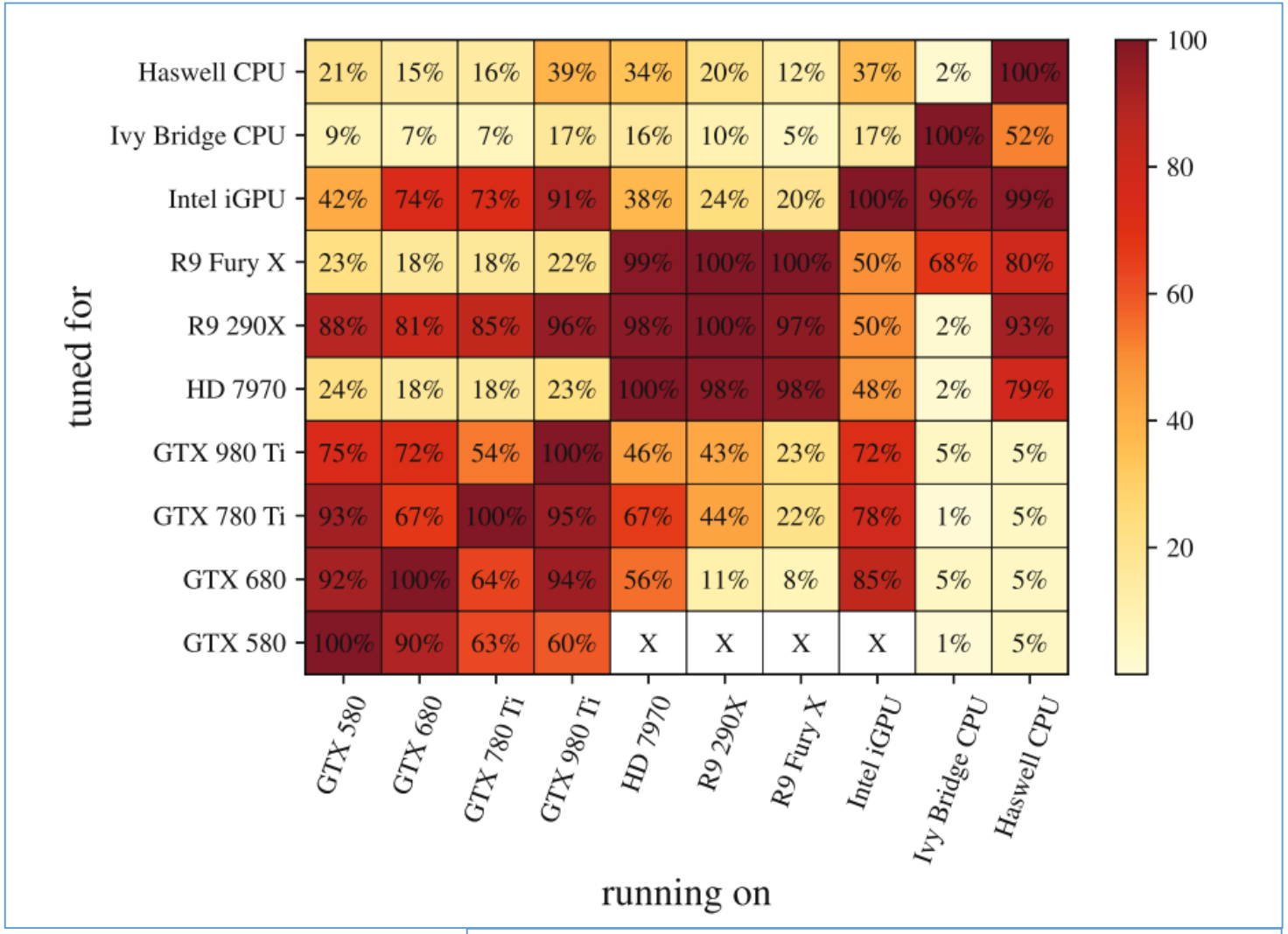
```
a = b + c;
if (a < 0)
    ++b;
else
    ++c;
```

AVX	ALU	ALU	ALU	ALU	...	ALU
vaddps	+	+	+	+		+
vcmpltps	T	F	F	T		T
vaddps, vblendvps	++			++		++
vaddps, vblendvps		++	++			

(Based on http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf)

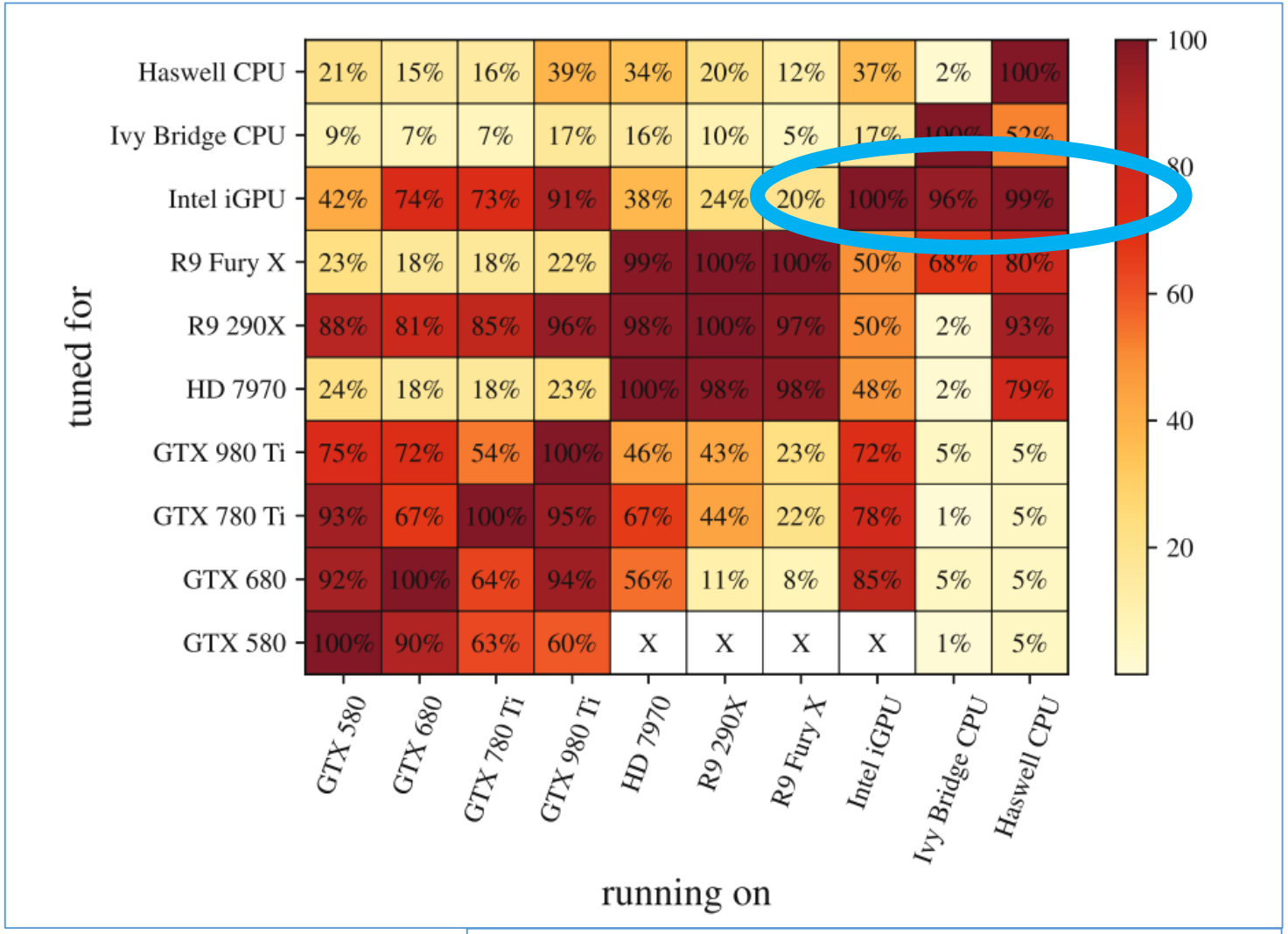
- *Execution* divergence across SIMD lanes reduces SPMD performance
- *Memory access* divergence across SIMD lanes reduces SPMD performance

Performance Portability?



Price, et al. ISC High Performance Workshop, 2017

Performance Portability?



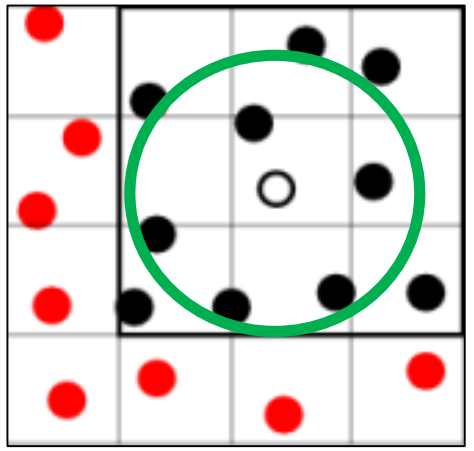
Price, et al. ISC High Performance Workshop, 2017

More challenging example: molecular dynamics

```
For each timestep     // sequential
    For each atom A   // can be parallel
        Compute forces on A from all other atoms
        Move A
        record statistics
```

N**2

Use force cutoff distance to avoid N**2 behavior



CHARMM (Bio) Force-Field from Baseline LAMMPS

```
for (ii = 0; ii < inum; ii++) {
  i = ilist[ii];
  qtmp = q[i]; xtmp = x[i][0]; ytmp = x[i][1]; ztmp = x[i][2];
  itype = type[i];
  jlist = firstneigh[i];
  jnum = numneigh[i];

  for (jj = 0; jj < jnum; jj++) {
    j = jlist[jj];
    factor_lj = special_lj[sbmask(j)];
    factor_coul = special_coul[sbmask(j)];
    j &= NEIGHMASK;

    delx = xtmp - x[j][0]; dely = ytmp - x[j][1]; delz = ztmp - x[j][2];
    rsq = delx*delx + dely*dely + delz*delz;

    if (rsq < cut_bothsq) {
      // MATH
      f_x += delx*fpair; f_y += dely*fpair; f_z += delz*fpair;
      if (newton_pair) {
        f[j][0] -= delx*fpair; f[j][1] -= dely*fpair; f[j][2] -= delz*fpair;
      }
    }
  }
  f[i][0] += f_x; f[i][1] += f_y; f[i][2] += f_z;
  f_x = f_y = f_z = 0.0;
}
```

Loop over local atoms

Loop over neighbors

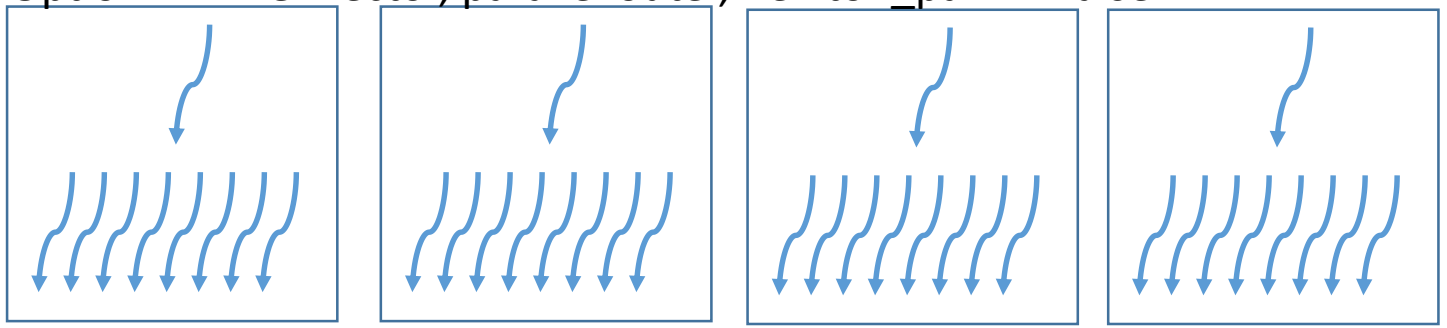
j atom data not usually contiguous in memory

Newton's 3rd: Force calculated once for each pair and updated for both atoms in memory. Introduces a race condition in the "i" loop.

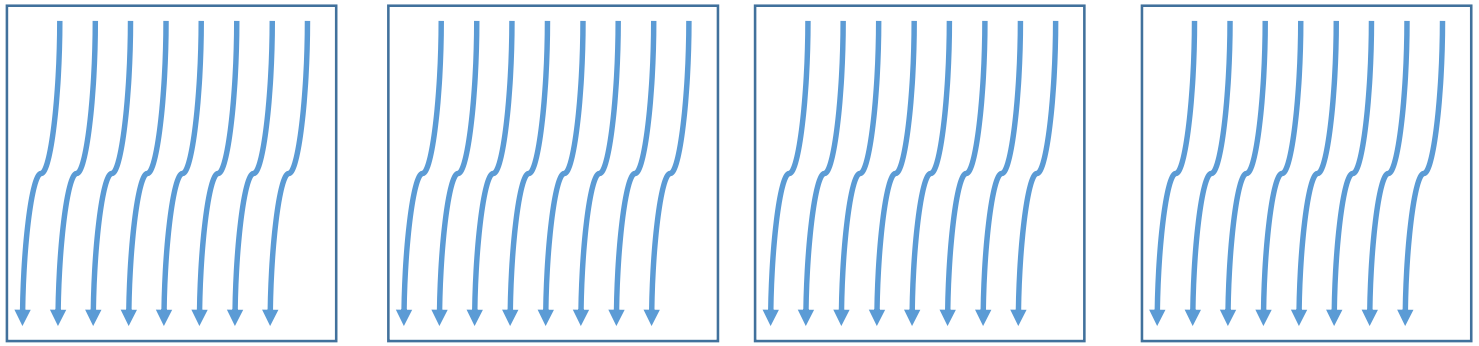
CHARMM (Bio) Force-Field from Baseline LAMMPS

```
for (ii = 0; ii < inum; ii++) {
  i = ilist[ii];
  . . .
  for (jj = 0; jj < jnum; jj++) {
    j = jlist[jj];
    . . .
    if (rsq < cut_bothsq) {
        . . .
        if (newton_pair) {
           f[j][0] -= . . .
        }
    }
  }
  f[i][0] += . . .
}
```

Option 1: **Inner** vector, parallel outer, newton_pair == false



Option 2: **Outer** vector, newton_pair == false



Option 3: **Privatize,** inner vector, parallel outer, newton_pair == true

Option 4: **Atomic**, inner vector, parallel outer, newton_pair == true

Outline

Programmer's performance model

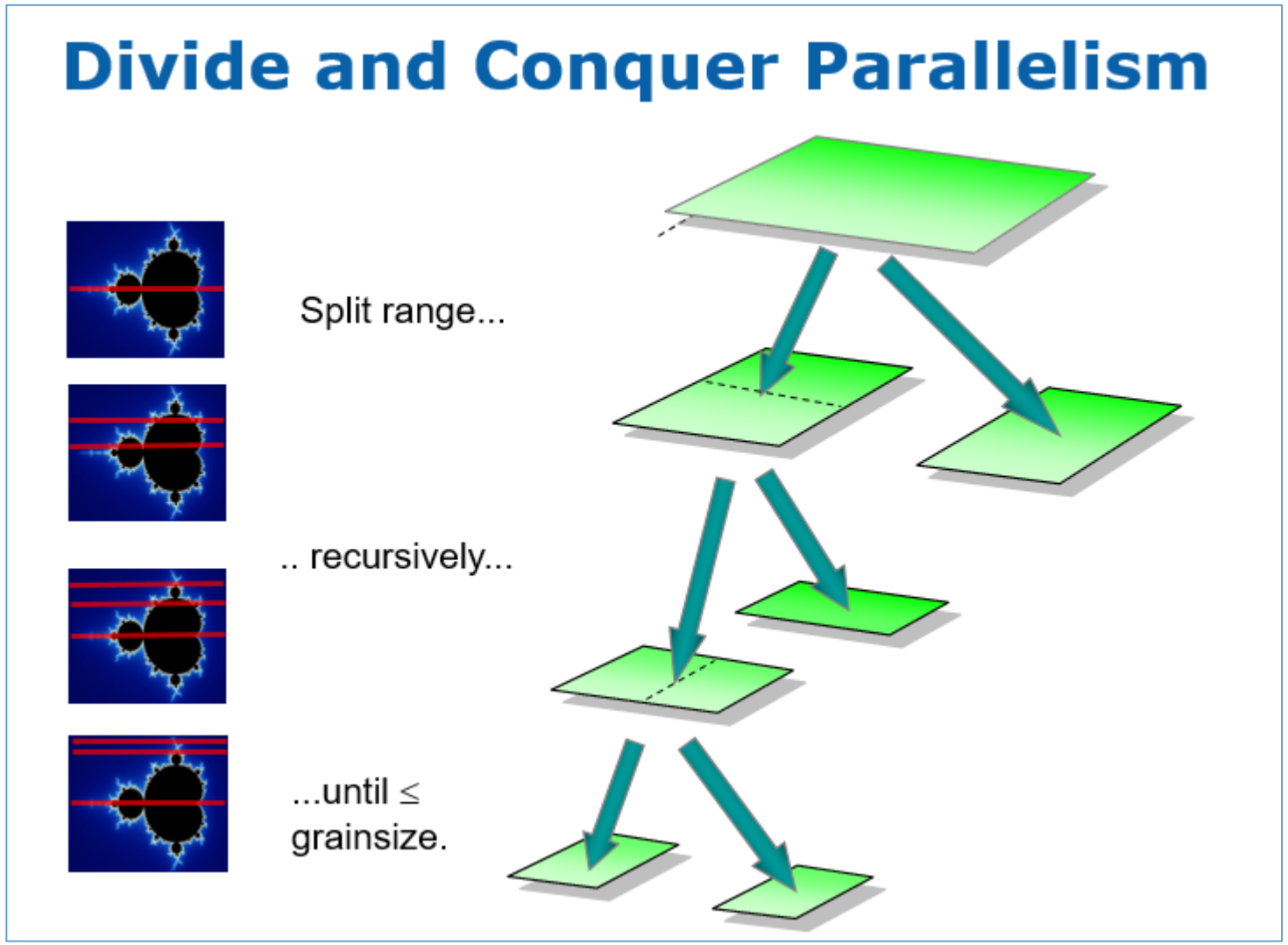
Vector processors and auto-vectorization

GPUs and fine-grained SPMD (Single program, multiple data)

Multi-core CPUs and work-stealing

FPGAs and systolic algorithms

Intel Threaded Builing Blocks (TBB)



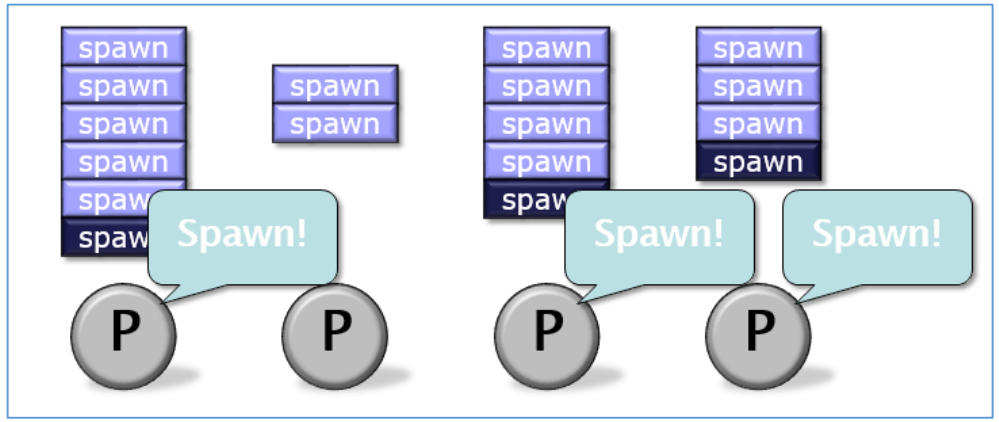
Tasks, not threads

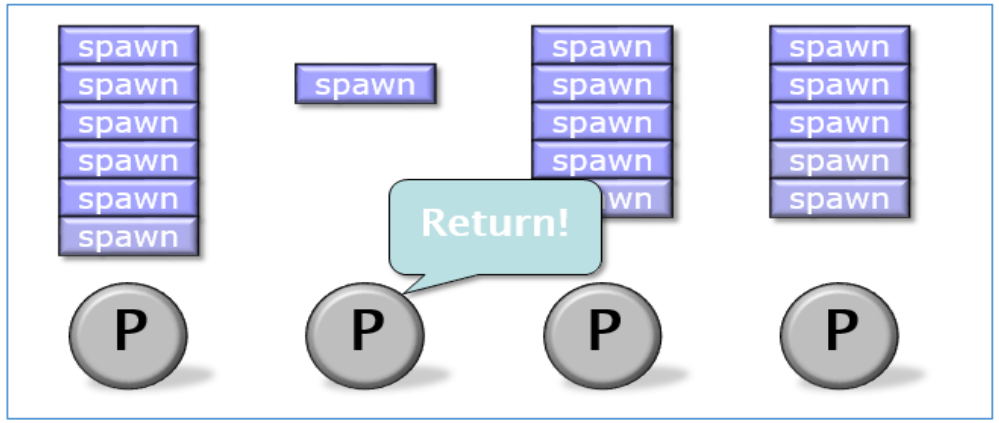
Over-decompose (parallel slack)

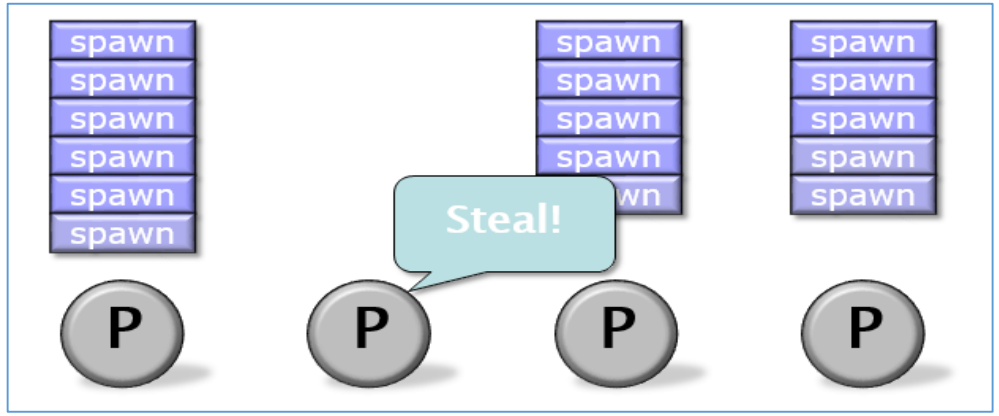
Work-stealing for load-balancing

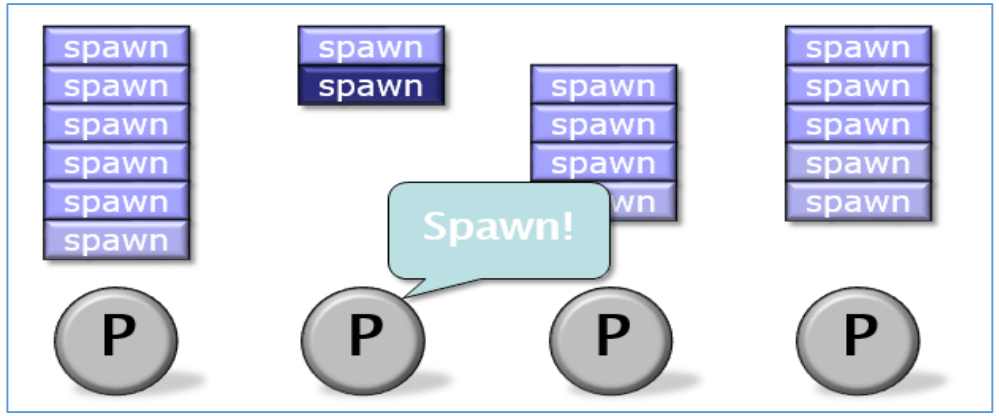
Adapts to the available resources and to the workload

Work-stealing scheduler





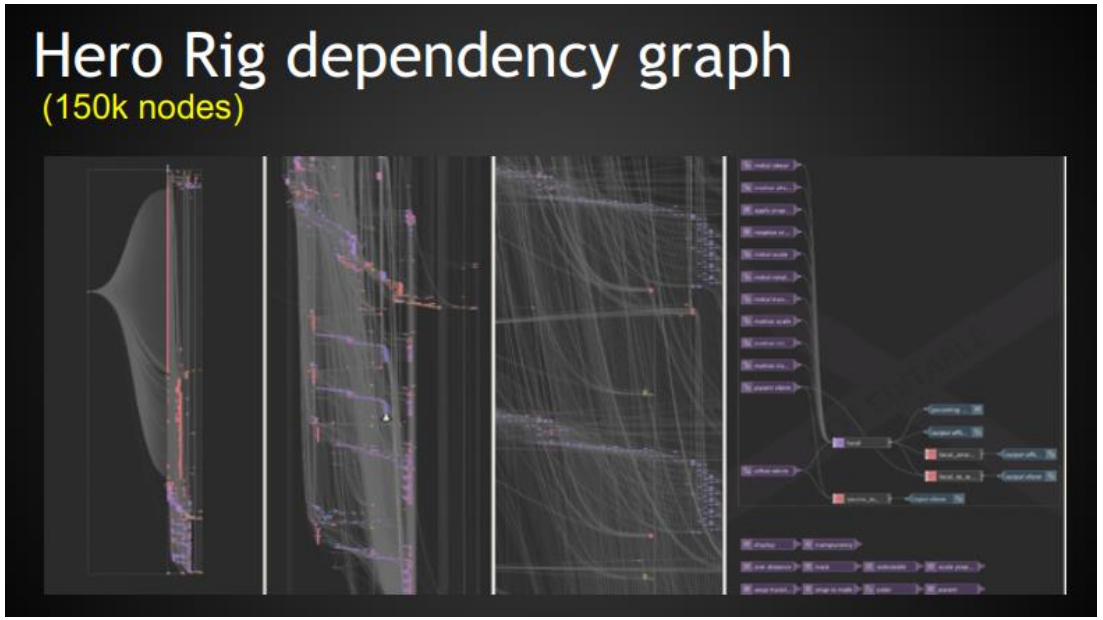




Adapted from copyrighted originals developed by Charles E. Leiserson of MIT.

DreamWorks Animation and Intel

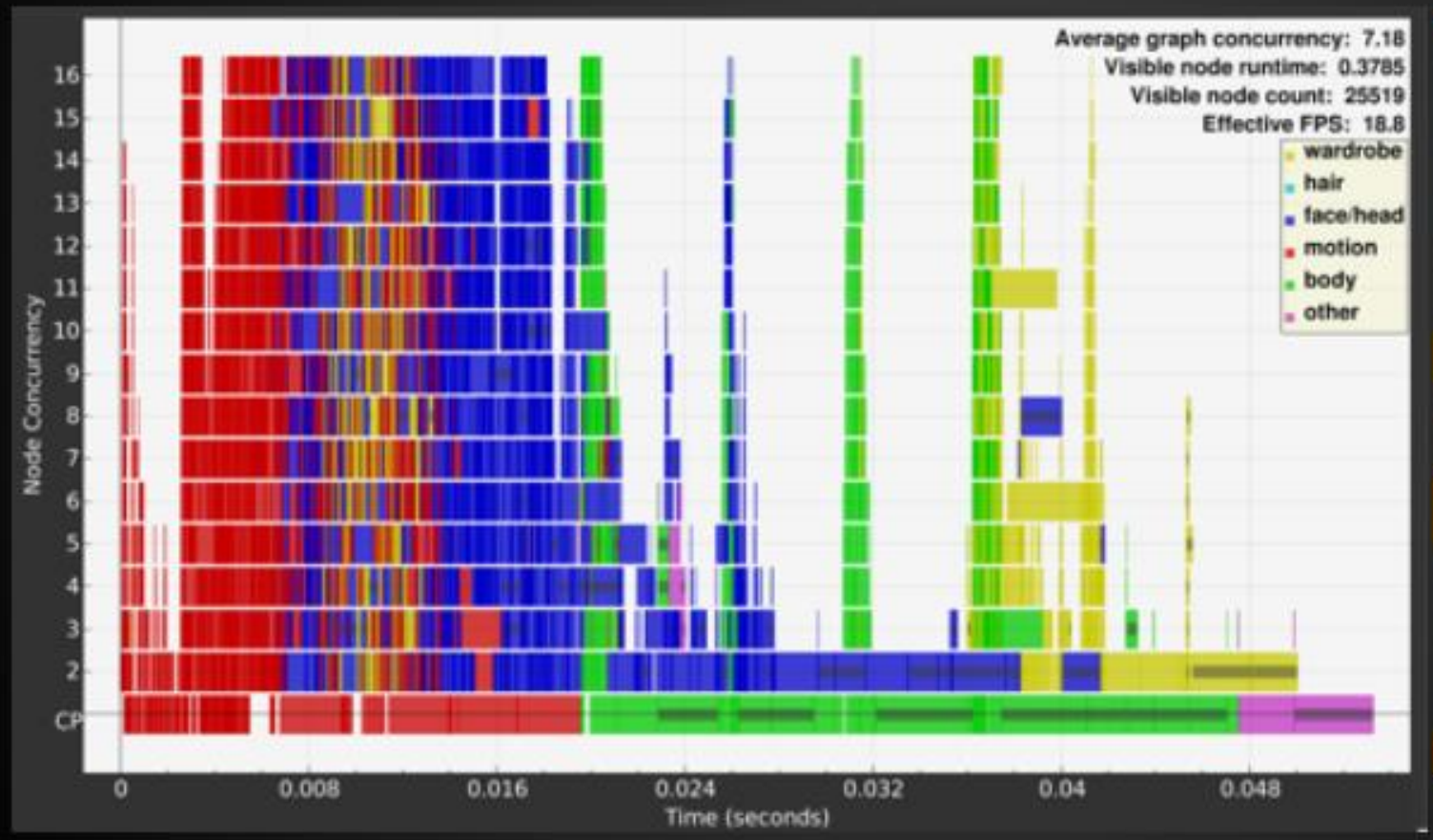
Siggraph 2012

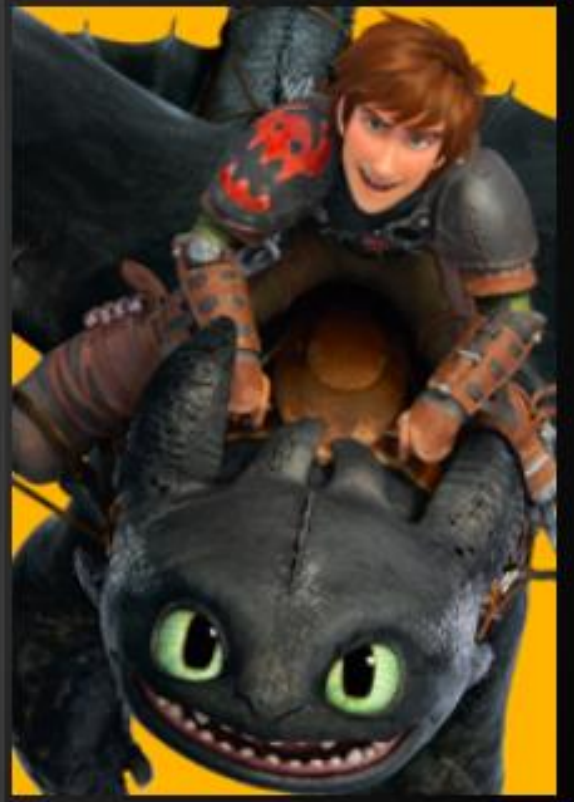


Siggraph 2015

https://www.fxguide.com/quicktakes/dreamworks-premo-animation-system-at-the-sci-tech-awards/

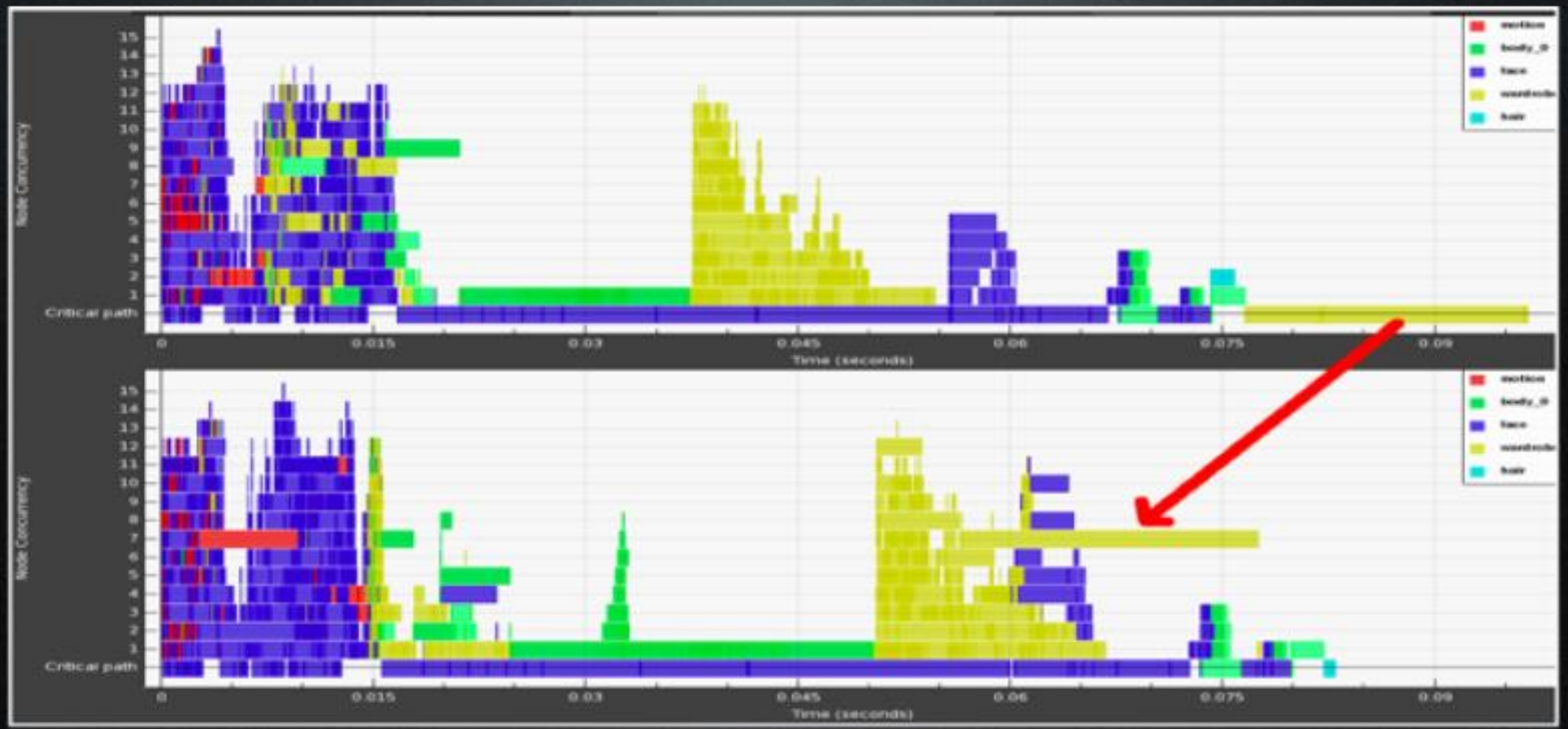
Parallel view (Hiccup, HTTYD2)





Siggraph 2015

Reordering evaluation



Siggraph 2015

Outline

Programmer's performance model

Auto-vectorization and its limits

GPUs and fine-grained SPMD (Single program, multiple data)

Multi-core CPUs and work-stealing

FPGAs and systolic algorithms

Systolic arrays and spatial architecture

Systolic array

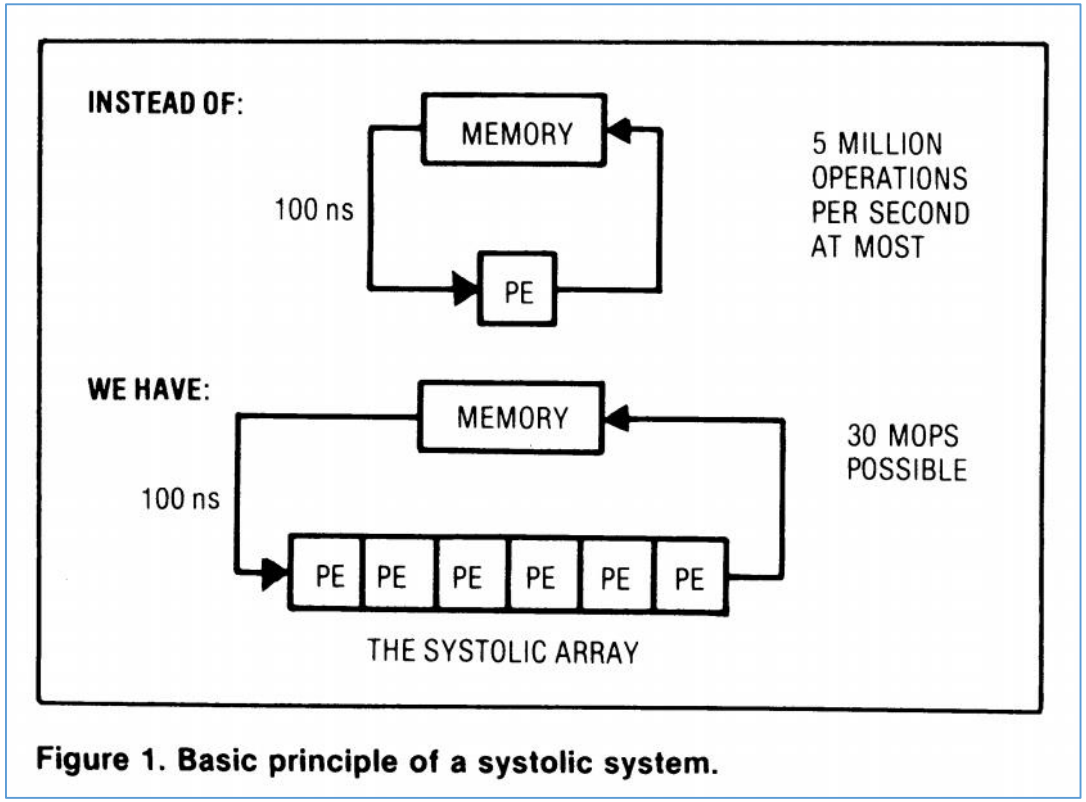
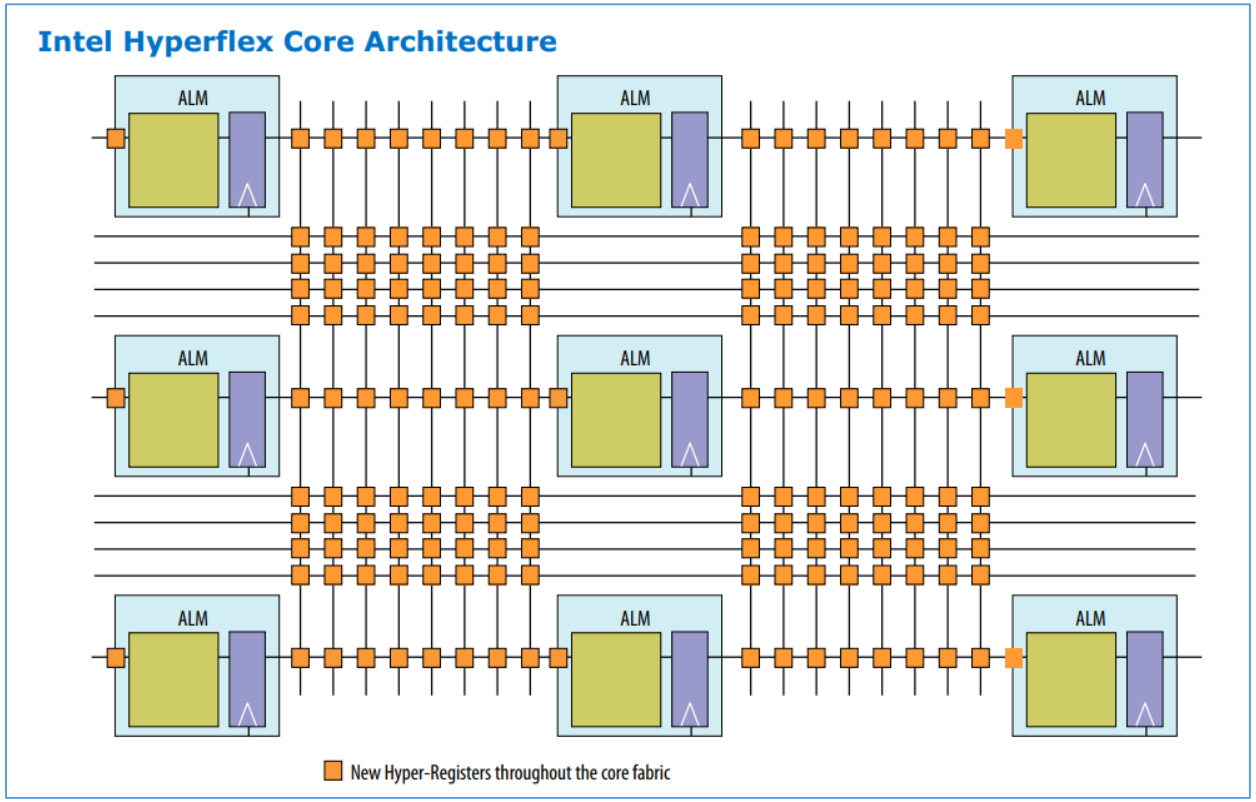


Figure 1. Basic principle of a systolic system.

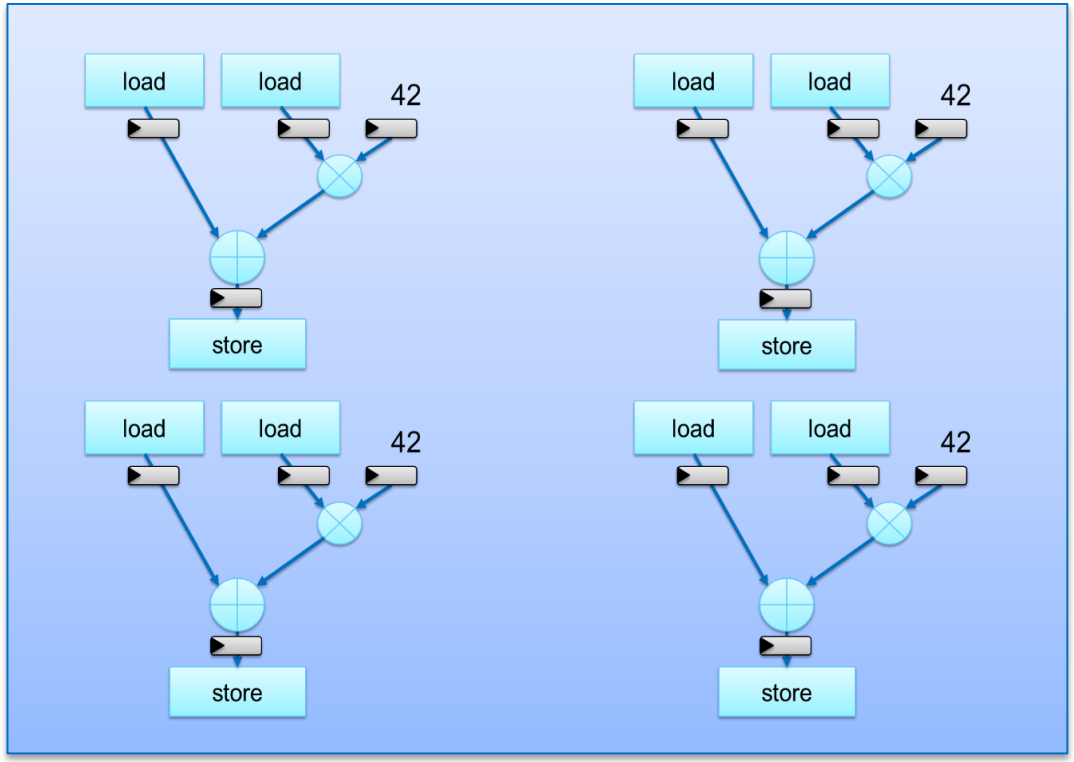
HT Kung, IEEE Computer, 1982

Intel Stratix 10 FPGA



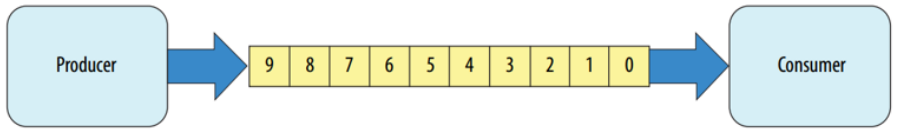
Programming FPGAs

Data parallel kernels communicating through channels



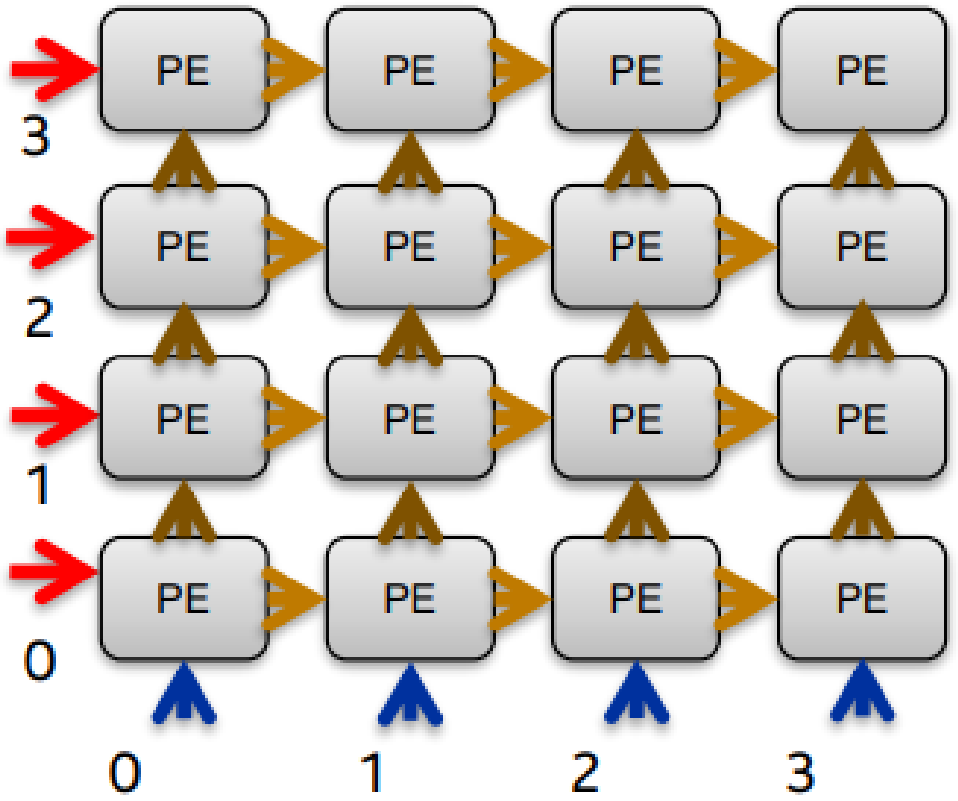
Channel Example

```
#pragma OPENCL EXTENSION cl_altera_channels : enable

channel int c0;

kernel void producer() {
  for(int i=0; i < 10; i++) {
    write_channel_altera(c0, i);
  }
}

kernel void consumer( global uint * restrict dst ) {
  for(int i=0; i < 10; i++) {
    dst[i] = read_channel_altera(c0);
  }
}
```



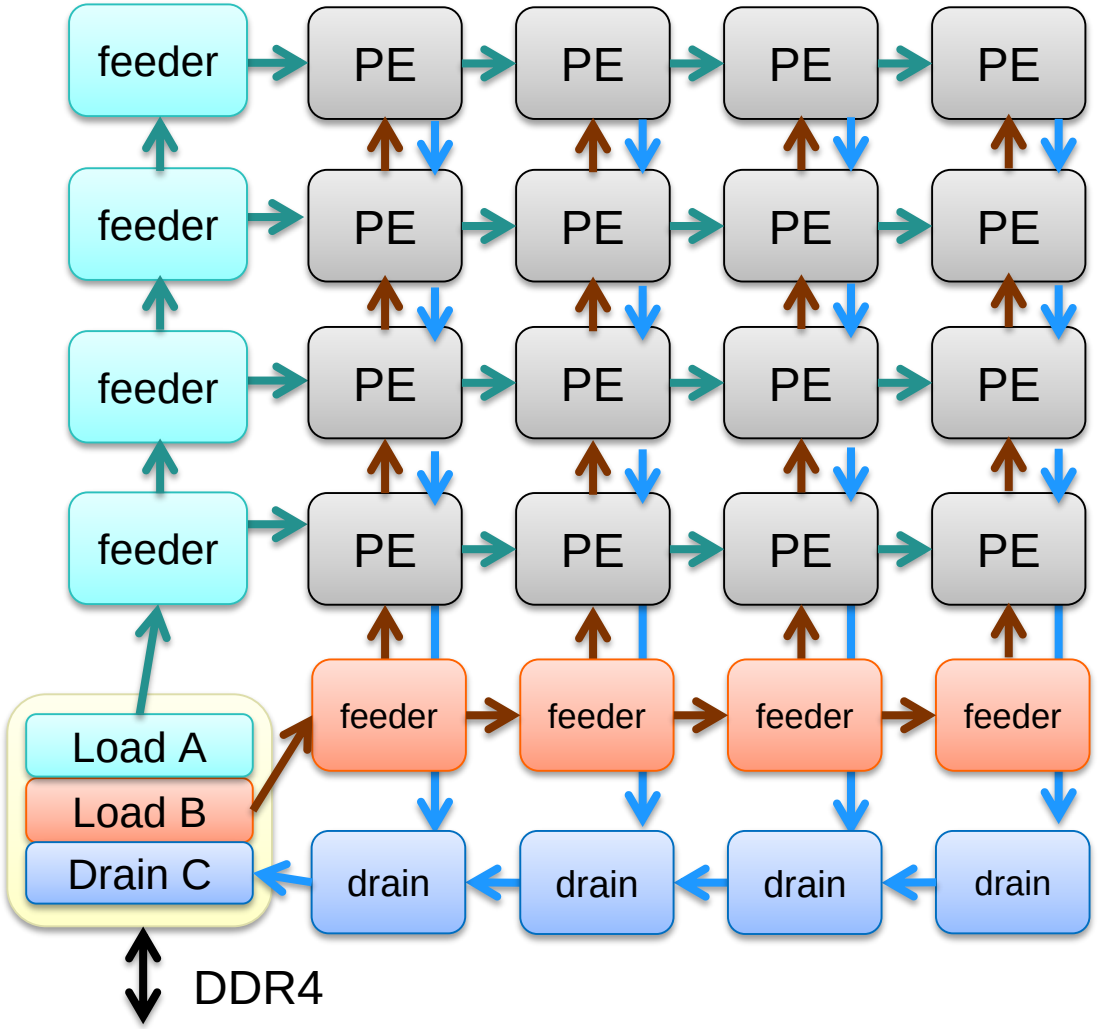
Building a systolic array

```
channel float4 ch_PE_row[4][4];
channel float4 ch_PE_col[4][4];
channel float4 ch_PE_row_side[4];
channel float4 ch_PE_col_side[4];

__attribute__((num_compute_units(4,4)))
kernel void PE() {
   row = get_compute_id(0);
   col = get_compute_id(1);

   float4 a,b;

   if (row==0)
     a = read_channel(ch_PE_col_side[col]);
   else
     a = read_channel(ch_PE_col[row-1][col]);

   if (col==0)

      …
}
```



Systolic matrix multiply

- Every PE is a kernel
- Every feed/drain is a kernel
- Communicate via OpenCL channels
- ~750 lines of OpenCL kernel code



Systolic design obtains peak peformance

Summary

Programmer's performance model

Auto-vectorization and its limits

GPUs and fine-grained SPMD (Single program, multiple data)

Multi-core CPUs and work-stealing

FPGAs and systolic algorithms