The Ct Virtual Machine: Enabling High Performance Domain Specific Languages and Libraries

Anwar Ghuloum
Agenda

• Ct Primer
• The Ct VM
What is Intel Ct Technology?

- Ct adds parallel collection objects & methods to C++
  - Library interface and is fully ANSI/ISO-compliant (works with ICC, VC++, GCC)
- Ct abstracts away architectural details
  - Vector ISA width / Core count / Memory model / Cache sizes
  - Focus on what to do, not how to do it
  - Sequential semantics
- Ct forward-scales software written today
  - Ct is designed to be dynamically retargetable to SSE, AVX, LRB, ...
- Ct is safe, by default
  - ...but with expert controls to override for performance

Programmers think sequential, not parallel
Collection Objects

**Vec** are the basic type of parallel collection object
- a handle to a value
- managed by the runtime
- flat, multidimensional, or irregularly nested
- created and manipulated exclusively via the API
  - determinism and isolation
  - overrides and control for extra performance

```
#include "ct.h"
using namespace Ct;
int main(int argc, char *argv[]) {
    // A Vec declaration must specify a base
    // type: Vec<basetype> aTypedVector;
    // For example, DoubleVec can refer to any
    // vector of doubles.
    Vec<F64> DoubleVec;

    // A regular 2 dimensional vector:
    Vec2D<I8> 2DMatrix ("{{0,1,2}, {3,4,5}, {6,7,8}}");

    // An irregularly nested vector:
    VecNested<I32> IrregularVector ("{{0,1,2},
                                           {3,4}, {5,6,7,8}}");

    return 0;
}
```
Parallel Operations on Ct Collections

The Ct Runtime Automates This Transformation

Vector Processing

Kernel Processing

Native/Intrinsic Coding

Or Programmers Can Choose Desired Level of Abstraction

Linear algebra, global data movement/communication

Embarrassingly parallel, shaders, image processing

Vec<F32> A, B, C, D;
A += B/C * D;

Elt<F32> kernel(Elt<F32> a, b, c, d) {
    return a + (b/c)*d;
}

Vec<F32> A, B, C, D;
A = map(kernel)(A, B, C, D);

NVec<F32>native(NVec<F32> …) {
    __asm__ {
    …
    }
}

NVec<F32> A, B, C, D;
A = map(native)(A, B, C, D);

CMP
VPREFETCH
FMADD
INC
JMP

CMP
VPREFETCH
FMADD
INC
JMP

Vec<F32> A, B, C, D;
A = map(native)(A, B, C, D);
3D order-6 stencil

```c
#include <stdio.h>

typedef int T;

void proc3DStencilC(T *in, T *out, int nx, int ny, int nz)
{
    for (int k = 0; k < nz; k++)
        for (int j = 0; j < ny; j++)
            for (int i = 0; i < nx; i++)
            {
                out[i+j*nx+k*nz*ny] = 2 * in[i+j*nx+k*nz*ny] - out[i+j*nx+k*nz*ny]
                + out[i+j*nx+k*nz*ny-1]
                + out[i+j*nx+k*nz*ny+1]
                + out[i+j*nx+k*nz*ny-nx]
                + out[i+j*nx+k*nz*ny+nx]
                + out[i+j*nx+k*nz*ny-nx+1]
                + out[i+j*nx+k*nz*ny+nx-1]
                + out[i+j*nx+k*nz*ny-nx-1]
                + out[i+j*nx+k*nz*ny+nx+1];
            }
}

void proc3DStencilCCT(T *in, T *out, int nx, int ny, int nz)
{
    for (int k = 0; k < nz; k++)
        for (int j = 0; j < ny; j++)
            for (int i = 0; i < nx; i++)
            {
                out[i+j*nx+k*nz*ny] = 2 * in[i+j*nx+k*nz*ny] - out[i+j*nx+k*nz*ny]
                + out[i+j*nx+k*nz*ny-1]
                + out[i+j*nx+k*nz*ny+1]
                + out[i+j*nx+k*nz*ny-nx]
                + out[i+j*nx+k*nz*ny+nx]
                + out[i+j*nx+k*nz*ny-nx+1]
                + out[i+j*nx+k*nz*ny+nx-1]
                + out[i+j*nx+k*nz*ny-nx-1]
                + out[i+j*nx+k*nz*ny+nx+1];
            }
}
```

Original Code

```c
#include <stdio.h>

typedef int T;

void proc3DStencilC(T *in, T *out, int nx, int ny, int nz)
{
    for (int k = 0; k < nz; k++)
        for (int j = 0; j < ny; j++)
            for (int i = 0; i < nx; i++)
            {
                out[i+j*nx+k*nz*ny] = 2 * in[i+j*nx+k*nz*ny] - out[i+j*nx+k*nz*ny]
                + out[i+j*nx+k*nz*ny-1]
                + out[i+j*nx+k*nz*ny+1]
                + out[i+j*nx+k*nz*ny-nx]
                + out[i+j*nx+k*nz*ny+nx]
                + out[i+j*nx+k*nz*ny-nx+1]
                + out[i+j*nx+k*nz*ny+nx-1]
                + out[i+j*nx+k*nz*ny-nx-1]
                + out[i+j*nx+k*nz*ny+nx+1];
            }
}
```

Ct Code
How Does it Really Work?

_Ct is really a high-level APIs..._

...that streams opcodes to an optimizing virtual machine

The source (front-end) can be anything:

- A new language
- A bytecode parser
  - Experiments with Python, HLSL
- An application-specific library
- A compiler front-end
The Ct VM

Ct API (Average C++ Developer)

VM IR (Language Implementor)

Ct’s Hardware Abstraction Layer

IA-based Virtual ISA

Backend JIT/Compiler

Ct+ Opcode API

Ct JIT/Compiler

Debug/PerfSvcs

Memory Manager

Task/Threading Runtime

Other Languages!

Other Back-ends

C, C2, Ci*

LRB

Hybrid

Other Languages!
float src1[], src2[], dest[];
Vec<F32>a(src1,N), b(src2,N);
rcall(foo)(a, b)
...
foo(Vec<F32> a, Vec<F32> b) {
    Vec<F32> c = a + b;
    Vec<F32> d = c * a;
    return;
}
Why Does this Matter for C/C++ Developers?

*It’s not just a single kernel…*

- Productivity craters when many kernels have to be tuned
  - Focusing energy on 1 algorithm makes sense, if it is the dominant algorithm

...*in one place*

- Widely used libraries often give up performance for well designed generic interfaces

→ Inherently spreads compute across methods
• Software is often architected for reuse, replacement, extension:
  – Use of abstract classes, virtual function calls, C++ iterators, indirection is the norm...
• “Performance paths” are often spread across many objects and files
Performance Without De-architecting Software

• Performance tools typically want to see *everything*!
• You look at all possible/likely paths
  – Brittle
  – Difficult to maintain
  – Difficult to extend
  – Difficult to program

De-architecting for performance
• Combine good software practices and performance with Ct:
  - Pepper your models/classes with Ct
  - Ct’s VM takes care of dynamically gathering the performance paths

Ct in your Classes
Concluding Remarks

• Managed/dynamic runtimes are no longer synonymous with poor performance
  → You don’t have to sacrifice productivity for performance

• The pace of credible language emergence will be sustained
  – A new language every 18 months
  – It may even grow, driven by architectural/application innovation and specialization