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

Can the language with the compiler express the performance features?

Program

Can the compiler and runtime map the model to the hardware?

Performance model

Compiler and runtime

Hardware Platform

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

<table>
<thead>
<tr>
<th>Data Parallel Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
</tr>
<tr>
<td>CTA</td>
</tr>
<tr>
<td>Grid</td>
</tr>
</tbody>
</table>

**Parallel tasks with SIMD kernels**

---

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

**Intel Parallel Programming Model**

<table>
<thead>
<tr>
<th>Multiple cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware threads</td>
</tr>
</tbody>
</table>

**Tasks**
- Intel® Threading Building Blocks
- Intel® Cilk

**Vectors**
- Array Notation
- Intel® Ct technology

---

May 13, 2019
Two multi-core, multi-threaded, SIMD architectures
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

```c
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

May 13, 2019
Simple mapping to CUDA

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*(squez-0.5*con1));
            L[i] = L[i]*vrat;
        }
        ...
    }
}
Similar solutions today in OpenMP and SYCL

```c
#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;
        }
    }
    ...
}
```
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

Instruction set with vectors

A vector register holds 64 64b floating point numbers

Gaussian elimination

\[
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
0 & a_{22} & \cdots & a_{2n} \\
0 & 0 & \cdots & a_{3n} \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & a_{nn} \\
\end{bmatrix}
\begin{bmatrix}
b_1 \\
b_2 \\
b_3 \\
\vdots \\
b_n \\
\end{bmatrix}
\]

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

Pipelined, not SIMD

Cray 1 1976
Cray XMP 1982
DO I = 1,N
A(I) = 3.0*A(I) + (2.0+B(I)) * C(I)
Simple performance model

Programmers model:

**Inner-loop auto-vectorization**, performance measured in chimes.

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

May 13, 2019

Levesque, *A Guidebook to Fortran on Supercomputers*
Similar to RISC pipeline

\[
\text{LD A(1)} \rightarrow r1 \\
\text{LD A(2)} \rightarrow r2 \\
\text{LD B(1)} \rightarrow r3 \\
\text{LD B(2)} \rightarrow r4 \\
\text{FADD 2.0, r1} \rightarrow r5 \\
\text{FADD 2.0, r2} \rightarrow r7 \\
\text{FMUL 3.0, r3} \rightarrow r8 \\
\text{FMUL 3.0, r4} \rightarrow r9 \\
\text{LD C(1)} \rightarrow R10 \\
\text{LD C(2)} \rightarrow R11
\]

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.

Dual issue RISC pipeline

Performance portability with compiler optimization of inner loops
Inner loops are not enough for SIMD
Livermore loop #1

http://www.netlib.org/benchmark/livermore
1000 line kernel, not shown
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

C, C++, ..., FORTRAN

Array of scalar threads

Scalar Compiler
Scalar ISA
Thread Virtualization Tech
SIMD Units

50+ year history of optimizations
NVIDIA GPUs have always used
30+ year history of TLP-to-ILP conversion
Where the efficiency comes from

Volta expands TLP exploitation!

Hot Chips 2017
Matt Pharr: Bring GPU model to CPU

- **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

May 13, 2019
# Performance Portability?

![Performance Portability Chart](image)

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

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.
for (ii = 0; ii < inum; ii++) {
  i = ilist[ii];
  for (jj = 0; jj < jnum; jj++) {
    j = jlist[jj];
    if (rsq < cut_both_sq) {
      if (newton_pair) {
        f[j][0] -= ..
      }
    }
  }
  f[i][0] += ..
}
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
Tasks, not threads

Over-decompose (parallel slack)

Work-stealing for load-balancing

Adapts to the available resources and to the workload
Work-stealing scheduler

May 13, 2019
DreamWorks Animation and Intel

Siggraph 2012

Property of DreamWorks Animation

Siggraph 2015

Parallel view (Hiccup, HTTYD2)
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

Figure 1. Basic principle of a systolic system.

HT Kung, IEEE Computer, 1982

INSTEAD OF:

100 ns

MEMORY

PE

5 MILLION OPERATIONS PER SECOND AT MOST

WE HAVE:

100 ns

MEMORY

PE PE PE PE PE PE PE

THE SYSTOLIC ARRAY

HT Kung, IEEE Computer, 1982

Intel Stratix 10 FPGA

New Hyper-Registers throughout the core fabric
Programming FPGAs

Data parallel kernels communicating through channels

Channel Example

```c
#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);
    }
}
```

May 13, 2019
Building a systolic array

```c
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)))
kernels 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 performance
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