Ct: Channeling NeSL and SISAL in C++

Anwar Ghuloum
Corporate Technology Group

Agenda

- Entering the Many-core Era
- Data Parallelism
- Ct

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

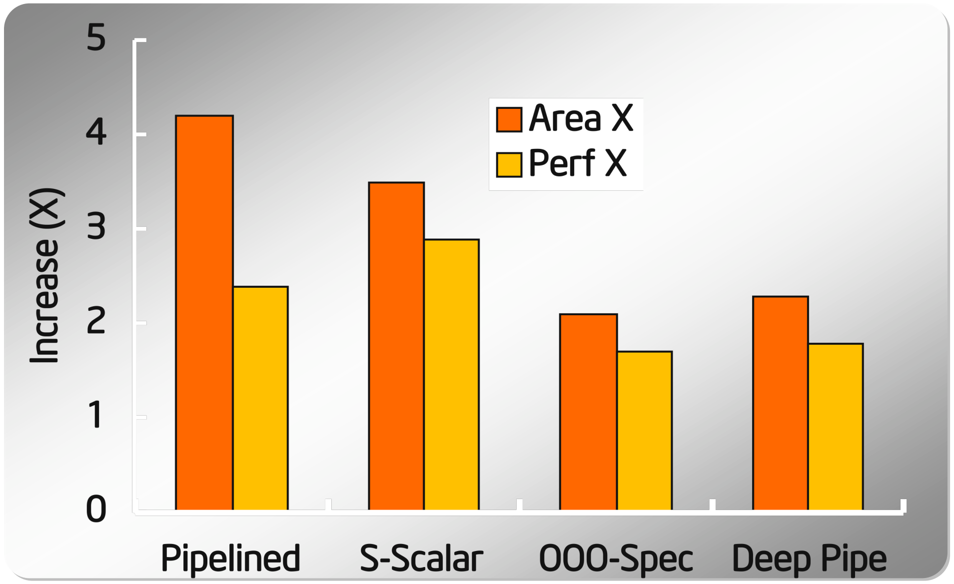
Power Scaling

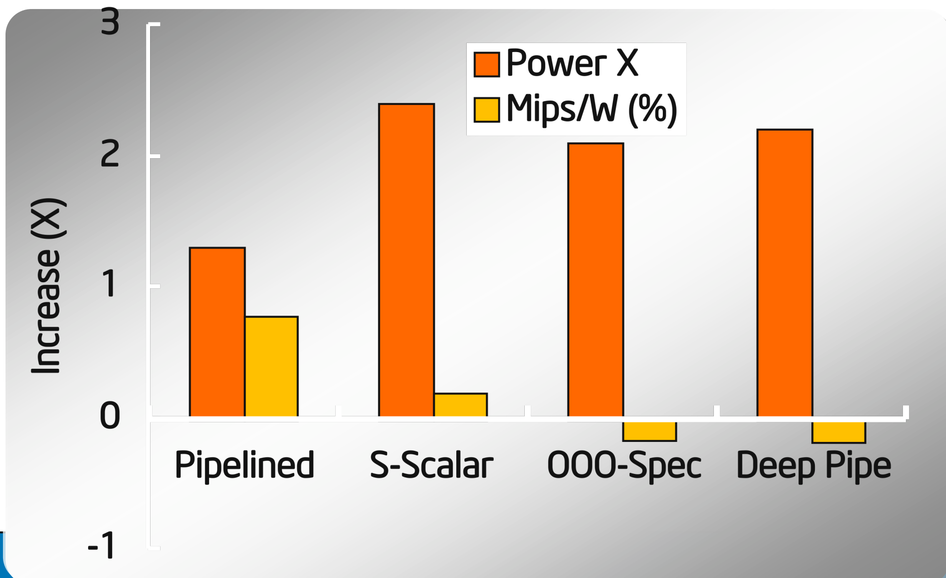
~= transistors * cap/trans * voltage^2 * 1/time

~= 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.

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

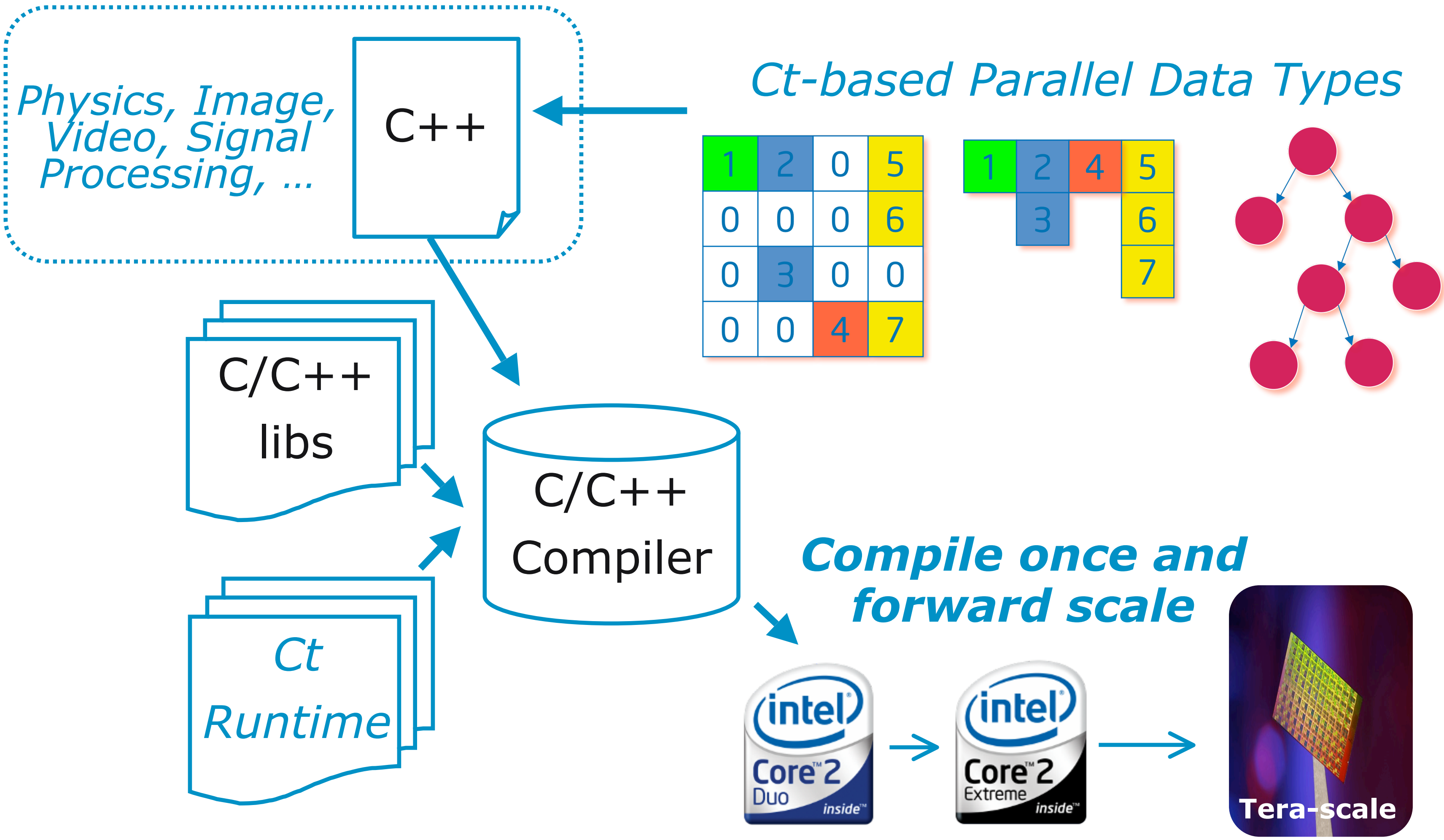
**Depends on which developer you ask.*

What Is Ct?

Extend C++ for Throughput-Oriented Computing

- Ct adds new data types (parallel vectors) & operators
  - Library-like interface and is fully ANSI-compliant
- Ct abstracts away architectural details
  - Vector ISA width / Core count / Memory model
- Nested data parallelism and deterministic task parallelism differentiates Ct on parallelizing irregular data and algorithms
- Ct platform-level API, Virtual Intel Platform (VIP), is designed to be retargetable to SSE, SSEx, *NI

Ct: Nested Data Parallelism in C/C++



Ct Technical Vision

Make parallel programming easier through:

- Fully leverage *deterministic* parallel programming models
  - I.e. Make data races impossible
- Express complex behaviors through simple operators
- Present a simple and predictable performance model
- **Provide a forward-scaling programming model**
  - **"Future-proof"**

So, 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

The Data Parallel Model

Parallel operations across a collection of data elements
… But, allows programmer to think "serially".

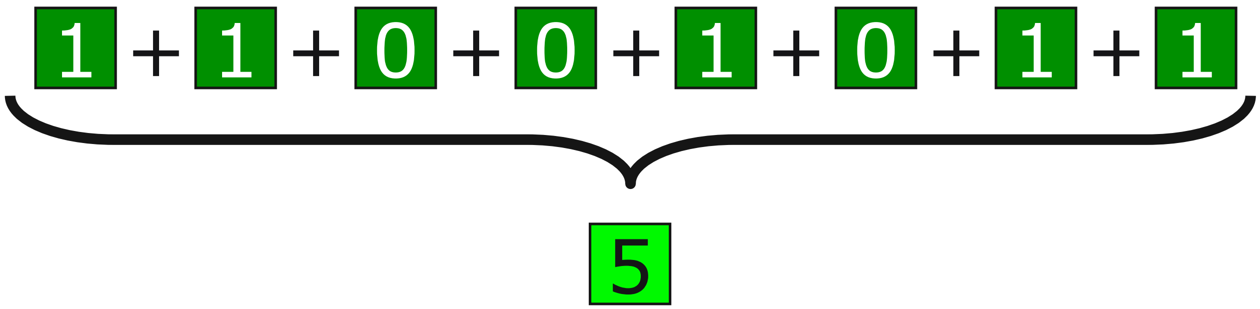
- Compiler + runtime map automatically onto parallel HW
- Widely useful in emerging Tera-scale "killer apps"

Example

Write a program that sums a vector's elements

Sum of 8 element vector

- For “short” vectors, serially add elements



Sum of 32 element vector

- For "medium" vectors, first use SIMD to generate 16 element partial

1	1	0	1	0	0	0	1	0	1	0	0	0	0	1	1

+

1	1	0	1	0	0	0	1	0	1	0	0	0	0	1	1

=

2	2	0	2	0	0	0	2	0	2	0	0	0	0	2	2

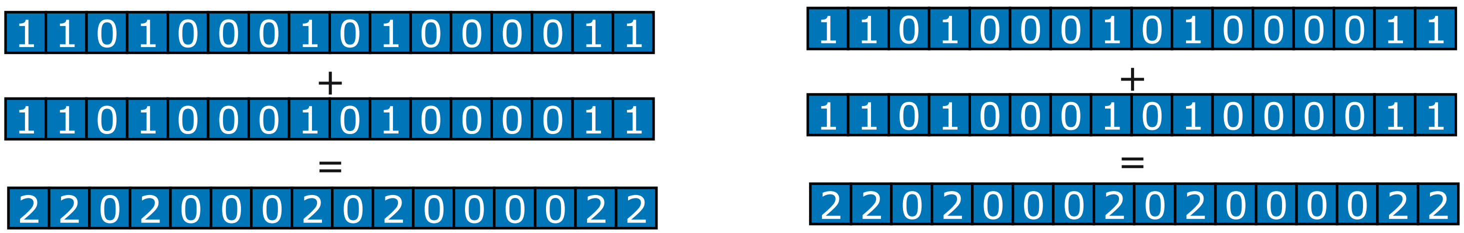
- Then serially add partial sum

2 + 2 + 0 + 2 + 0 + 0 + . . . 2

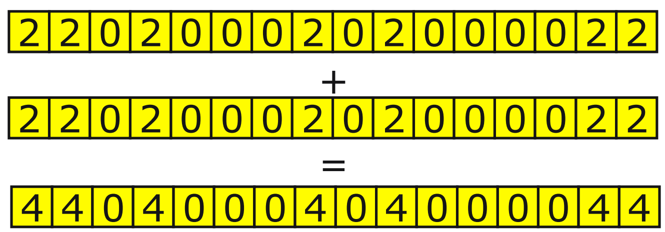
14

Sum of 8000 element vector

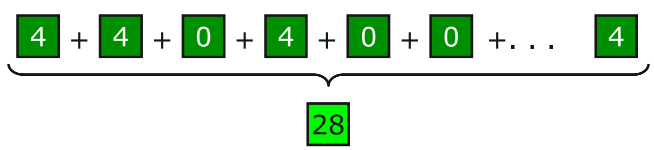
- For “long” vectors, break vector into 2 pieces and use SIMD hardware on 2 cores



- Have a thread synchronization on each partial result, then have 1 core add the SIMD result of other core



- Then have core serially add 16 elements



Data Parallel Programming Simplifies The Choices

Choosing the optimal algorithm is hard – it is a function of

- Low level hardware details
- Thread synchronization costs
- Number of cores (multiple hardware configurations)
- Vector length (not always known at compile time)

Data parallel models hide these choices for the programmer

What is “Nested” Data Parallelism?

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

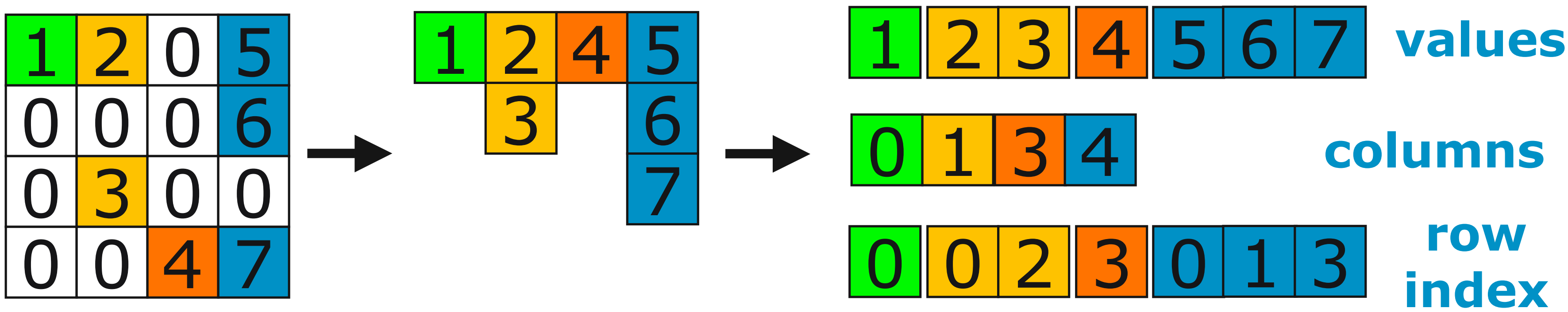
- Flat (or limited dimensionality) vectors
- Operators over vectors
  > Element-wise operators
  > Limited collective communication operations (reductions)
  > Some constrained permutation
  > Masking operations

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

Irregular Data Structures



4x4 sparse matrix

4 element vector of variable length vectors

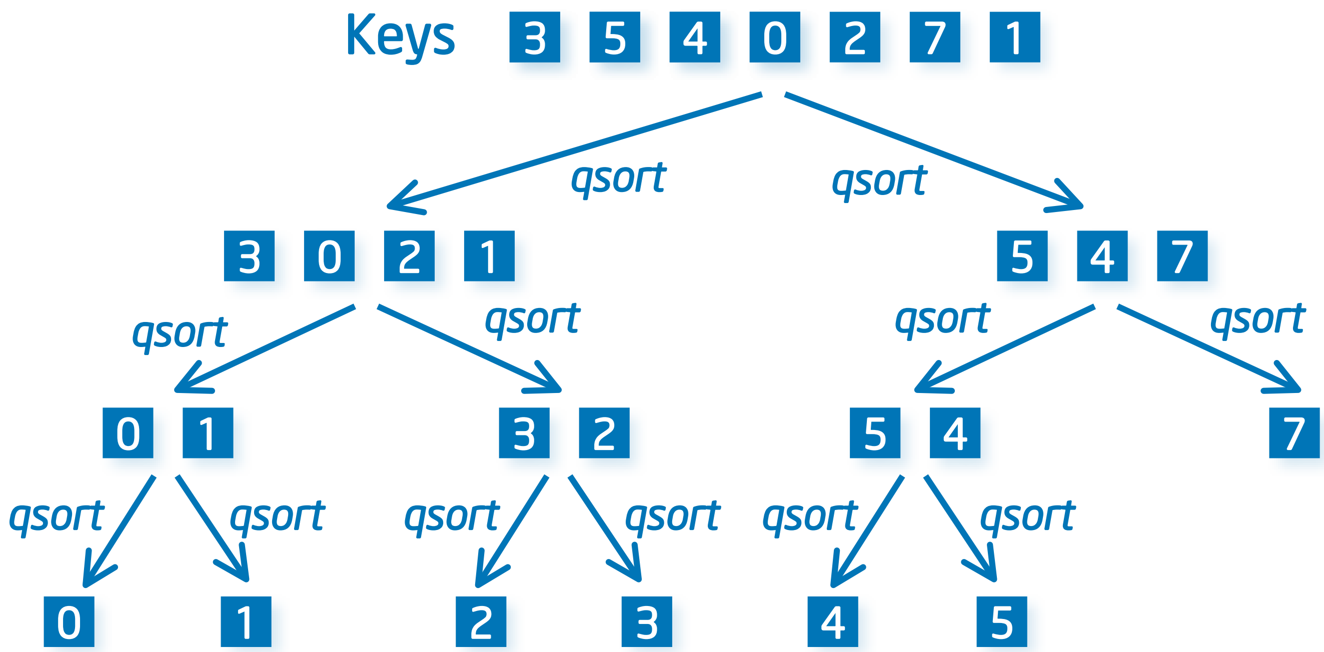
Flattened representation with column & row metadata

- A classic example: Sparse matrices
  - Common in RMS applications
  - Difficult for a programmer to deal with

Nested data parallelism handles irregular structures automatically

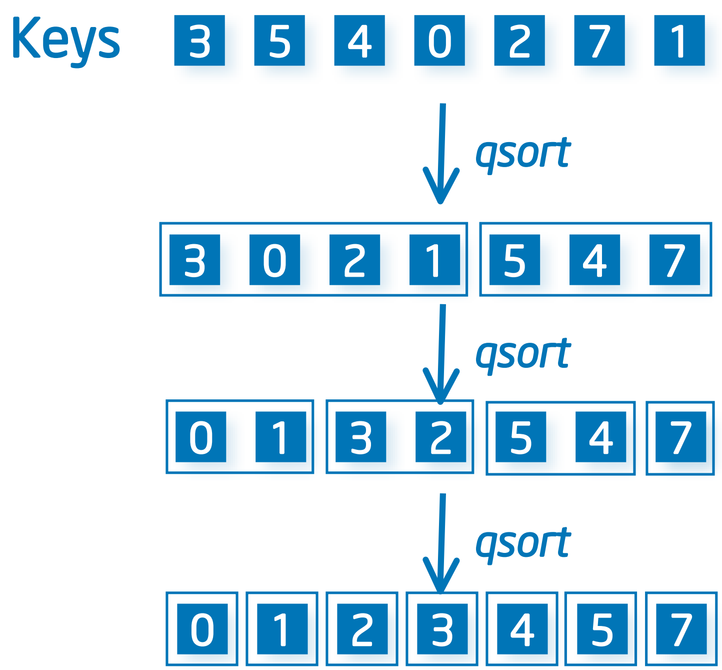
Nested Data Parallelism: Quicksort

Classic quicksort: Increasing task parallelism & decreasing data parallelism as we recurse



Nested Data Parallelism: Quicksort

Nested data parallel quicksort: unifies irregular divide-and-conquer parallelism with data parallelism



Design Constraints

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

- These are and will continue to be the dominant languages for high performance for the next 5+ years
  - Java/C# do not solve many of the real "problems" associated with parallel programming

…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

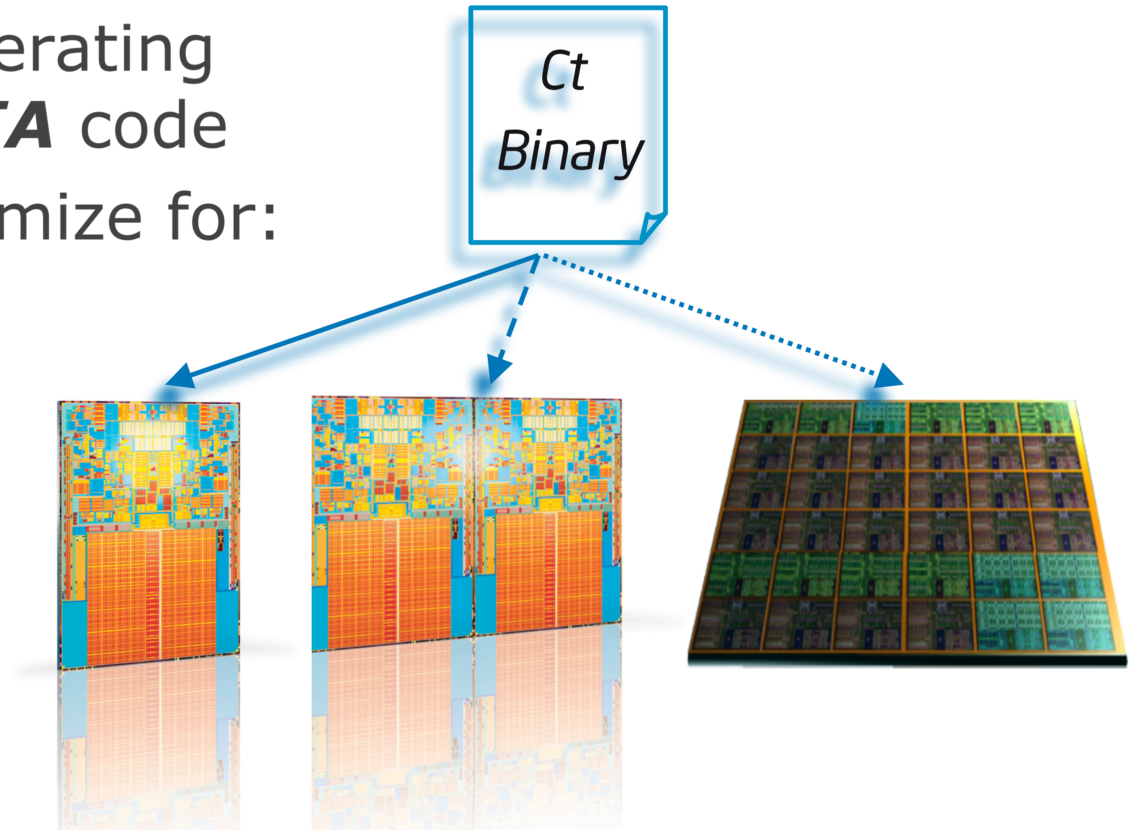
- Must co-exist with legacy APIs, libraries
- Must co-exist with prevailing parallelism APIs (Pthreads, winthreads, OpenMP, MPI)

Ct is….

- …an “extension” of C++ for throughput computing using Nested Data Parallelism (and deterministic task parallelism)
- …like a library implementation of a STL-style container
- …using a (dynamically linked) runtime to optimize and generate code
- …designed to *forward-scale* software

Forward Scaling with Ct

- Compile *once*, generating optimized, native ***IA*** code
- Dynamically reoptimize for:
  - More cores
  - More cache
  - More bandwidth
  - More instruction set enhancements



Ct forward scales software with Moore's law in the age of Tera-scale

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 simply references to immutable values

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

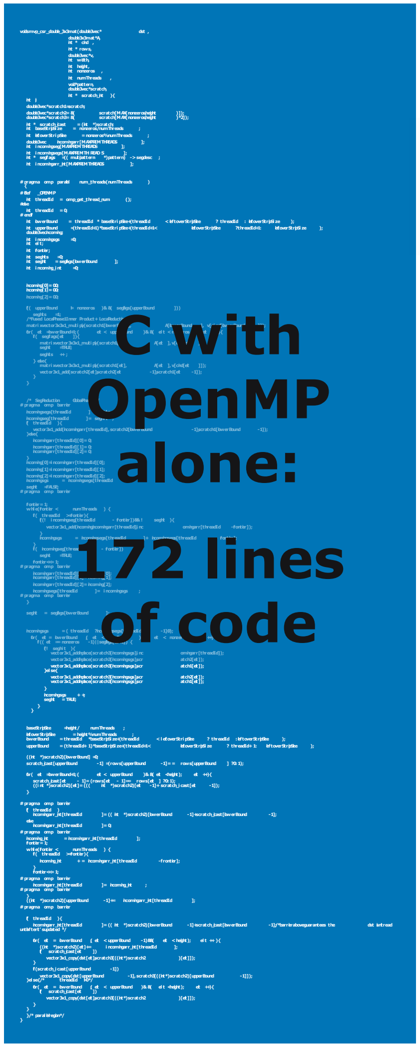
Assigning a value to DoubleVec doesn't modify the value representing the result of the add, it simply refers to a *new* value.

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 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



Ct: <6 lines of code, faster, scalable

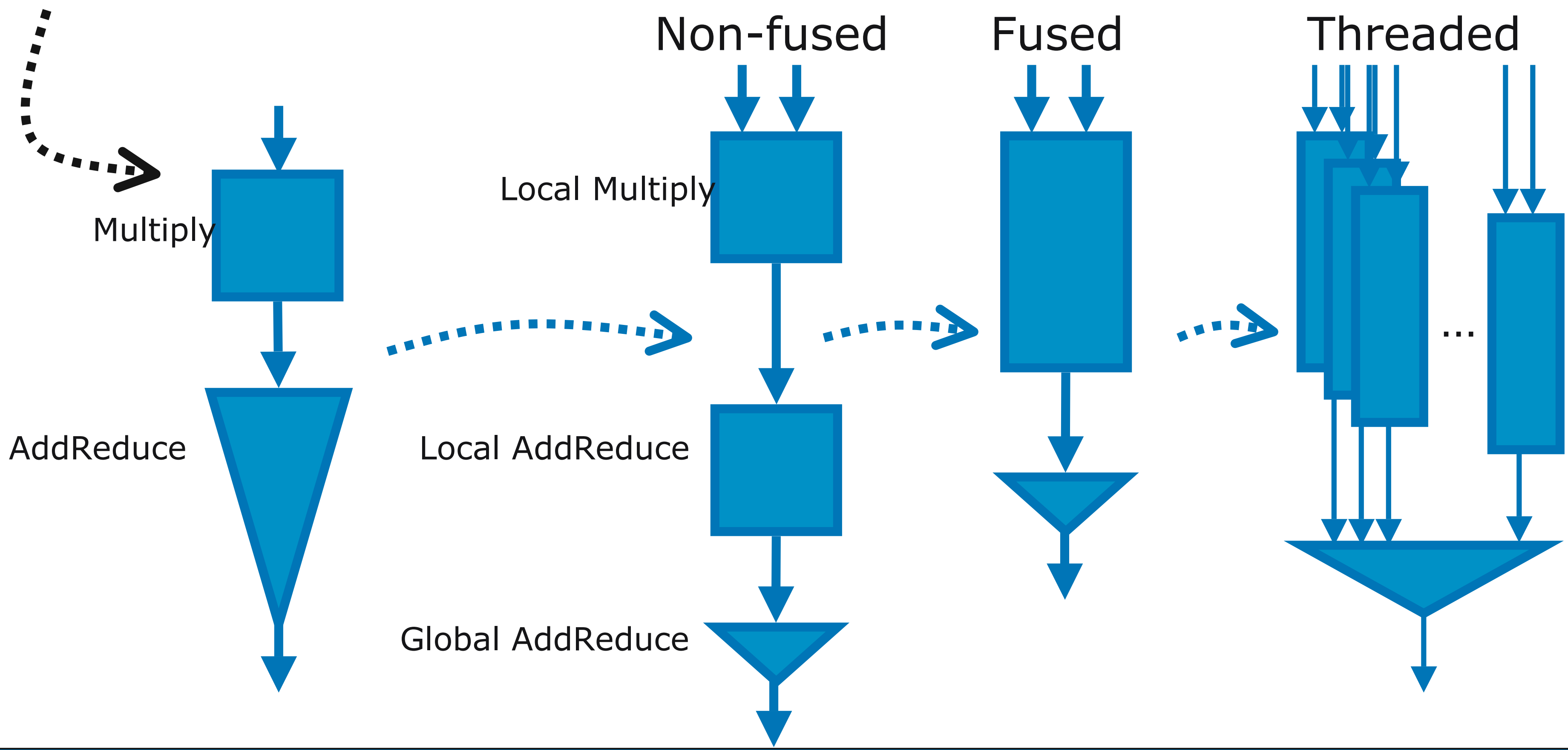
```
TVEC<F64> smvpCSC(TVEC<F64> A, TVEC<I32> rind,
                  TVEC<I32> cols, TVEC<F64> v) {
    TVEC<F64> expv, product, result;
    expv = v.distribute(cols);
    product = A*expv;
    product = product.applyNesting(rind,
ctSparse);
    return product.reduceSum();
}
```

- 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

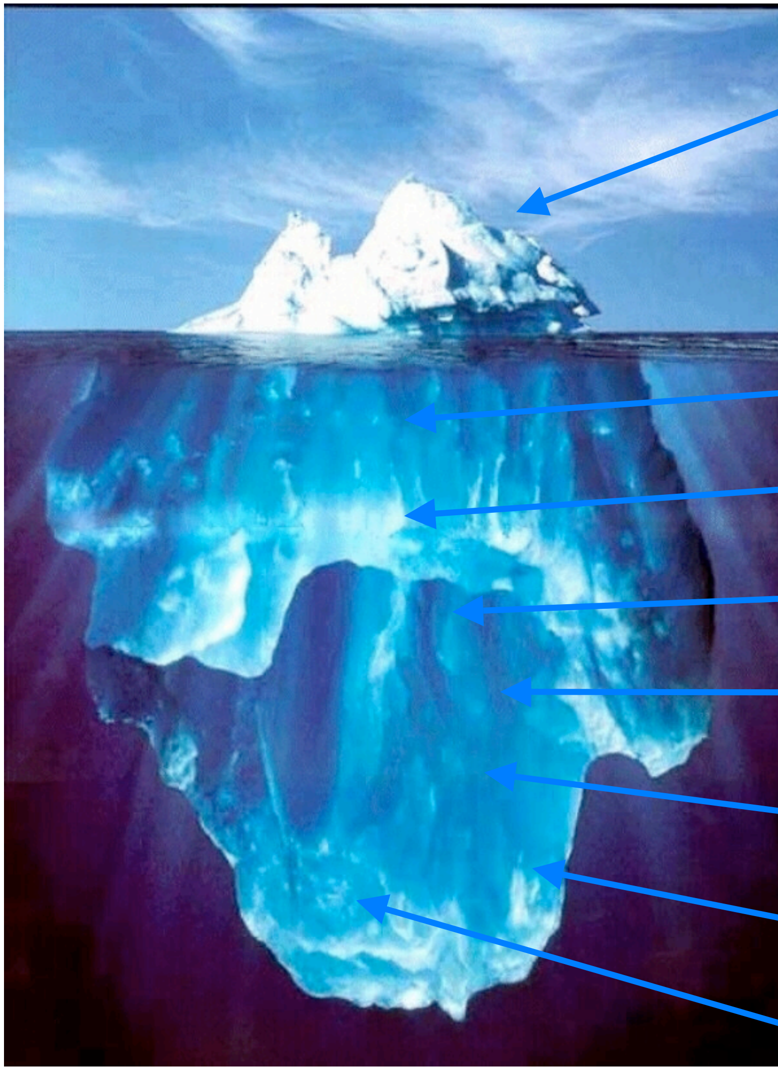
Ct's Threading Model: *Incrementally evaluated, fine-grained dataflow*

```
product = A*expv
SMVP = addReduce(product);
```

"Static" or Compile-Time *Dynamic*



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

Grand Vision

Make parallel programming easier through:

- Pushing limits of *deterministic* parallel programming models
  - I.e. Data races not possible
  - We already know we can take this to tasks
- Expressing complex behaviors through simple operators
- Presenting simple and predictable performance models
- Provide a forward-scaling (I.e. “future-proof”) programming model

Ct Adoption Paths for Developers

In order of increasing effort and payoff:

- Use Ct-enabled libraries (e.g. Blas, Physics, etc.) in place of existing
- Rewrite “leaves” (or kernels) of code in Ct
- Rewrite application to use Ct pervasively

The goal is to support all models at (at least) good performance levels.

Ct In Action: C User Migration Path

```
float s[N], x[N], r[N], v[N], t[N];
float result[N];

for(int i = 0; i < N; i++) {
  float d1 = s[i] / ln(x[i]);
  d1 += (r[i] + v[i] * v[i] * 0.5f) * t[i];
  d1 /= sqrt(t[i]);
  float d2 = d1 - sqrt(t[i]);

  result[i] = x[i] * exp(r[i] * t[i]) *
    ( 1.0f - CND(d2)) + (-s[i]) * (1.0f - CND(d1));
}
```

1
```
#include <ct.h>
```

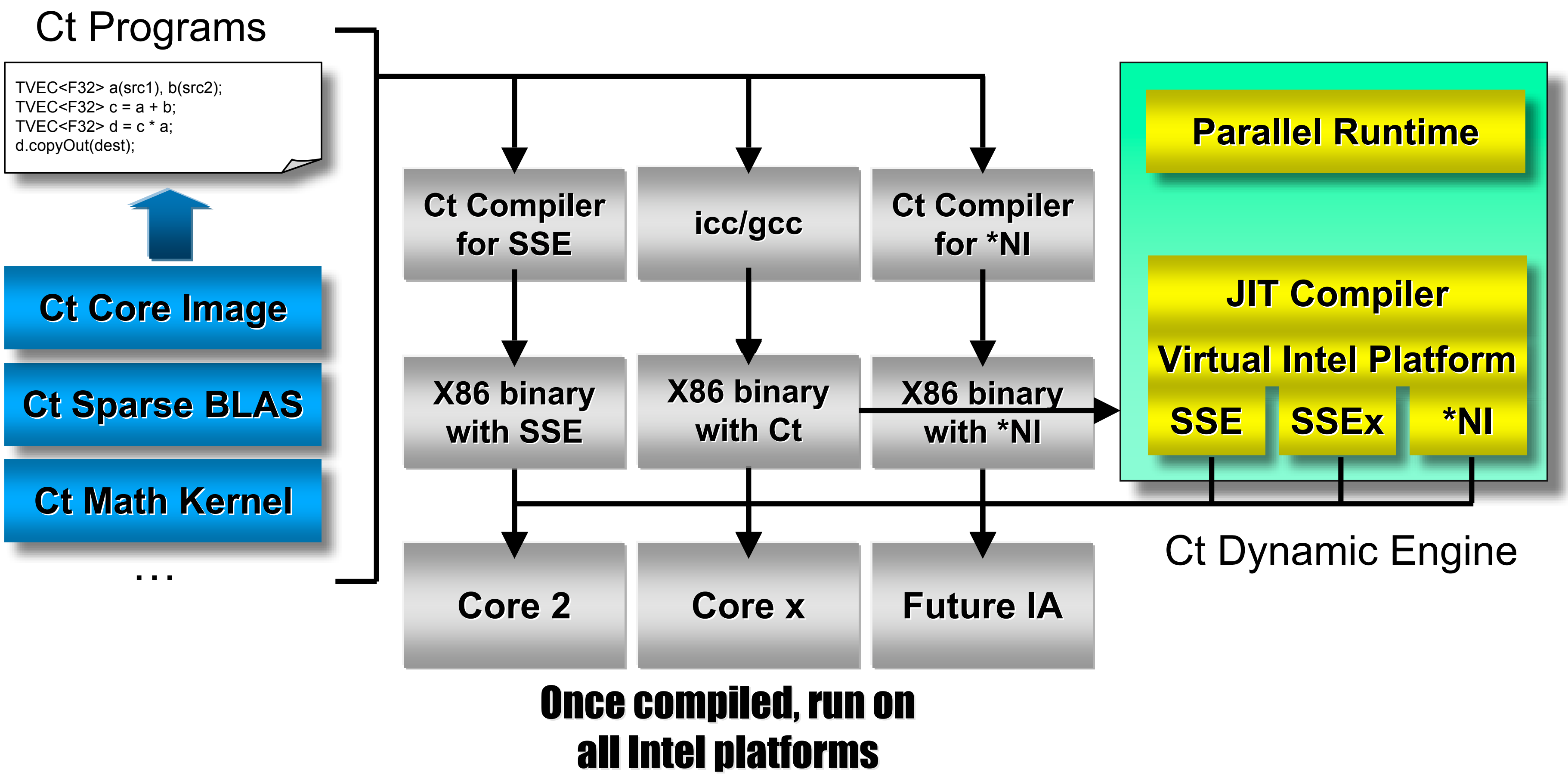
2
```
T s[N], x[N], r[N], v[N], t[N];
T result[N];
TVEC<T> S(s, N), X(x, N), R(r, N), V(v, N), T(t, N);
```

3, 4
```
TVEC<T> d1 = S / ln(X);
d1 += (R + V * V * 0.5f) * T;
d1 /= sqrt(T);
TVEC<T> d2 = d1 - sqrt(T);

TVEC<T> tmp = X * exp(R * T) *
  ( 1.0f - CND(d2)) + (-S) * (1.0f - CND(d1));
```

5
```
tmp.copyOut(result, N);
```

Use Animation

Ct: Supporting All Intel Platforms

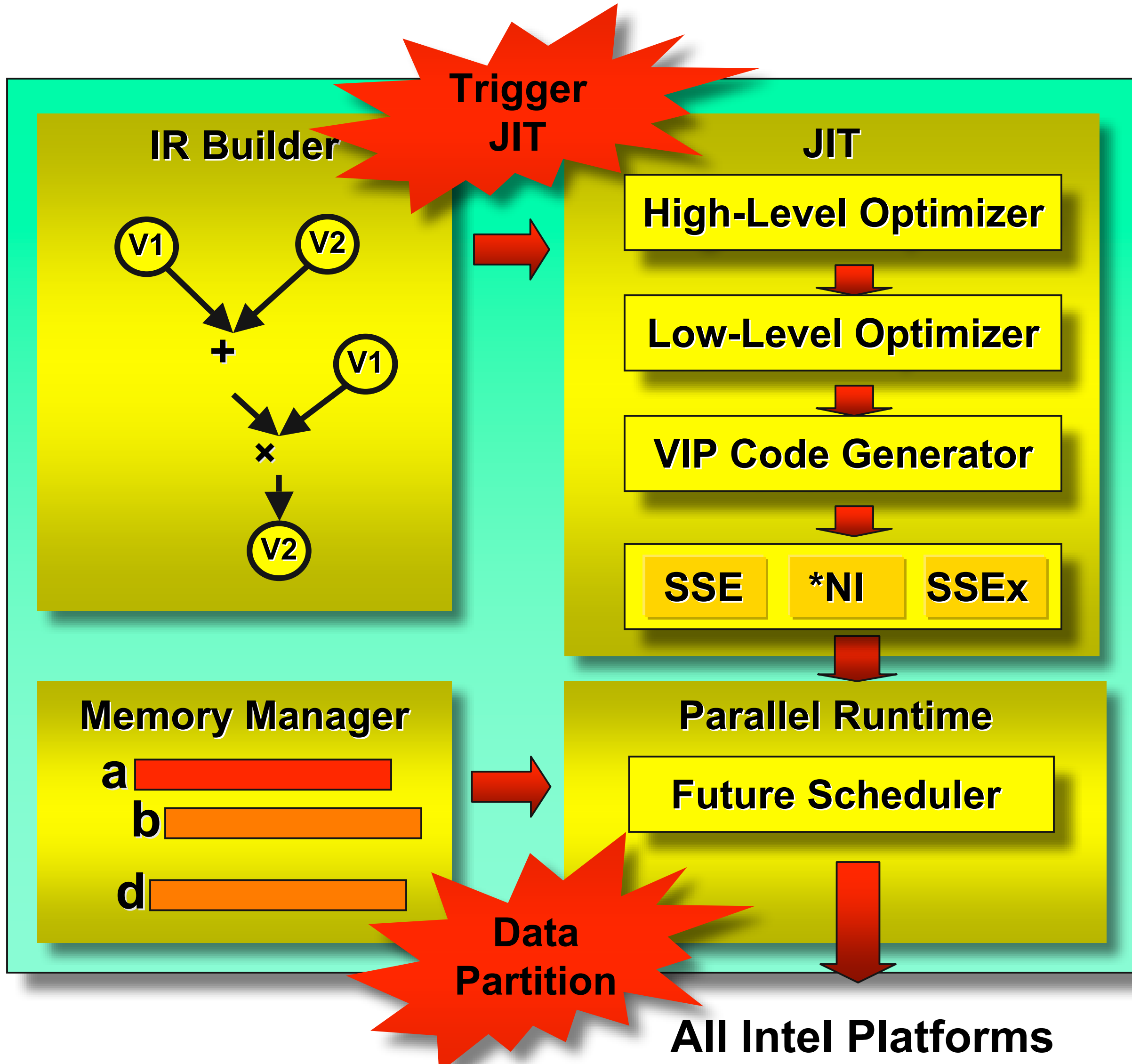


Use Animation

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);
```

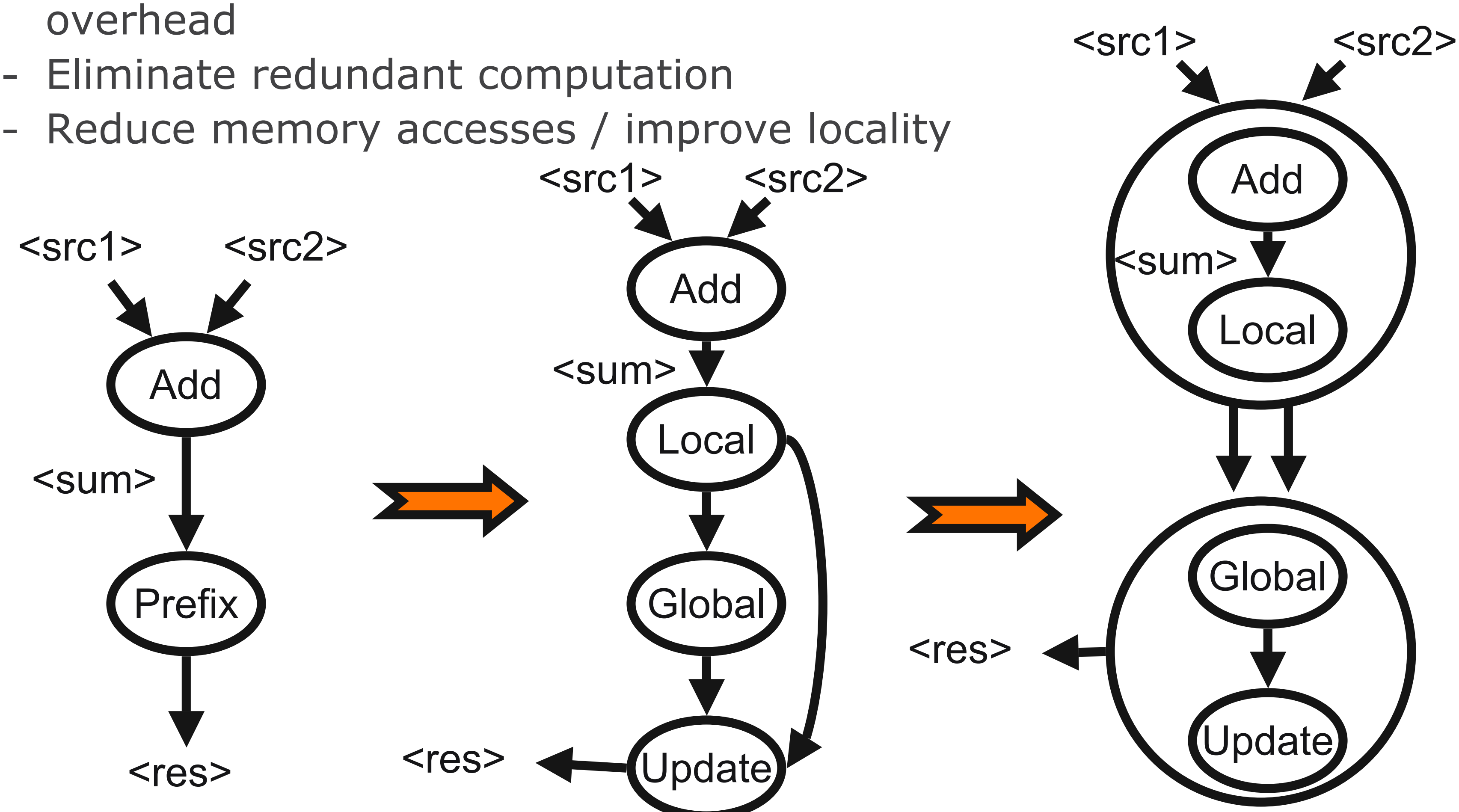
Ct Dynamic Engine



Use Animation

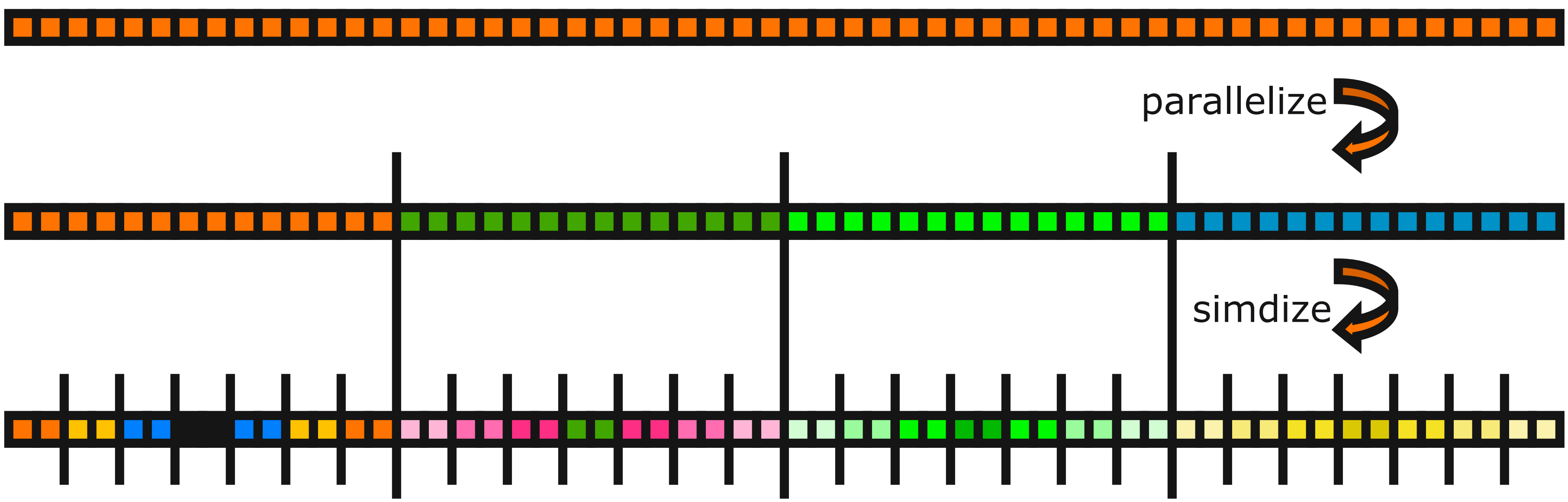
High-Level Optimizer

- ~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.



Black-Scholes: Ct vs. SSE

```
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_price,
              int num_options,
              T *stkprice,
              T *strike
               ate
              T *vol
              T *time
{
      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

42 lines ➔

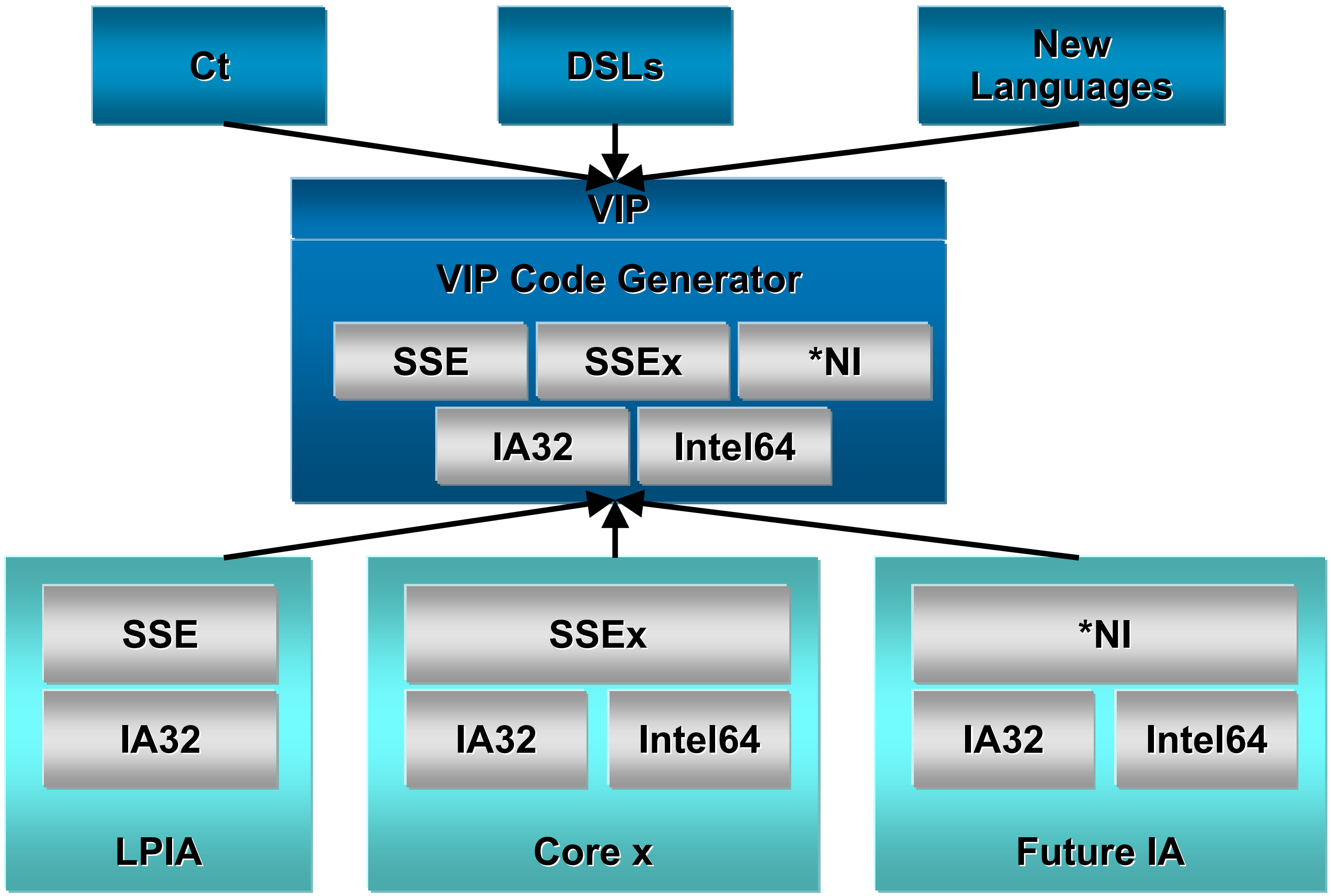
Programmability,
Forward scalability

```
#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) {
         Inp
         si
      } el
         s     0;
   }

   xInput = _MM_

   // Co      NPrimeX ter     mon     h four & six d
accuracy
   for (i=0                  ) {
      expValue              0.5f * InputX[i] * Inp
      // 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)0.319381530));
   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]);
   }

   xLogTerm = _MM_LOAD(&(logValues[0]));

   xPowerTerm = _MM_MUL(xVolatility, xVolatility);
   xPowerTerm = _MM_MUL(xPowerTerm, _MM_SET(0.5));
   // xPowerTerm = _mm_div_pd(xPowerTerm, _MM_SET(2.0,
2.0));

   xD1 = _MM_ADD(xRiskFreeRate, xPowerTerm);
   xD2 = _MM_SUB(xRiskFreeRate, xPowerTerm);

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

                   1, xLogTerm);
                   2, xLogTerm);

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

                   , 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);
   }
}
```

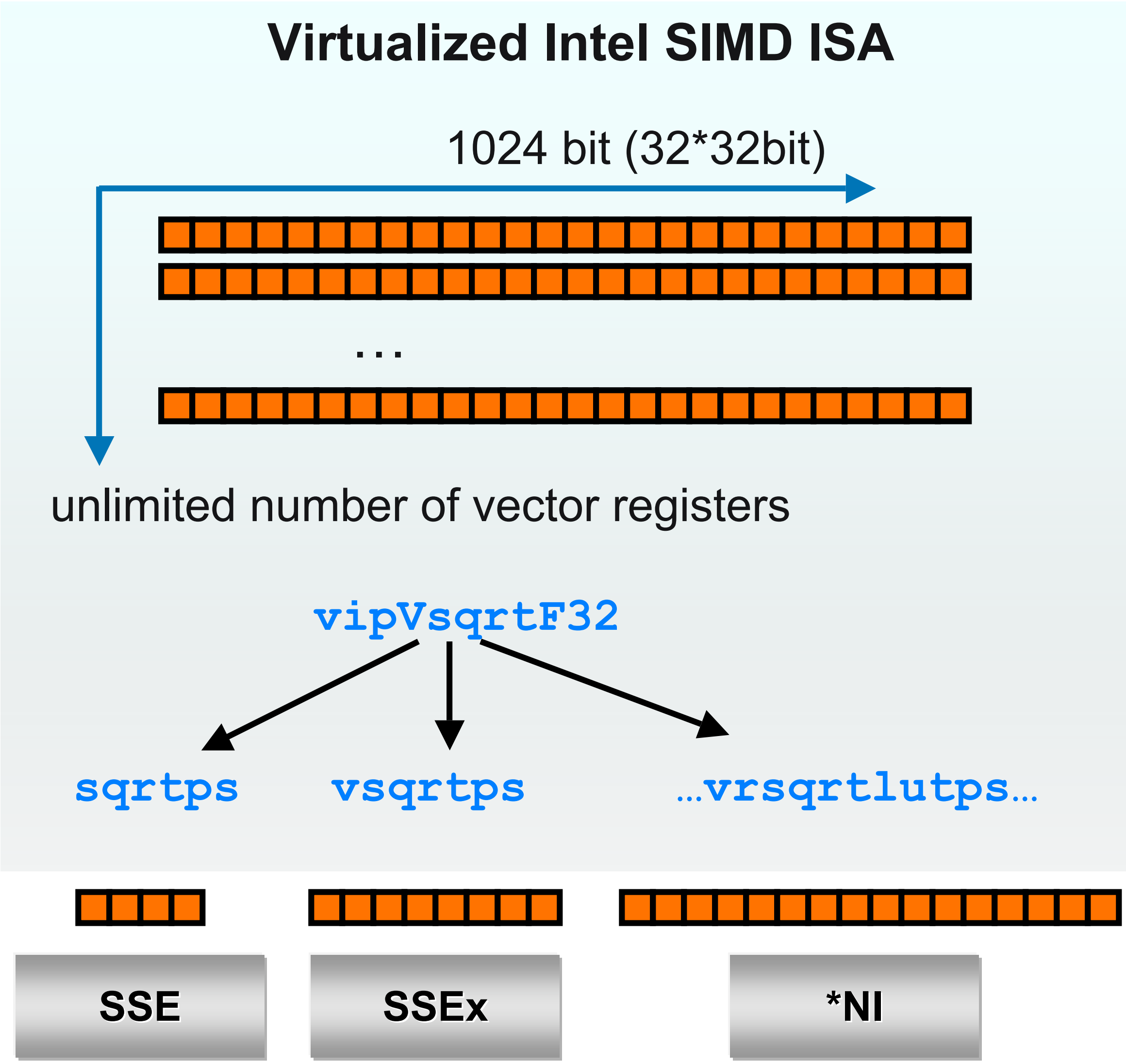
SSE

186 lines (single threaded)

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



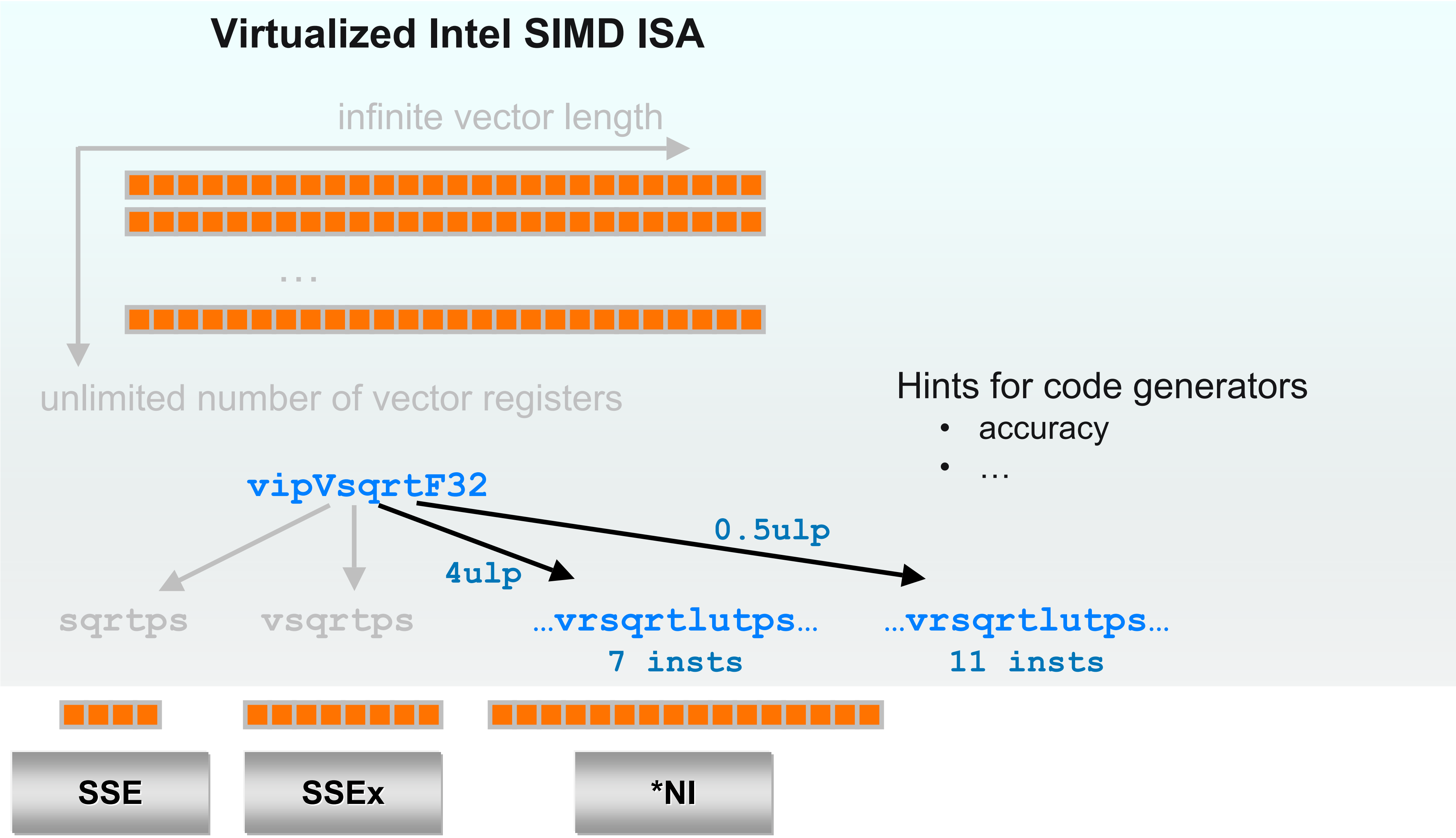
VIP = Virtualized Intel SIMD ISA + A Subset of X86



Virtualized vector instructions

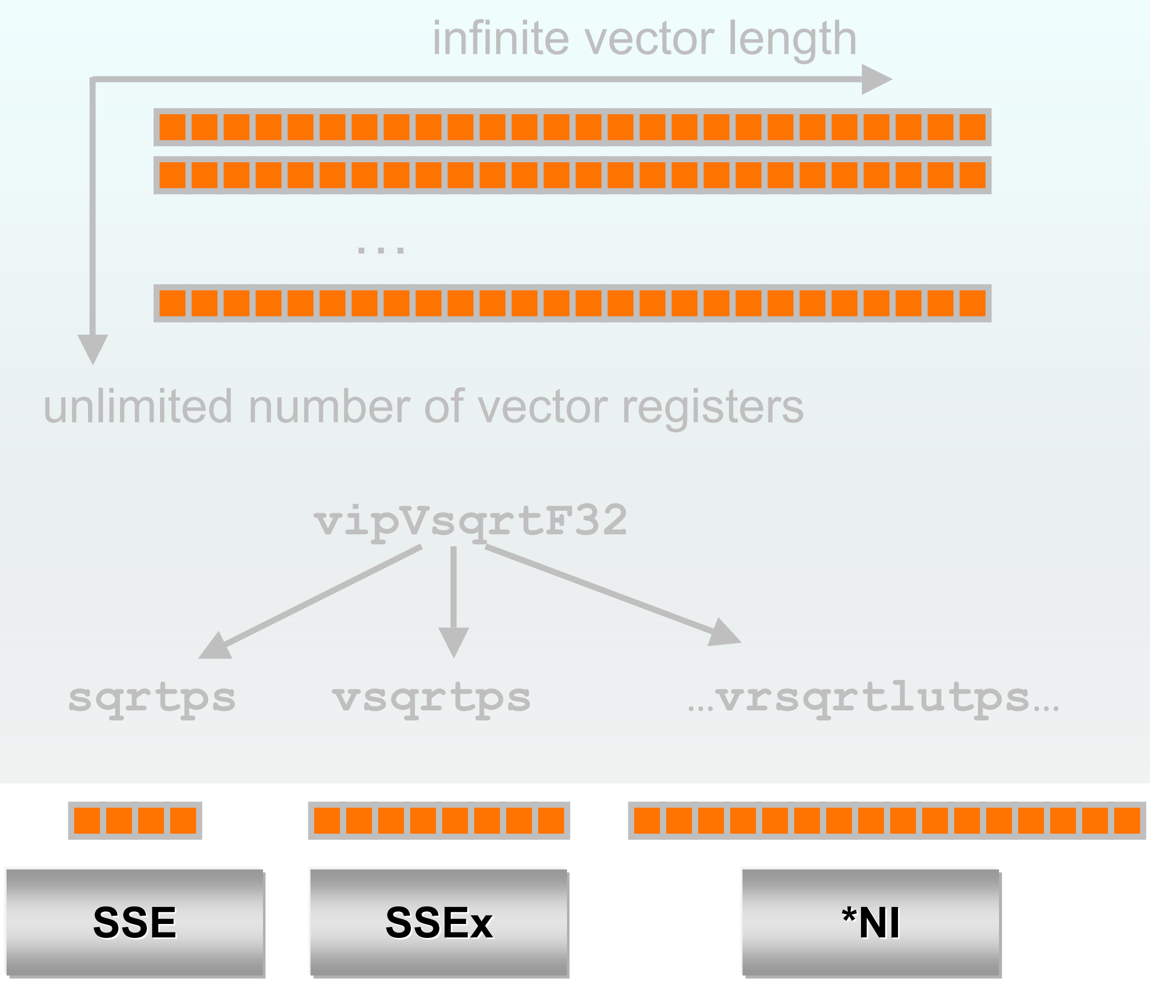
- mask
- cast/conversion
- shuffle/swizzle
- gather/scather
- …

VIP = Virtualized Intel SIMD ISA + A Subset of X86



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**

Ct Threading Model

What we needed:

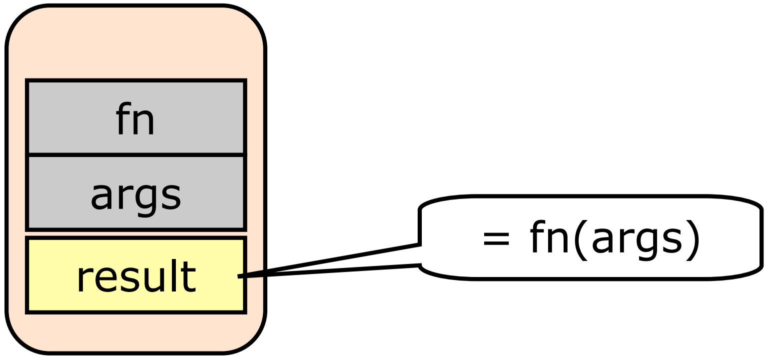
- Fine-grained concurrency and synchronization support
- Novel optimizations and usage patterns

What we came up with:

- New primitives for constructing parallel programs
  - >BulkSpawns - data-parallel computations
  - >SyncJoins - synchronization patterns
  - >FutureGraphs - collections of bulkSpawns and syncJoins

Feather-weight Threads: Futures

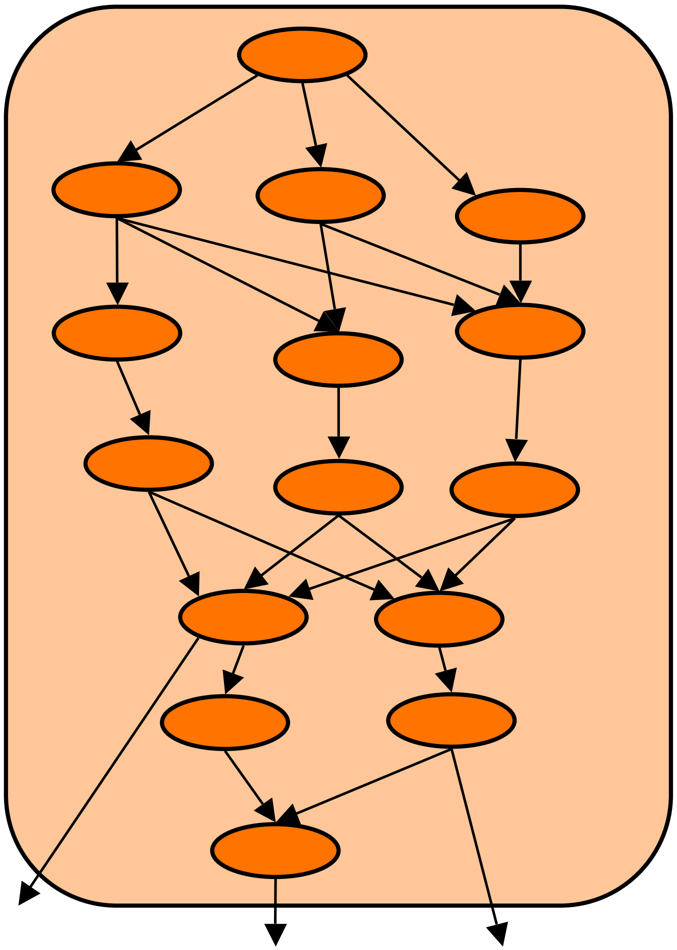
- (Almost) stateless task



Internals

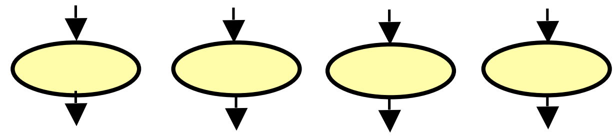
- API: Spawn & Read
- Task-queue based usage model
  - > Enqueued futures serviced by underlying worker(McRT)threads
- Futures about 2 orders of magnitude cheaper than threads

Complexity through Data-parallel Future Patterns

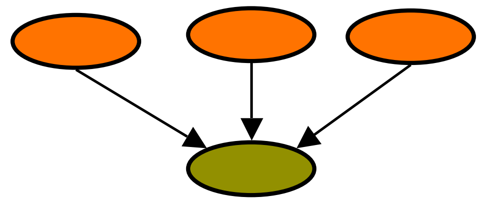


Element-wise operations

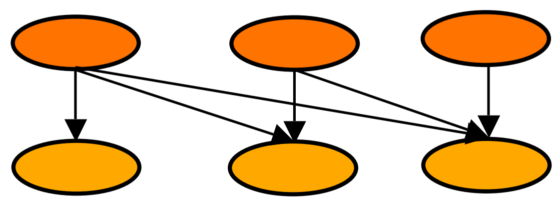
e.g. A[] = B[]+C[]



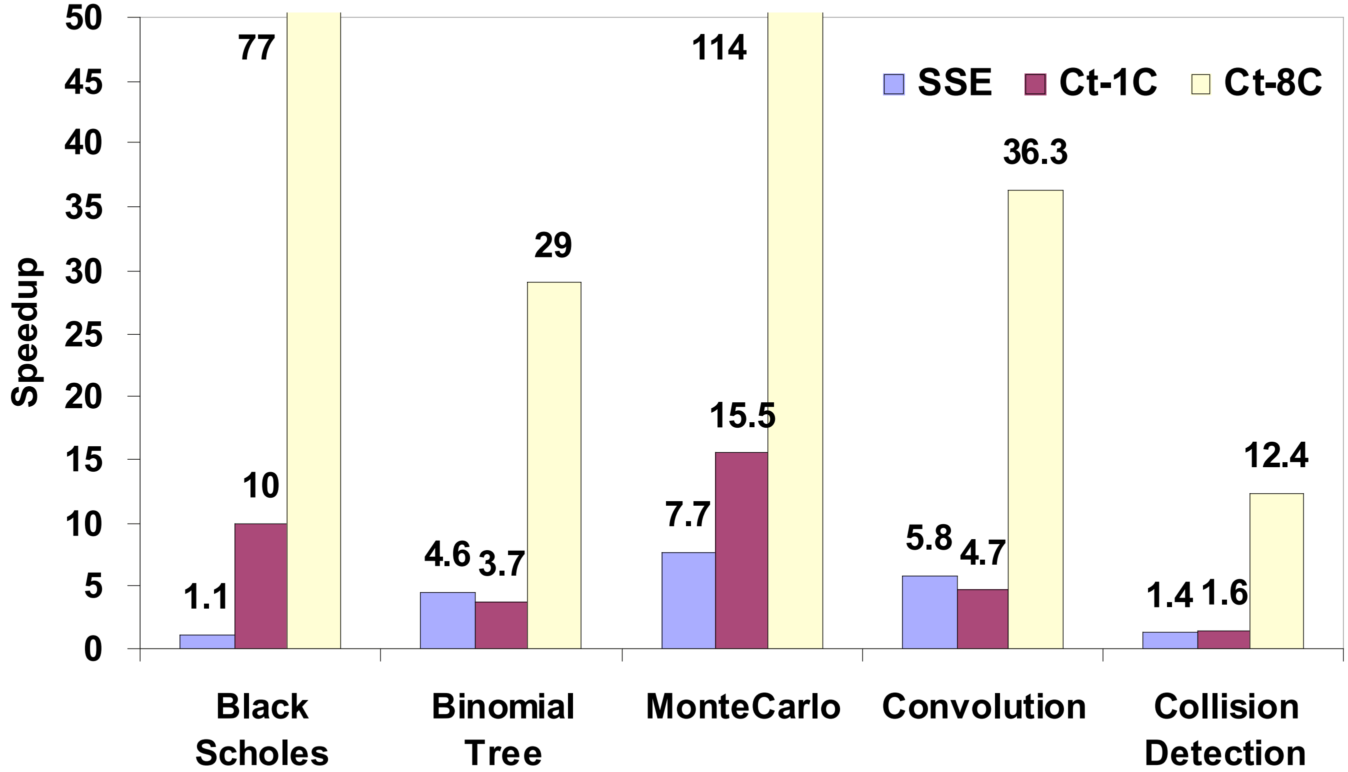
Reduction



Prefix



Application Kernels & Performance



Ct Workloads: What We Have

- Image Processing
- Signal Processing
- Seismic/Geophysics
- Gaming
- Dense and Sparse Linear Algebra
- Financial Analytics
- 13 Dwarves (At least 8 are straightforward)
- 15 of 26 MCBench workloads already covered

Ct Workloads: Next Steps

- Image Processing
- Signal Processing
- Seismic/Geophysics
- Gaming
- Dense and Sparse Linear Algebra
- More Financial Analytics - QuantLib
- More Dwarves
- More MCBench
- Crypto
- Astrophysics
- Teraspec

intel®

Leap ahead™