Ct: Channeling NeSL and SISAL in C++

Anwar Ghuloum

CLARA Group

Programming Systems Lab

Corporate Technology Group

Agenda

- Entering the Many-core Era
- Ct: A Bridge
- The Long Term Opportunity

Process Scaling Trends

Every process step :

- Shrinks linear dimension by 30%
- Capacitance shrinks by 30%
- Max voltage decreases by 10%
- Switching time (@Vmax) shrinks by 30%
  - Frequency increases by ~40%

Transistor Scaling

~= 2x density

~= 50% less area

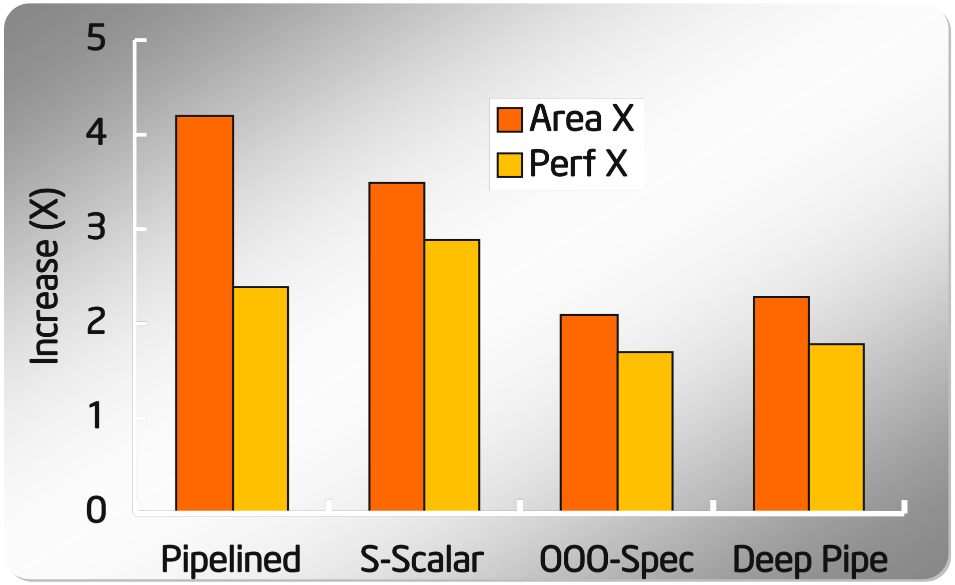
Power Scaling

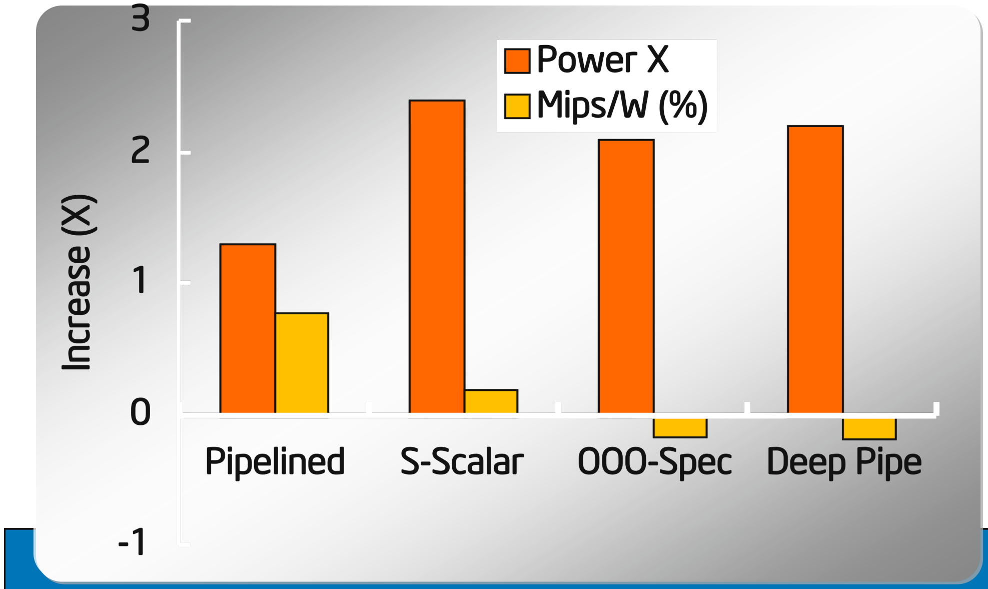
~= transistors * cap/trans * voltage^2 * 1/time

~= 2 * 0.7 * 0.9^2 * 1/0.7

~= 1.62X power increase

uArch Features and Perf/Watt





Moore’s Law ⇨ more transistors for advanced architectures

Pushed frequency beyond limit

Dramatically increased transistor subthreshold leakage

Increased pipeline depth

Delivered higher peak performance

But…

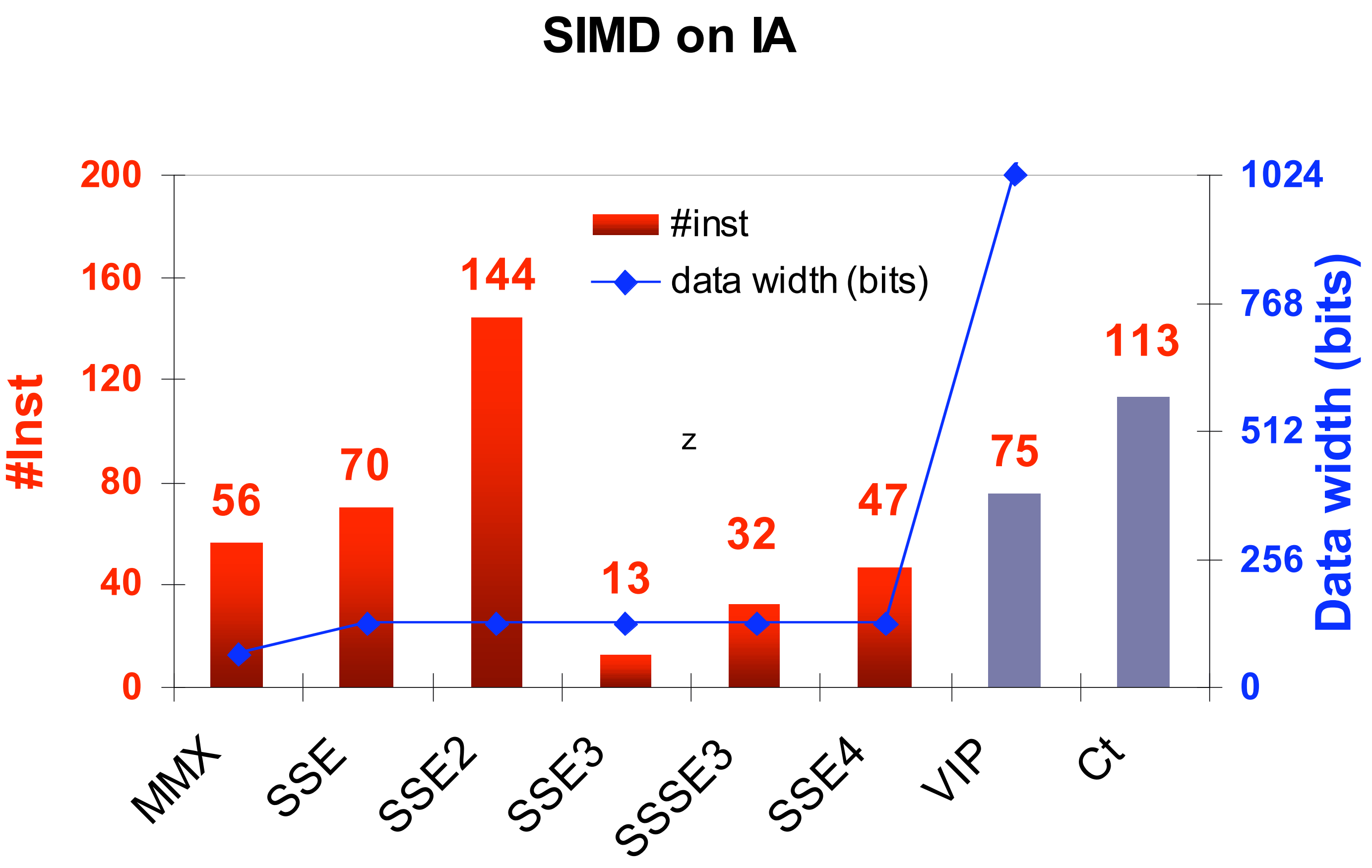
With lower power efficiency

Architecture is Power Limited

- Power increasing ~50% each generation
  → Perf/Watt is increasingly important
- Power efficiency can be gained through:
  - More, simpler cores
    - >Leverage increased density while decreasing per core power
  - Longer vector ISA
    - >Reduced front-end power
  - VLIW
    - >Expose ILP to compiler

All of these approaches expose parallelism to software.

Example: An Acceleration in ISA Enhancement?



What Software Vendors are Telling Us

- Programming parallel applications is 10,100,1000x* *less* productive than sequential
  - Non-deterministic programming errors
  - Performance tuning is extremely microarchitecture-dependent
- Parallel HW is here today, better programming tools are needed to take advantage of these capabilities
  - Quad core on desktop arrived nearly a year months ago
  - Multi- and Many-core DP and MP machines are on the way
  - (Also, programmable GPUs going on 8 years)
- Strong interest by ISVs for a parallel programming model which is:
  - **Easy to use *and* high performance: sounds difficult already!**
  - **Portable**: Desire the flexibility to target various HW platforms and adapt to future variations

**Depends on which developer you ask.*

Our Approach(es)

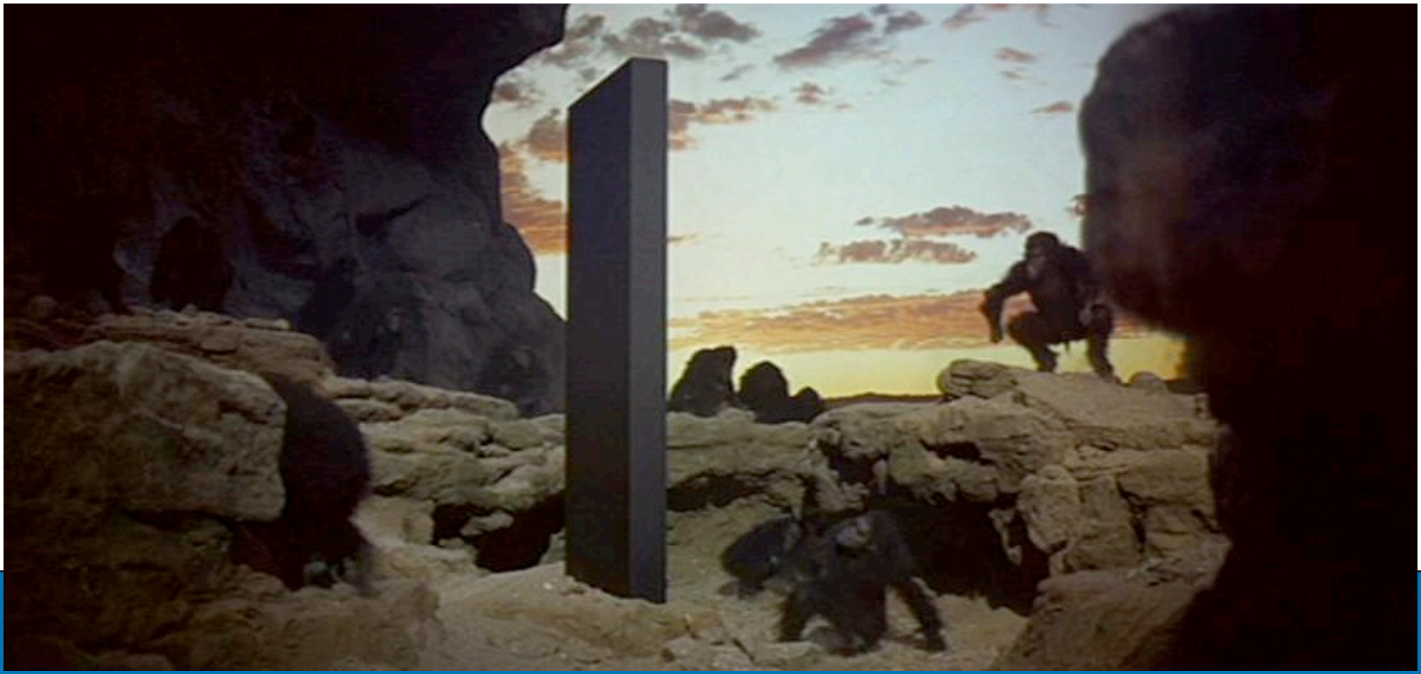
As a chip company shipping multi-core CPUs, we want to enable as many applications as possible

- As a company shipping multi-core CPUs *today* (actually, the last couple of years), we need to prioritize near term enabling
- As a chip company shipping many-core CPUs soon, we need start working on a longer term solution

We are *commercially using functional programming* ideas (in the near term) and languages (in the long term) to enable *a lot of parallel applications*

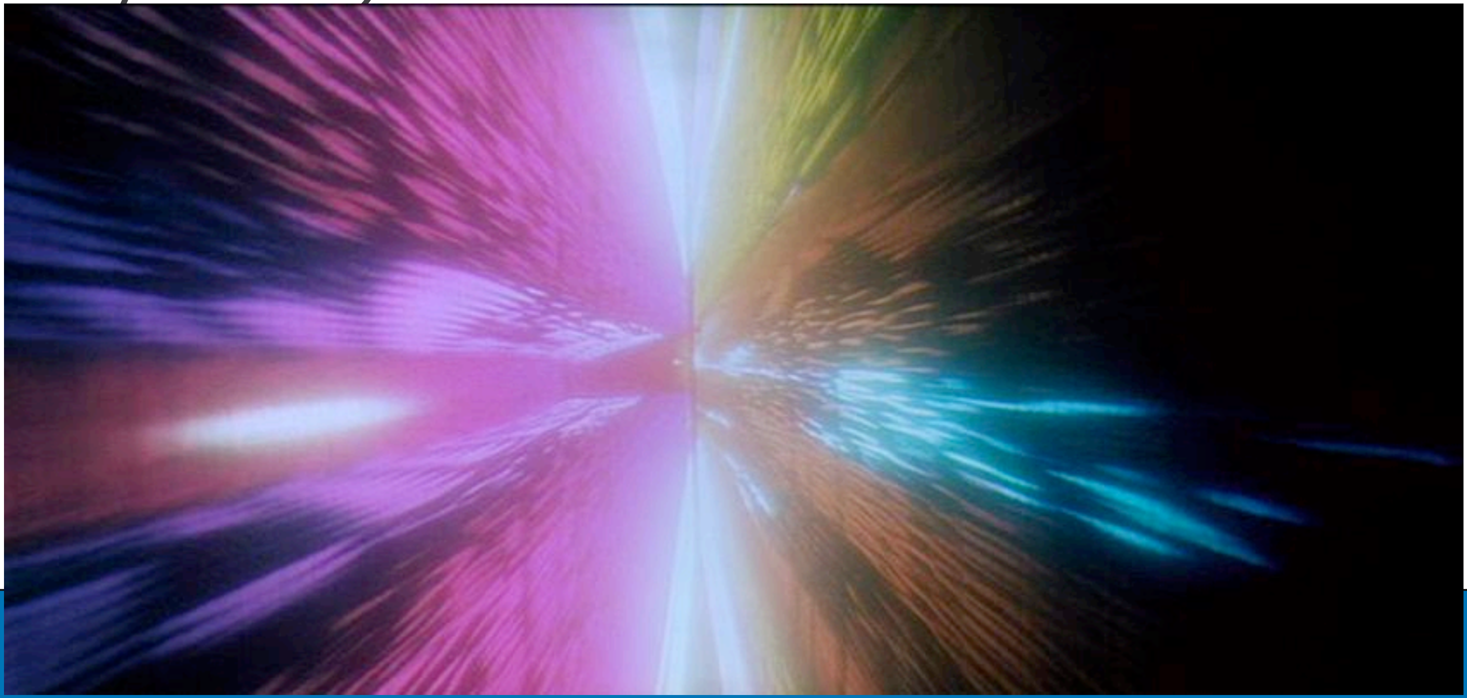
Approach 1: A Bridging Model (e.g. Ct)

- Ease the transition to parallelism
- Experiment with parallel programming idioms
- Based on existing prevalent languages
- "Retrofit" useful features and semantics into the language
- (Focus of this talk)



Approach 2: Start Designing the Languages of the Next Decade

- Use wealth of experience with languages
- Use experiences gained with approach 1
- Strong intuition that one or more mainstream languages will come from the PL community
- (Stay tuned)



Approach 1: Why Data Parallelism?

"Good" reasons

- Deterministic model
  - >Data races are designed out
  - >Behavior on 1 core is the same as behavior on n cores
- Performance is predictable
  - >Simple model for each flavor of data parallel operator
- High performance is achievable
- Highly portable
  - >Threaded & SIMD architectures
- Expressive
  - >Especially when application usage patterns considered

"Bad" reasons

- Bottom-up design: Architectural constraints

What is “Nested” Data Parallelism?

Flat data parallel models (e.g. APL, F90/HPF, GPGPU)

- Flat (or limited dimensionality) vectors
- Operators over vectors
  - > Element-wise operators
  - > Limited collective communication operations (reductions)
  - > Some constrained permutation
  - > Masking operations

IMO: Streaming & flat data parallel are roughly equivalent in expressiveness

Nested data parallel models added (e.g. Nesl, APL2, Paralations)

+ (Irregularly) nested and sparse/indexed vectors
  + Extend all operators to work generically on various vector types
  + Richer set of collective communication operations
    +Scans, Combining-send/Multi-reduce, Multi-prefix

Design Constraints (For a Bridging Model)

Target language: C/C++ (and maybe Fortran, Java, etc.)

- These are and will continue to be the dominant languages for high performance computing for the next 5 years

…and we *mean* **standard** C and C++!

- Custom syntactic extensions face huge barriers to adoption
- It is possible to design a desirable semantics through an API-like interface with some Macro magic

…and all the “baggage” that comes with those languages

- Must co-exist with legacy APIs, libraries
- Must co-exist with prevailing parallelism APIs (Pthreads, winthreads, OpenMP, MPI)

Ct is….

- …an “extension” of C++ for throughput computing
- …like a library implementation of a STL-style container
    - A pure functional mini-language with call-by-value semantics
- …using a (dynamically linked) runtime to optimize and generate code
- …designed to *forward-scale* software

TVECs

The basic type in Ct is a TVEC

- TVECs are managed by the Ct runtime
- TVECs are single-assignment vectors
- TVECs are (opaquely) flat, multidimensional, sparse, or nested
- TVEC values are created & manipulated exclusively through Ct API

Declared TVECs are references to (immutable) values

```
TVEC<F64> DoubleVec;   // DoubleVec can refer to any vector of doubles
…
DoubleVec = Src1 + Src2;
…
DoubleVec = Src3 * Src4;
```

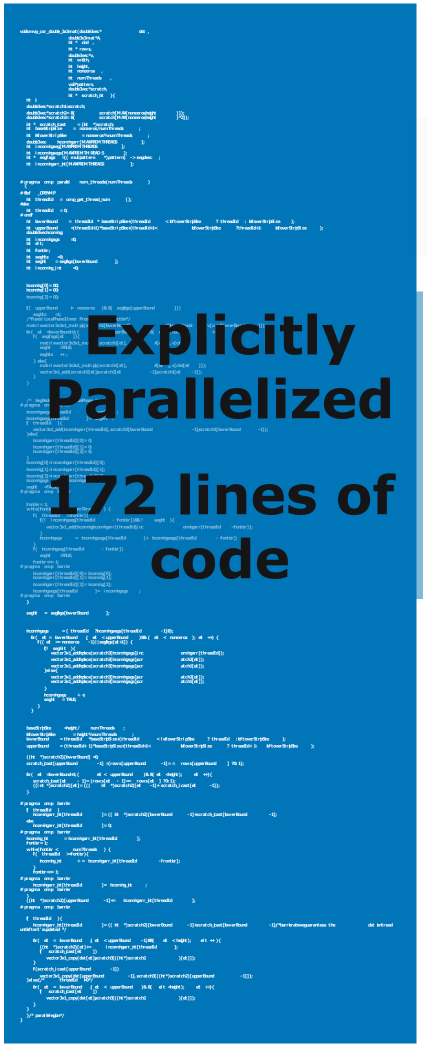
Assigning a value to DoubleVec doesn’t modify the value representing the result of the add, it simply refers to a *new* value.

Ct Example: Sparse Matrix Vector Product

```
TVEC<F64> SparseMatrixVectorProductCSC(TVEC<F64> A, TVEC<I32> rind,
                                          TVEC<I32> cols, TVEC<F64> v) {
// computes A*x, where A,rind,cols is a compressed sparse column vector
    TVEC<F64> expv, product, result;
    expv = v.distribute(cols);                // replicates elements of v
    product = A*expv;                         // performs inner product of A, v
    product = product.applyNesting(rind,ctSparse); // make the product indexed
    return product.reduceSum();               // performs row-wise reduction
                                              // (implicitly a combining-send)
}
```

Ct compiler and runtime automatically take care of threading and vector ISA

Productive Programming with Ct



Explicitly Parallelized

172 lines of code

Ct: <6 lines of code, faster, scalable

```
TVEC<F64> smvpCSC(TVEC<F64> A, TVEC<I32> rind,
                  TVEC<I32> cols, TVEC<F64> v) {
    TVEC<F64> expv, product, result;
    expv = v.distribute(cols);
    product = A*expv;
    product = product.applyNesting(rind,
ctSparse);
    return product.reduceSum();
}
```

- Race-free programming with on-the-fly, automatic generation of threads tailored to user's *multi-core* hardware
- Simpler, high performance, scalable, SSE-friendly parallel code
- Library-like interface compatible with existing programming environments and APIs

Code Comparison: Black-Scholes

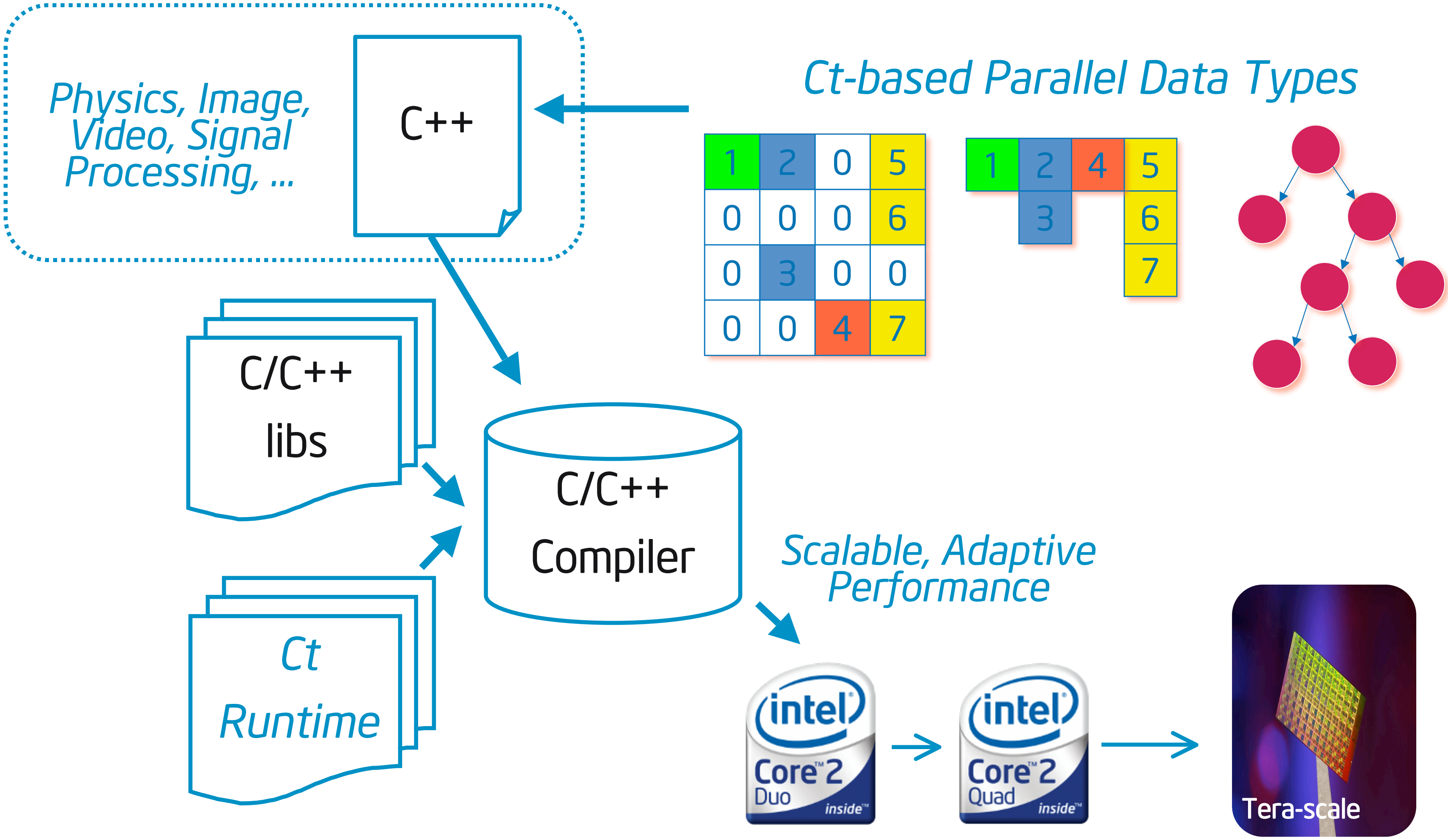
```
template <typename T>
TVEC<T> CND(TVEC<T> x)
{
    TVEC<T> l = abs(x);
    TVEC<T> k = 1.0f / ( 1.0f + 0.2316419f * l);

    TVEC<T> w =
        0.31938153f * k -
        0.356563782f * k * k +
        1.781477937f * k * k * k -
        1.821255978f * k * k * k * k +
        1.330274429f * k * k * k * k * k;

    w = w * inv_sqrt_2xPI * exp(l * l * -0.5f);
    w = select(x > 0, 1.0f - w, w);
    return w;
}

template <typename T>
void ctBlackScholes(T *option_p[illegible]e,
                  int num_options,
                  T *stkprice,
                  T *strike,
                  T *rate,
                  T *volatility,
                  T *time)
{
    __CT__ {
        TVEC<T> s(stkprice, num_options);
        TVEC<T> x(strike, num_options);
        TVEC<T> r(rate, num_options);
        TVEC<T> v(volatility, num_options);
        TVEC<T> t(time, num_options);

        TVEC<T> sqrt_value = v * sqrt(t);
        TVEC<T> d1 = ln(s / x) + (r + v * v * 0.5f ) * t) / sqrt_value;
        TVEC<T> d2 = d1 - sqrt_value;

        TVEC<T> result = x * exp(0f - r * t) * (1.0f - CND(d2)) + (-s) * (1.0 - CND(d1));
        result.copyOut(option_price, num_options);
    }
}
```

Ct

```
#define NCO 4

#if (NCO==2)
#define fptype double
#define SIMD_WIDTH 2
#define _MMR      __m128d
#define _MM_LOAD  _mm_load_pd
#define _MM_STORE _mm_store_pd
#define _MM_MUL   _mm_mul_pd
#define _MM_ADD   _mm_add_pd
#define _MM_SUB   _mm_sub_pd
#define _MM_DIV   _mm_div_pd
#define _MM_SQRT  _mm_sqrt_pd
#define _MM_SET(A)  _mm_set_pd(A,A)
#define _MM_SETR  _mm_set_pd
#endif

#if (NCO==4)
#define fptype float
#define SIMD_WIDTH 4
#define _MMR      __m128
#define _MM_LOAD  _mm_load_ps
#define _MM_STORE _mm_store_ps
#define _MM_MUL   _mm_mul_ps
#define _MM_ADD   _mm_add_ps
#define _MM_SUB   _mm_sub_ps
#define _MM_DIV   _mm_div_ps
#define _MM_SQRT  _mm_sqrt_ps
#define _MM_SET(A)  _mm_set_ps(A,A,A,A)
#define _MM_SETR  _mm_set_ps
#endif

__forceinline void CNDF ( fptype * OutputX, fptype * InputX )
{
   _MM_ALIGN16 int sign[SIMD_WIDTH];
   int i;
   _MMR xInput;
   _MMR xNPrimeofX;
   _MM_ALIGN16 fptype expValues[SIMD_WIDTH];
   _MMR xK2;
   _MMR xK2_2, xK2_3, xK2_4, xK2_5;
   _MMR xLocal, xLocal_1, xLocal_2, xLocal_3;

   for (i=0; i<SIMD_WIDTH; i++) {
      // Check for negative value of InputX
      if (InputX[i] < 0.0) {
         InputX[i] = -InputX[i];
         sign[i] = 1;
      } else
         sign[i] = 0;
   }

   xInput = _MM_LOAD(InputX);

   // Compute NPrimeX term common to both four & six decima
accuracy calcs
   for (i=0; i<SIMD_WIDTH; i++) {
      expValues[i] = exp(-0.5f * InputX[i] * InputX[i]);
      // printf("exp[%d]: %f\n", i, expValues[i]);
   }

   xNPrimeofX = _MM_LOAD(expValues);
   xNPrimeofX = _MM_MUL(xNPrimeofX,
_MM_SET(inv_sqrt_2xPI));

   xK2 = _MM_MUL(_MM_SET((fptype)0.2316419), xInput);
   xK2 = _MM_ADD(xK2, _MM_SET((fptype)1.0));
   xK2 = _MM_DIV(_MM_SET((fptype)1.0), xK2);
   // xK2 = _mm_rcp_pd(xK2);  // No rcp function for double-
precision

   xK2_2 = _MM_MUL(xK2, xK2);
   xK2_3 = _MM_MUL(xK2_2, xK2);
   xK2_4 = _MM_MUL(xK2_3, xK2);
   xK2_5 = _MM_MUL(xK2_4, xK2);

   xLocal_1 = _MM_MUL(xK2, _MM_SET((fptype)0.31938153(
   xLocal_2 = _MM_MUL(xK2_2, _MM_SET((fptype)-
0.356563782));
   xLocal_3 = _MM_MUL(xK2_3,
_MM_SET((fptype)1.781477937));
   xLocal_2 = _MM_ADD(xLocal_2, xLocal_3);
   xLocal_3 = _MM_MUL(xK2_4, _MM_SET((fptype)-
1.821255978));
   xLocal_2 = _MM_ADD(xLocal_2, xLocal_3);
   xLocal_3 = _MM_MUL(xK2_5,
_MM_SET((fptype)1.330274429));
   xLocal_2 = _MM_ADD(xLocal_2, xLocal_3);

   xLocal_1 = _MM_ADD(xLocal_2, xLocal_1);
   xLocal   = _MM_MUL(xLocal_1, xNPrimeofX);
   xLocal   = _MM_SUB(_MM_SET((fptype)1.0), xLocal);

   _MM_STORE(OutputX, xLocal);
   // _mm_storel_pd(&OutputX[0], xLocal);
   // _mm_storeh_pd(&OutputX[1], xLocal);

   for (i=0; i<SIMD_WIDTH; i++) {
      if (sign[i]) {
         OutputX[i] = ((fptype)1.0 - OutputX[i]);
      }
   }
}

void BlkSchlsEqEuroNoDiv (fptype * OptionPrice, int numOptions,
fptype * sptprice,
fptype * strike, fptype * rate, fptype * volatility,
                    fptype * time, int * otype, float timet)
{
   int i;
// local private working variables for the calculation
   _MMR xStockPrice;
   _MMR xStrikePrice;
   _MMR xRiskFreeRate;
   _MMR xVolatility;
   _MMR xTime;
   _MMR xSqrtTime;

   _MM_ALIGN16 fptype logValues[NCO];
   _MMR xLogTerm;
   _MMR xD1, xD2;
   _MMR xPowerTerm;
   _MMR xDen;
   _MM_ALIGN16 fptype d1[SIMD_WIDTH];
   _MM_ALIGN16 fptype d2[SIMD_WIDTH];
   _MM_ALIGN16 fptype FutureValueX[SIMD_WIDTH];
   _MM_ALIGN16 fptype NofXd1[SIMD_WIDTH];
   _MM_ALIGN16 fptype NofXd2[SIMD_WIDTH];
   _MM_ALIGN16 fptype NegNofXd1[SIMD_WIDTH];
   _MM_ALIGN16 fptype NegNofXd2[SIMD_WIDTH];

   xStockPrice = _MM_LOAD(sptprice);
   xStrikePrice = _MM_LOAD(strike);
   xRiskFreeRate = _MM_LOAD(rate);
   xVolatility = _MM_LOAD(volatility);
   xTime = _MM_LOAD(time);

   xSqrtTime = _MM_SQRT(xTime);

   for(i=0; i<SIMD_WIDTH;i ++) {
      logValues[i] = log(sptprice[i] / strike[i]);
   [illegible]

   xLo[illegible] = _MM_[illegible](&(logVa[illegible]));

   xPowerTerm = _M[illegible](xVolatility, xVolatility);
   [illegible]erTerm = _MM_[illegible]Term, _MM_S[illegible]
   [illegible]Term = _mm_[illegible]rm, _M[illegible]
2.0));

   xD1 = [illegible]_ADD[illegible]FreeRate, x[illegible]Term);
   xD[illegible]_SUB([illegible]Term);

   xD1 = _MM_MUL(xD1, xTime);
   xD2 = _MM_MUL(xD2, xTime);

   xD1 = _MM_ADD(xD1, xLogTerm);
   xD2 = _MM_ADD(xD2, xLogTerm);

   xDen = _MM_MUL(xVolatility, xSqrtTime);
   xD1 = _MM_DIV(xD1, xDen);
   // VL: 10/15/06. An optimization is not to recompute xD2, but to
derive it from xD1
   // xD2 = _MM_DIV(xD2, xDen);
   xD2 = _MM_SUB(xD1, xDen);

   _MM_STORE(d1, xD1);
   _MM_STORE(d2, xD2);

   CNDF( NofXd1, d1 );
   CNDF( NofXd2, d2 );

   for (i=0; i<SIMD_WIDTH; i++) {
      FutureValueX[i] = strike[i] * (exp(-(rate[i])*(time[i])));

      NegNofXd1[i] = ((fptype)1.0 - (NofXd1[i]));
      NegNofXd2[i] = ((fptype)1.0 - (NofXd2[i]));
      OptionPrice[i] = (FutureValueX[i] * NegNofXd2[i]) -
(sptprice[i] * NegNofXd1[i]);
   }
}

void sseBlackScholes(fptype *option_price,
               int num_options,
               fptype *stkprice,
               fptype *strike,
               fptype *rate,
               fptype *volatility,
               fptype *time)
{
   for (int i = 0; i < num_options; i += NCO) {
      // Calling main function to calculate option value based on
Black & Sholes's
      // equation.
      BlkSchlsEqEuroNoDiv(&(option_price[i]), NCO, &(stkprice[i]),
&(strike[i]),
                     &(rate[i]), &(volatility[i]), &(time[i]),
NULL/*&(otype[i])*/, 0);
   }
}
```

SSE

Using Ct with Legacy Application Build Environments



Physics, Image, Video, Signal Processing, …

C++

Ct-based Parallel Data Types

1	2	0	5
0	0	0	6
0	3	0	0
0	0	4	7

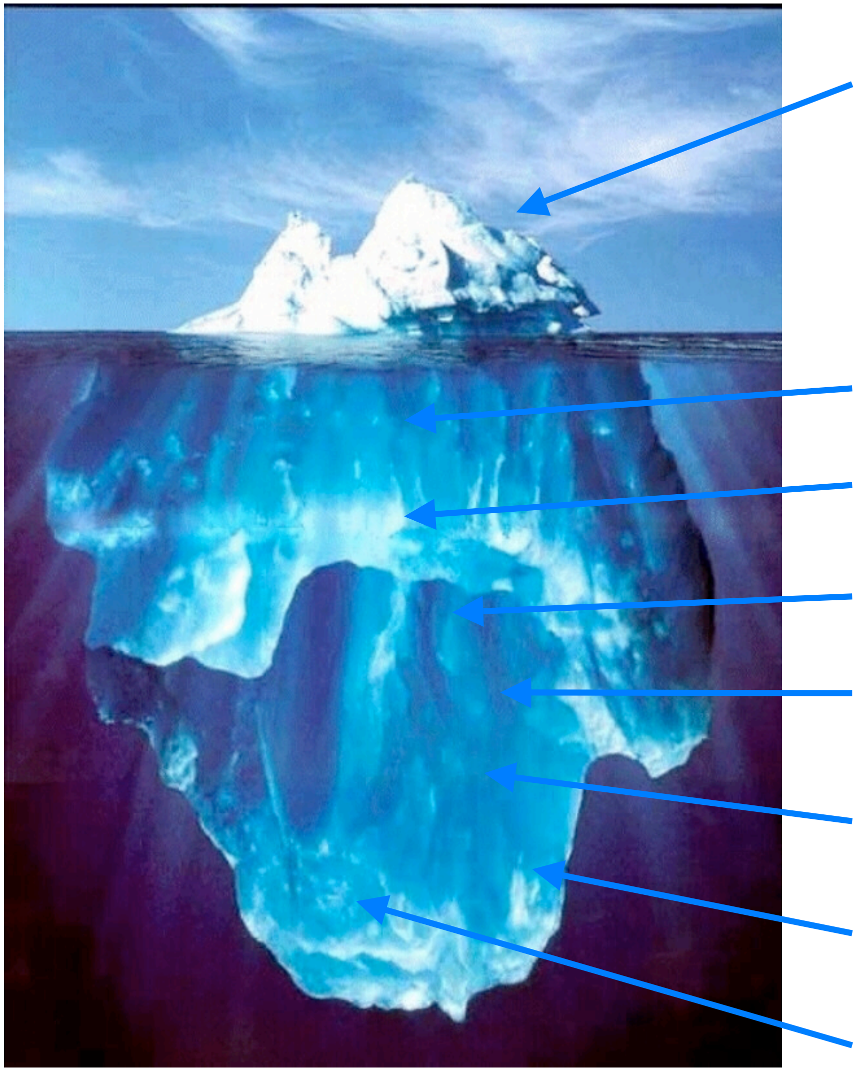
C/C++ libs

Ct Runtime

C/C++ Compiler

Scalable, Adaptive Performance

Language Vehicle for Parallel Programming Systems Research



Ct Api

- Nested Data Parallelism

- Deterministic Task Parallelism

Deterministic parallel programming

Fine grained concurrency and synch

Dynamic compilation for DP

High-performance memory management

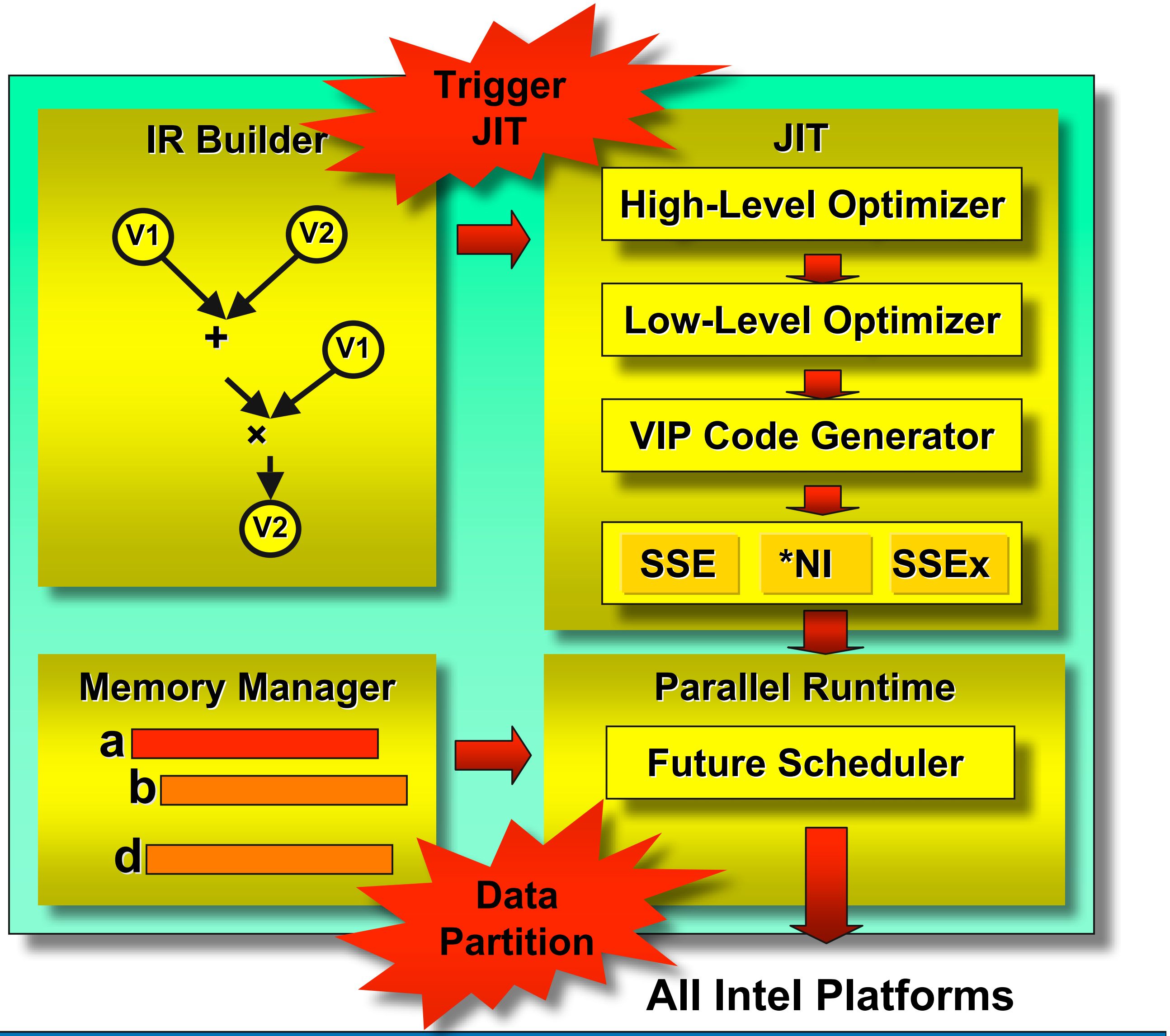
Forward-scaling binaries for SSE2/3/4/x, *NI

Parallel application library development

Performance tools for Future Architectures

Ct: Dynamic Compilation + Virtual Machine

```
float src1[], src2[], dest[];

TVEC<F32> a(src1), b(src2);
TVEC<F32> c = a + b;
TVEC<F32> d = c * a;
d.copyOut(dest);
```

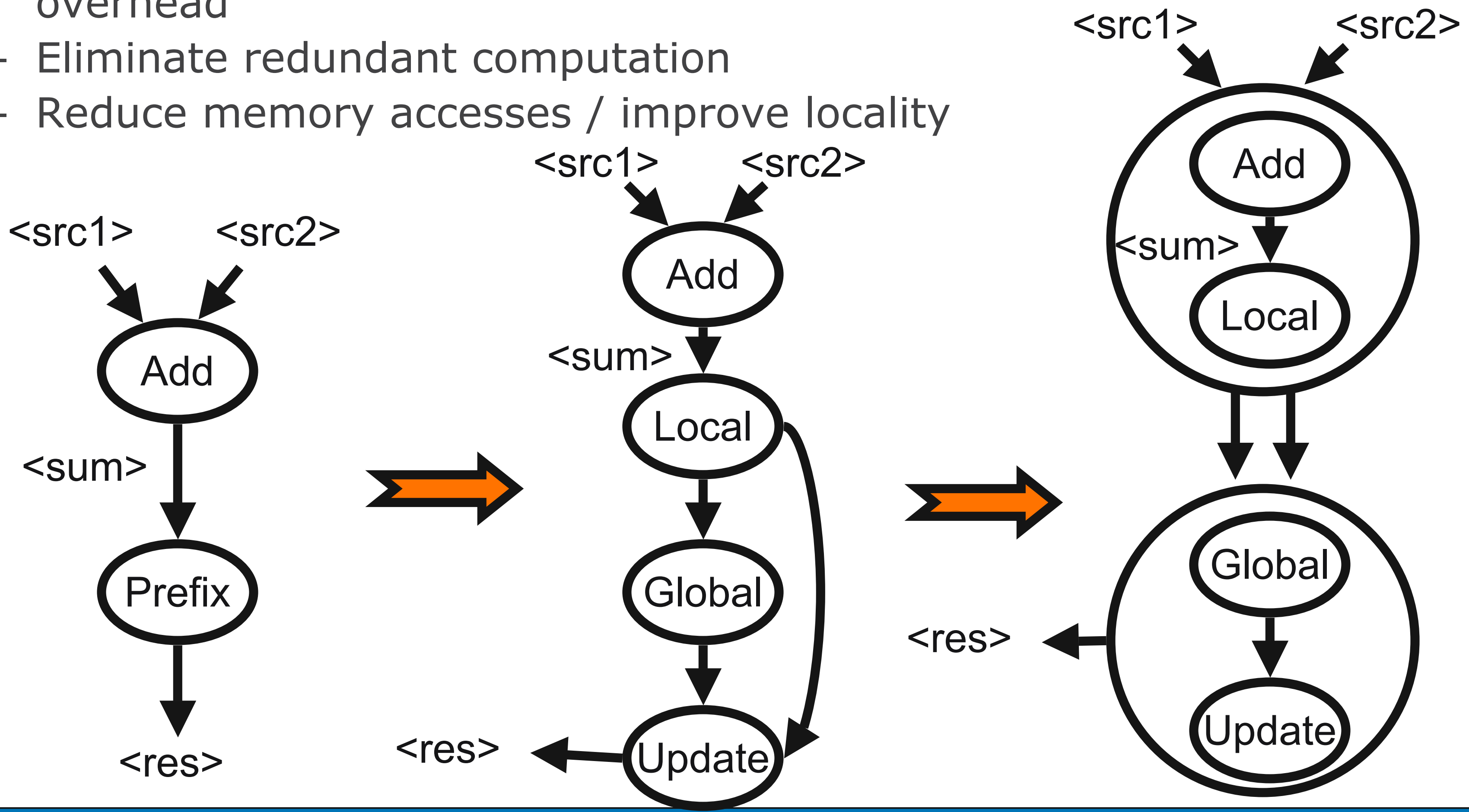


Ct Dynamic Engine

Use Animation

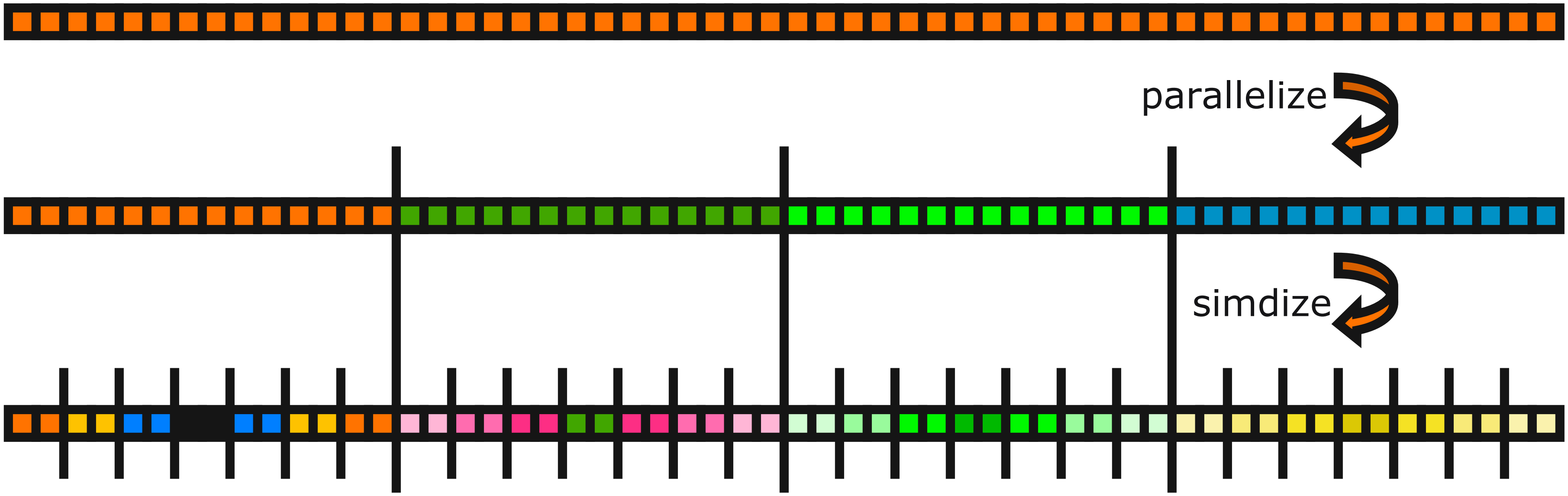
High-Level Optimizer

- ~20 optimizations (including classic opts)
- Increase granularity of parallelism / decrease threading overhead
- Eliminate redundant computation
- Reduce memory accesses / improve locality

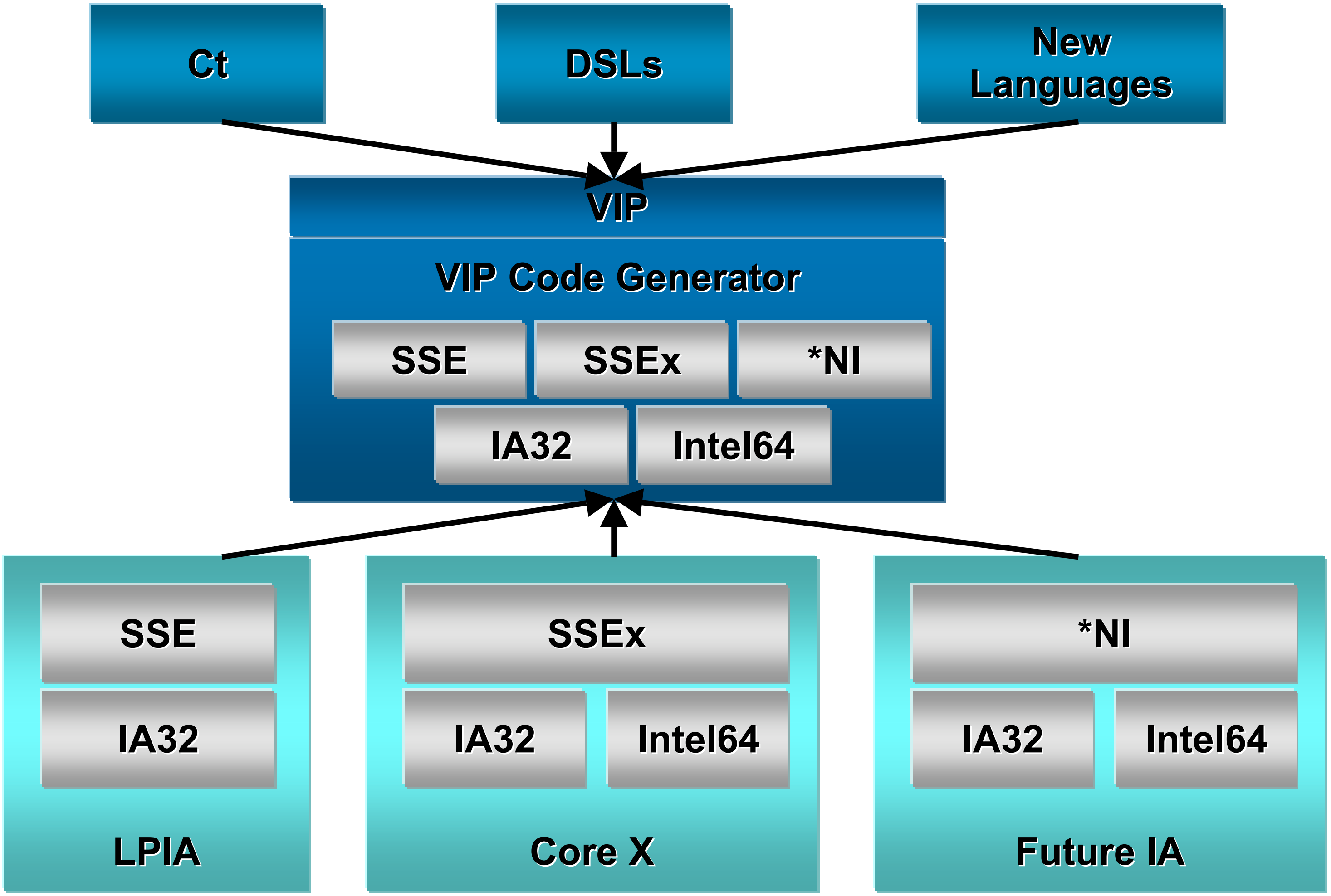


Low-Level Optimizer

- ~10 optimizations
- Eliminate redundant checks.
- Reorganize the data layout.
- Parallelize the data-parallel tasks on multi threads.
- SIMD-vectorize each thread.

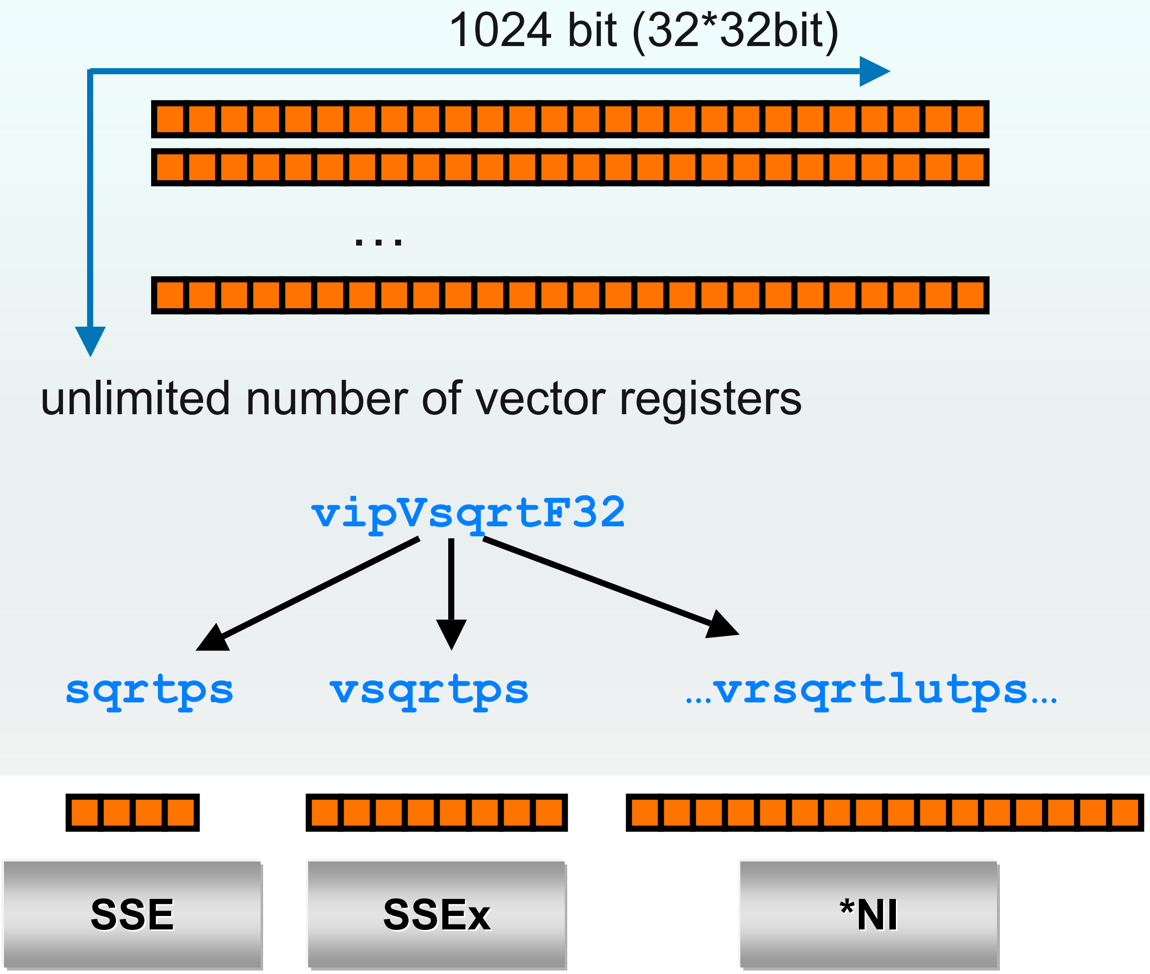


VIP (Virtual Intel Platform):
The ISA of Ct Virtual Machine



VIP = Virtualized Intel SIMD ISA + A Subset of X86

Virtualized Intel SIMD ISA



1024 bit (32*32bit)

…

unlimited number of vector registers

`vipVsqrtF32`

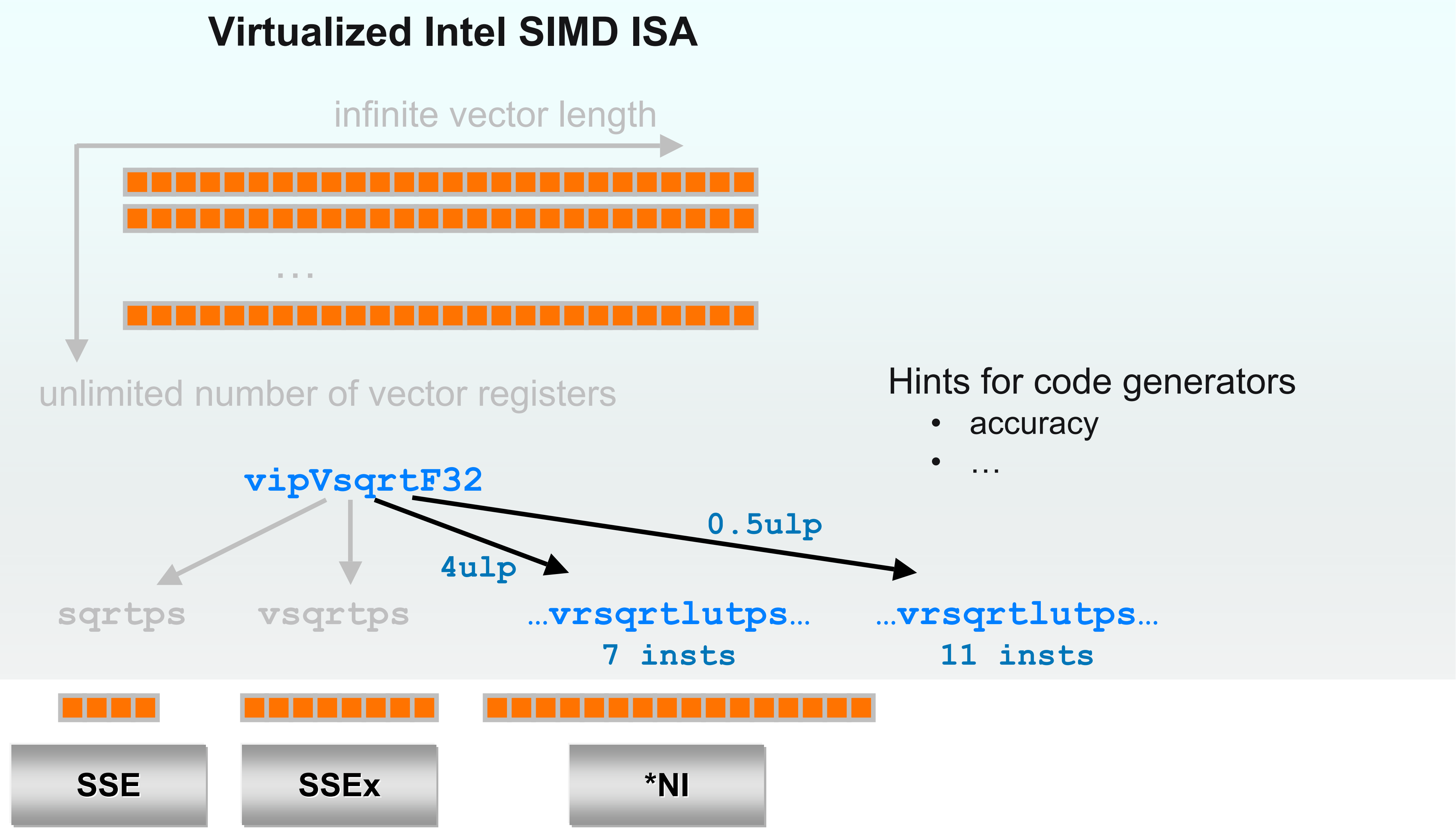
`sqrtps` `vsqrtps` `…vrsqrtlutps…`

SSE SSEx *NI

Virtualized vector instructions

- mask
- cast/conversion
- shuffle/swizzle
- gather/scather
- …

VIP = Virtualized Intel SIMD ISA + A Subset of X86



Virtualized Intel SIMD ISA

infinite vector length

. . .

unlimited number of vector registers

Hints for code generators

- accuracy
- …

vipVsqrtF32

4ulp

0.5ulp

sqrtps

vsqrtps

…vrsqrtlutps…
7 insts

…vrsqrtlutps…
11 insts

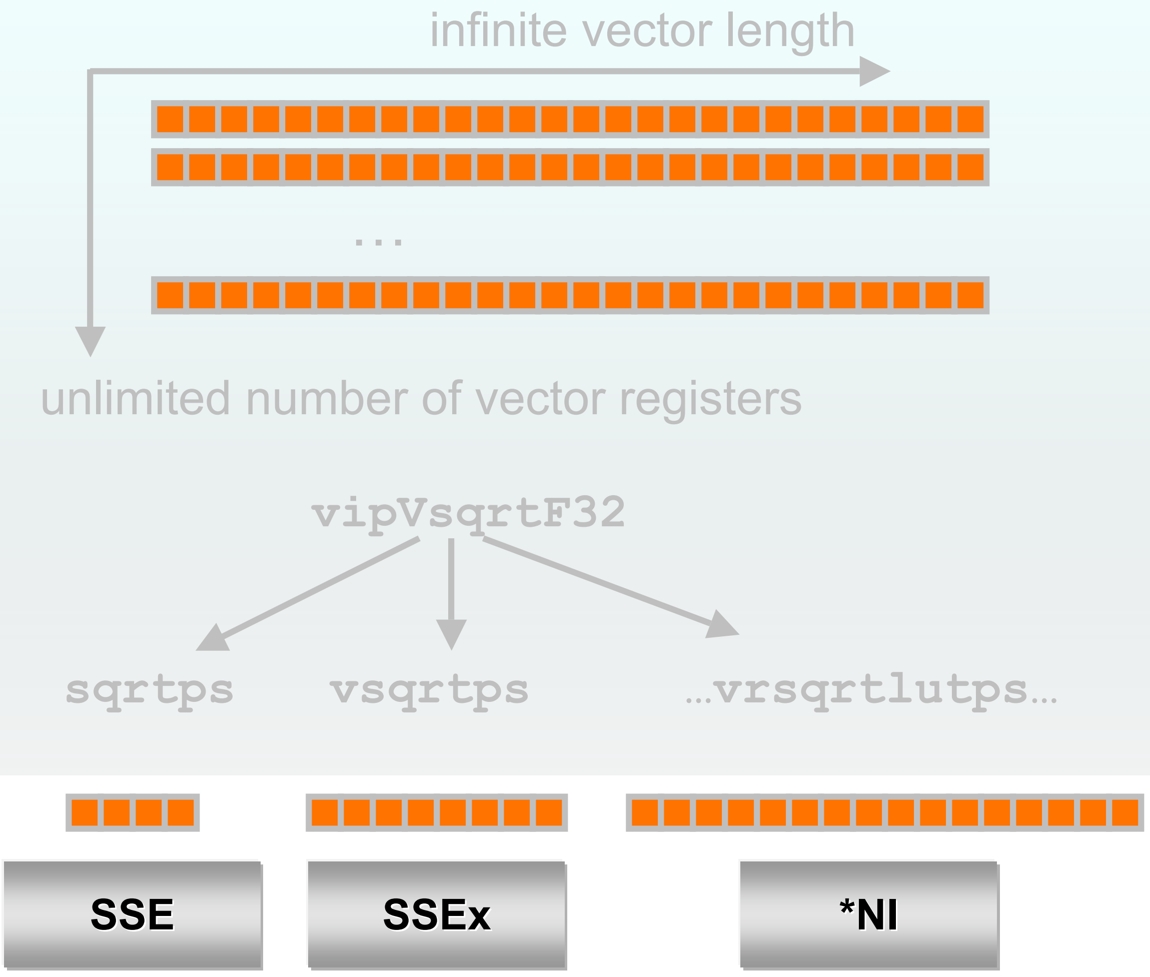
SSE

SSEx

*NI

VIP = Virtualized Intel SIMD ISA + A Subset of X86

Virtualized Intel SIMD ISA



A Subset of X86

Describe loop structures

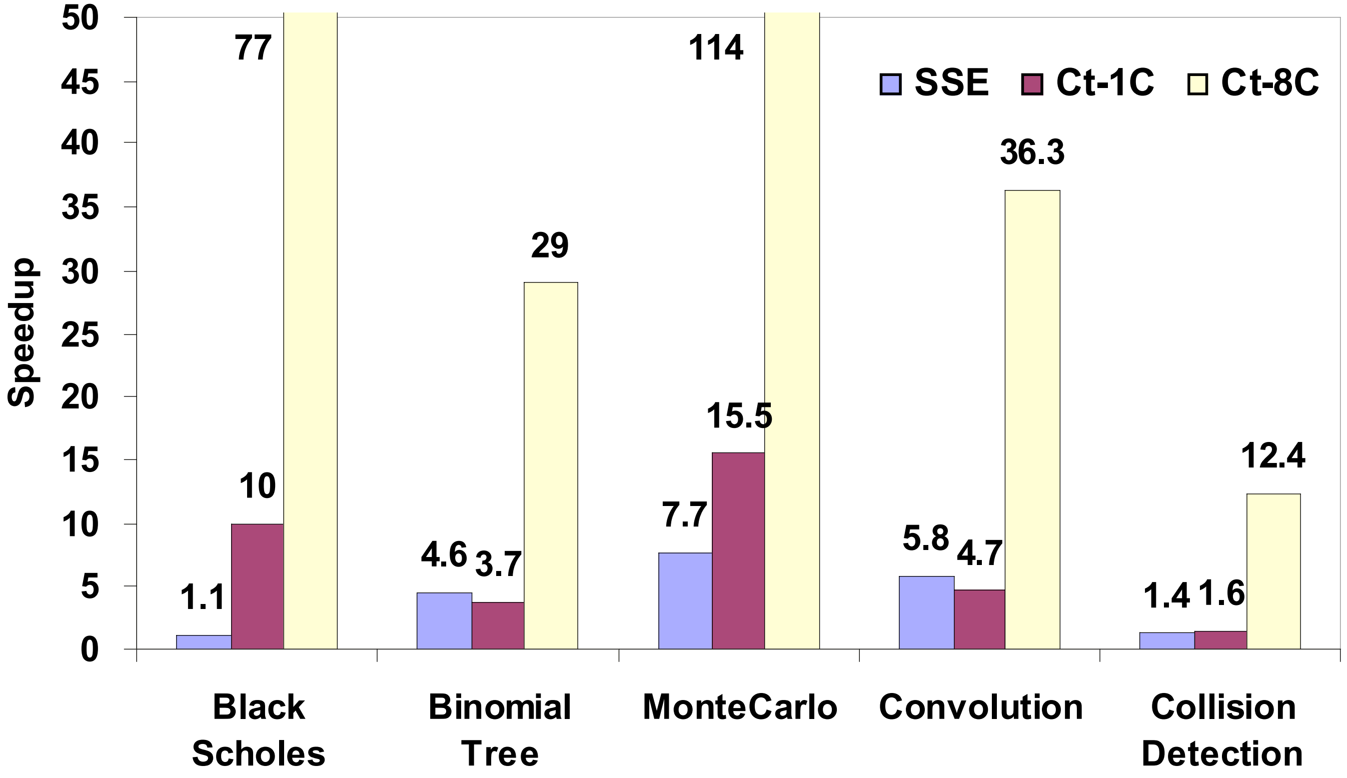
Deal with nested vectors

Perform optimizations

IA32

Intel64

Application Kernels & Performance



Approach 2: A Parallel Functional Language

- We are also working with an ISV on a compiler for a parallel functional language
  - Strict, but not call by value for implicit parallelism (maybe with lightweight annotations)
  - Arrays and array comprehensions for data parallelism
  - Effects system to contain impure features
  - Atomic for state updates

Who is Team CLARA @ Intel?

- We comprise a team of about 23 researchers in the US and China working on:
  - A bridging model: Ct
  - A parallel language implementation infrastructure: Pillar
  - A proposed long term solution: a new functional language that features implicit parallelism, dependent typing, and an effects type system
- We have diverse technical backgrounds
  - Java, C/C++, Fortran/F9x, Tcl/TK, SML compilers and runtimes; biotech, graphics, physics, image, and video applications
- We are interested in supporting work in the areas I've identified
  - E.g. we have been driving DAMP in its first two years (please attend DAMP next year :))

Summary

- Ct 1.0 source and examples will be "available" to collaborators and the curious under reasonable licensing (e.g. non-commercial use for academia) January 2008
- Next step: Full applications
- In the meantime, many-core architectures present a unique opportunity to language designers:
  - The combination of software engineering methodologies and requirements and parallel computing constraints seems tailor-made for the FP community

- Whitepaper, app notes at www.intel.com/go/terascale intel®

Leap ahead™