Cross-platform Software Optimization with Intel's Ct Technology

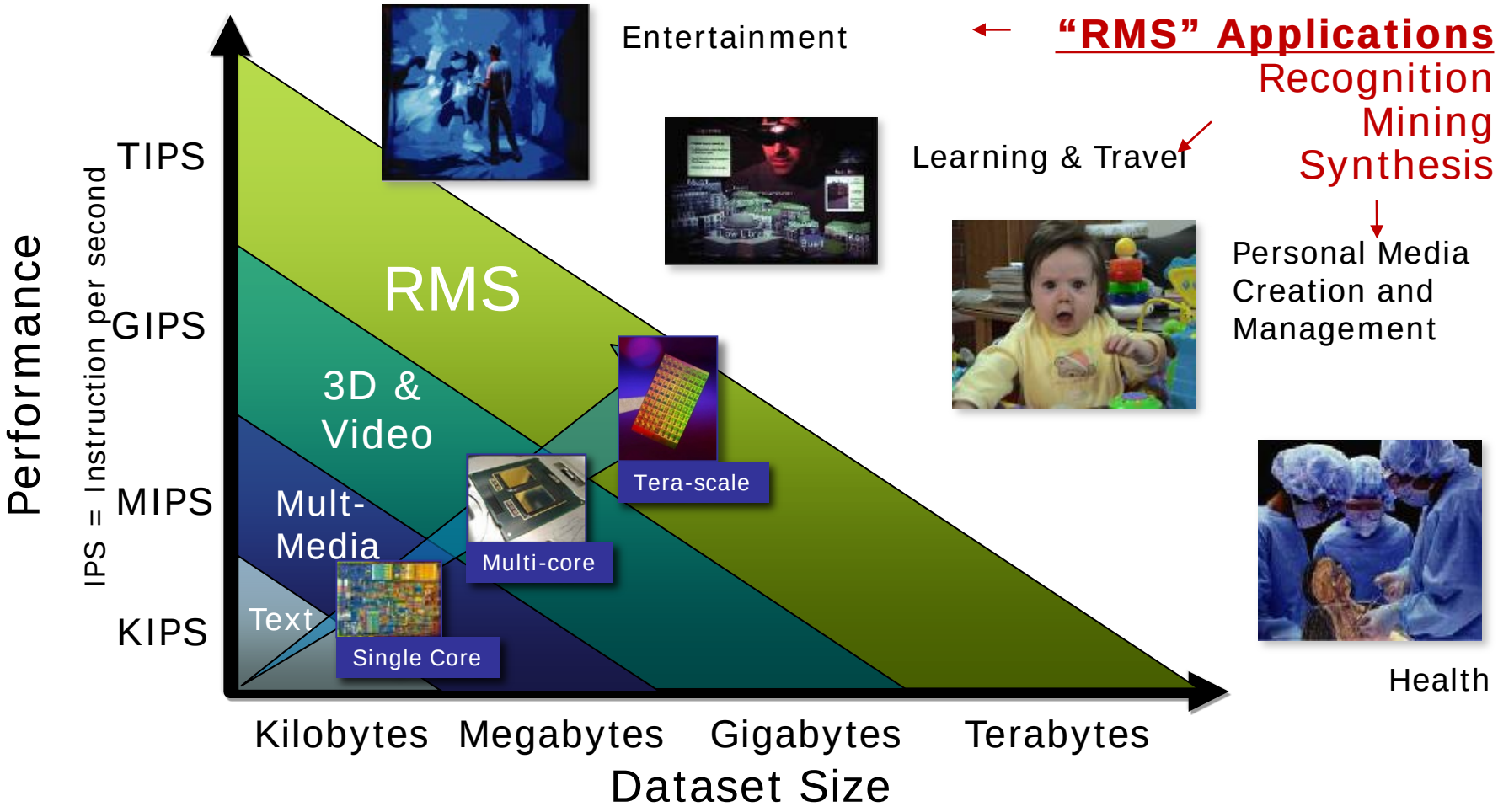
Software AND Services

Agenda

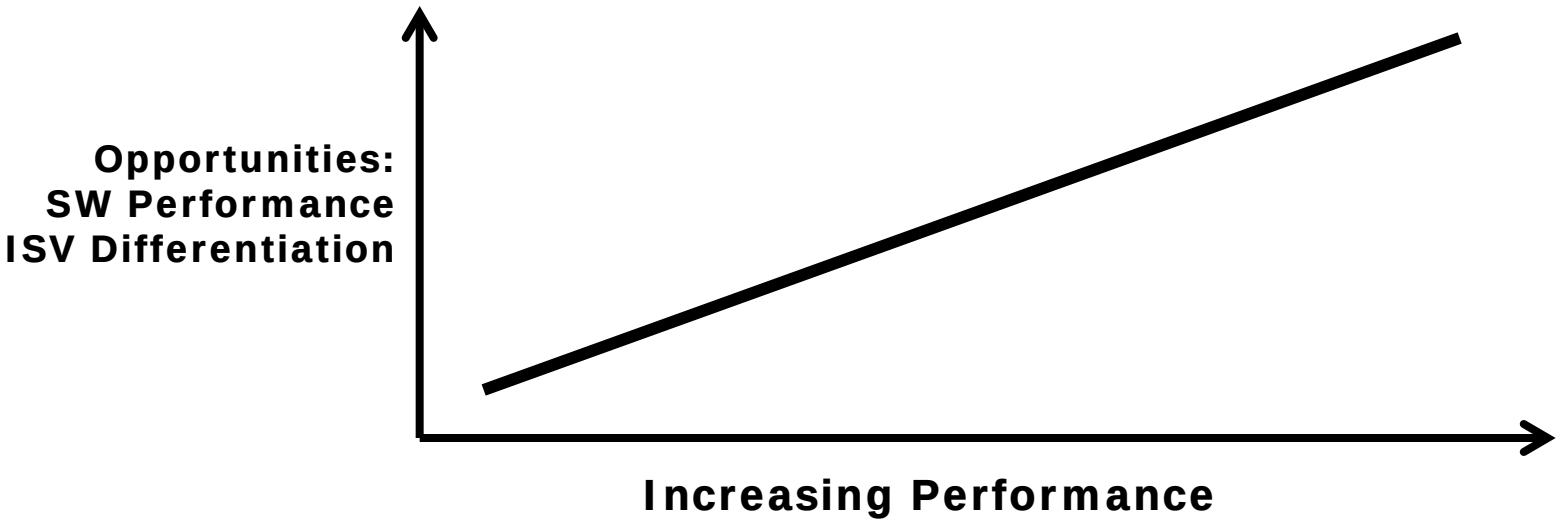
- What is Intel's Ct Technology?
- How does Intel's Ct Technology work?
- How to port C++ code to Intel's Ct Technology?

What is Intel’s Ct Technology?

Throughput and Visual Computing Applications



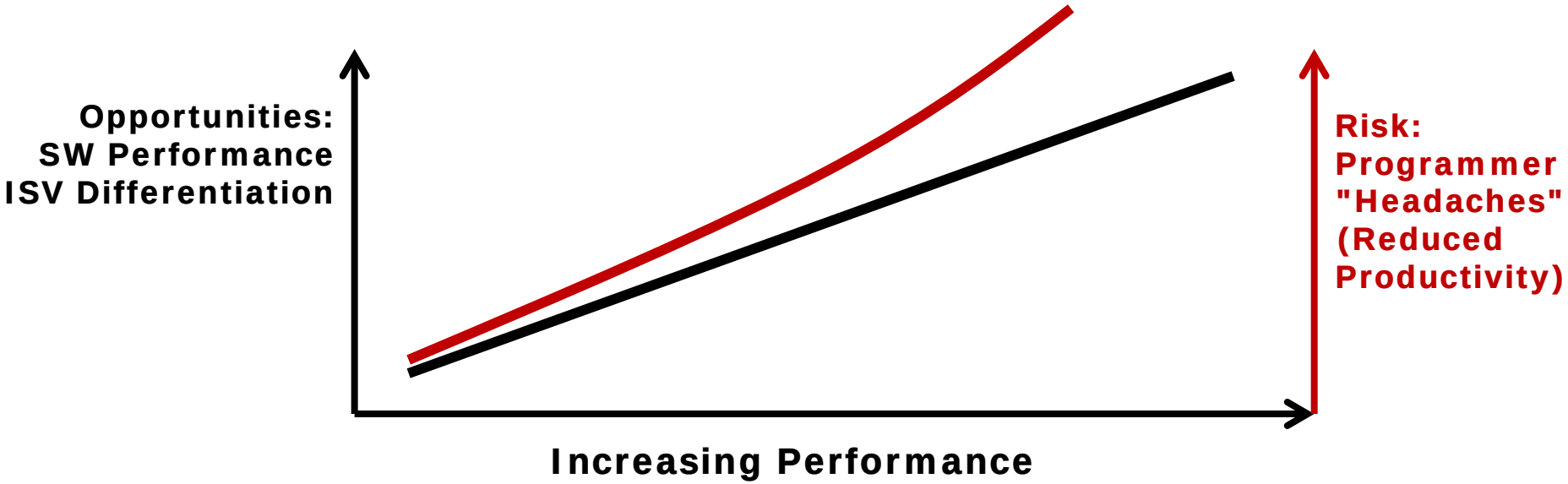
Software Opportunities and Risks



Many-core architecture offers unprecedented opportunities for differentiation:

- **Model-based applications**: New functionalities enriching the user's experience
- **Improved quality:** Higher resolution/accuracy in results
- **Increased usability:** Raw power to fuel more sophisticated usage models

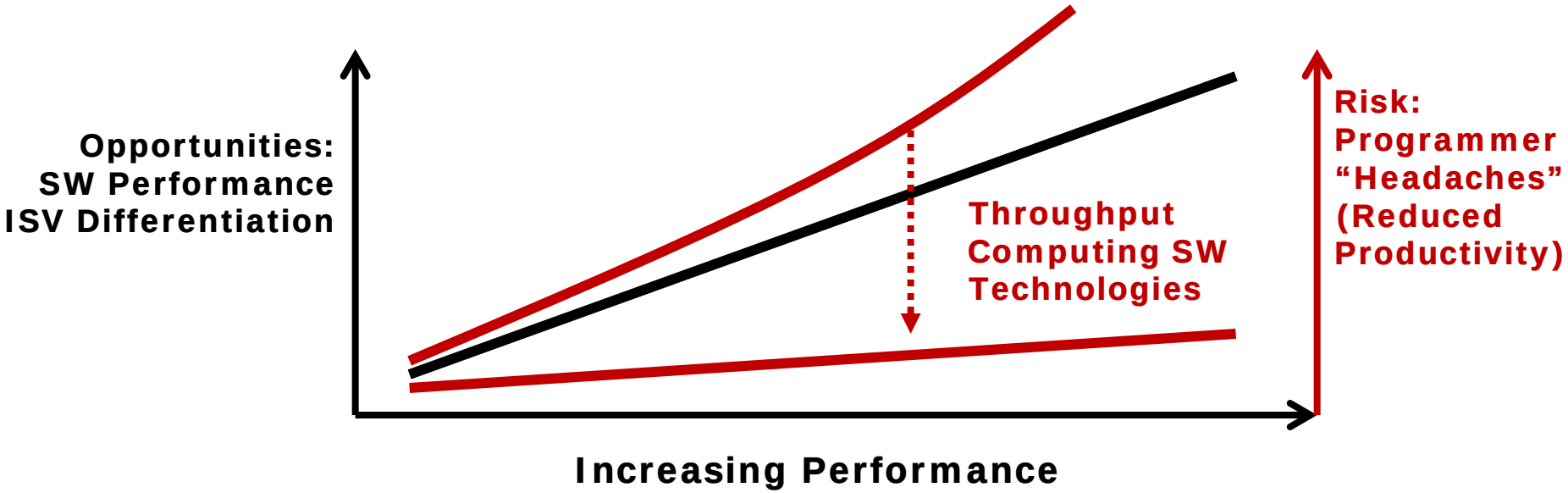
Software Opportunities and Risks



Reduced Productivity:

- **Data races:** New class of bugs that increase exponentially with degree of parallelism
- **Performance tuning:** Programmers can expect to spend most time here
- **Forward scaling:** Anticipating future HW enhancements
- **Modularity:** Difficult to compose parallel programs
- **Proprietary Tools:** Reduces choice and challenges build infrastructure

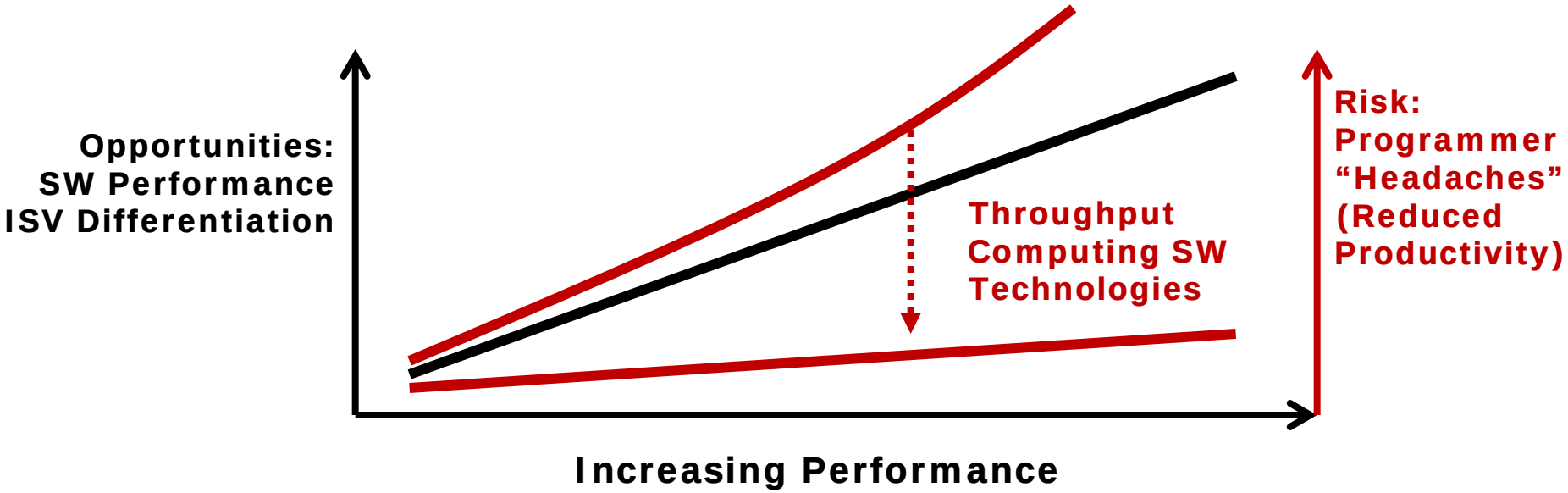
Software Opportunities and Risks



Visual computing software technologies make it easier:

- **Industry Standard APIs:** DirectX*, OpenGL*
- **Native Tools:** SSE/AVX/Co-processor Native Compilers, OpenMP*, Intel® Threading Building Blocks, etc.
- **New Tools**, including:
  - ***Ct - Forward-scaling, easy-to-use, and deterministic***

Software Opportunities and Risks

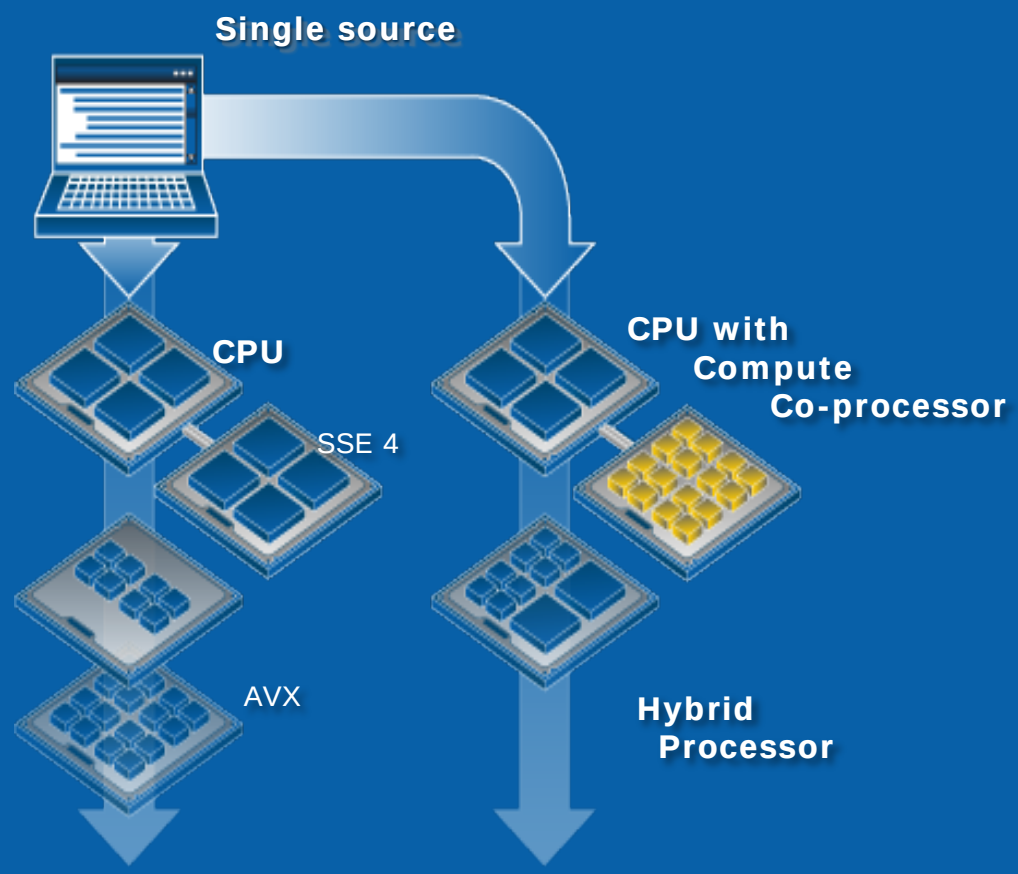


Productivity is greatly increased with the right choice of tools

Intel's Ct Technology

- **Productive Data Parallel Programming**
  - Presented as high-level abstraction, natural notation
  - Delivers application performance with ease of programming
- **Ct forward-scales software written today**
  - Ct is designed to be dynamically retargetable to SSE, AVX, co-processor, and beyond
- **Extends standard C++ using templates**
  - No changes to standard C++ compilers
- **Ct abstracts away architectural details**
  - Vector ISA width / Core count / Memory model / Cache sizes
- **Ct is deterministic**
  - No data races

Forward Scaling with Ct



Productivity

- Integrates with existing tools
- Applicable to many problem domains
- Safe by default: maintainable

Performance

- Efficient and scalable
- Harnesses both vectors and threads
- Eliminates modularity overhead of C++

Portability

- High-level abstraction
- Hardware independent
- Forward scaling

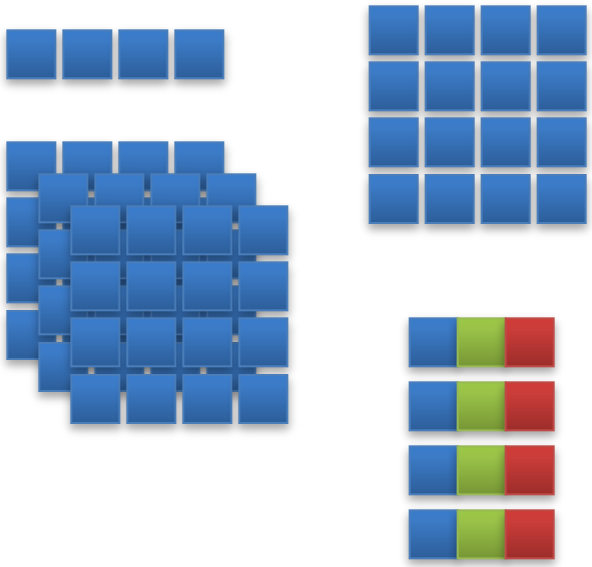
What Does the Product Based on Intel Ct Technology Look Like?

- Core API
  - Flexible, forward scaling data parallelism in C++
- Application Libraries
  - Linear Algebra, FFT, Random Number Generation
    - Powered by Intel® Math Kernel Library (Intel® MKL)!
- Samples
  - Medical Imaging, Financial Analytics, Seismic Processing, and more
- Initial release on Windows, followed by Linux*
  - IA-32 and Intel® 64
  - Works with Intel® C/C++ Compiler, Microsoft* Visual C++*, and GCC*
  - Works with Intel® VTune™ Analyzer

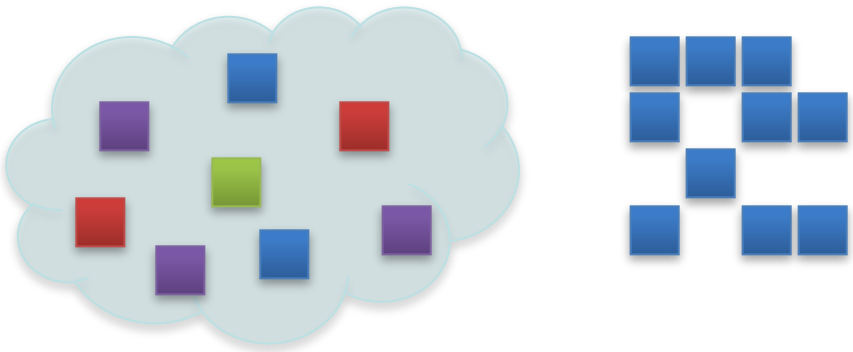
How does Intel’s Ct Technology work?

Ct's Parallel Collections

Regular dense containers



Irregular dense containers



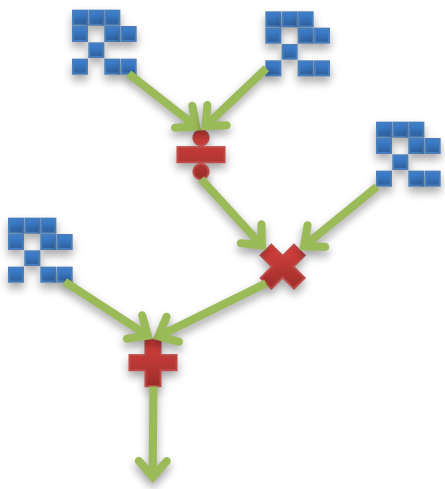
& growing…

- Ct operates on data collections called containers
  - Based on C++ templates
- Ct's containers represent a variety of regular and irregular data collections
  - dense<T>, dense<T,2>, dense<T,3>, nested<T>, indexed <T>

Expressing Computation in Ct

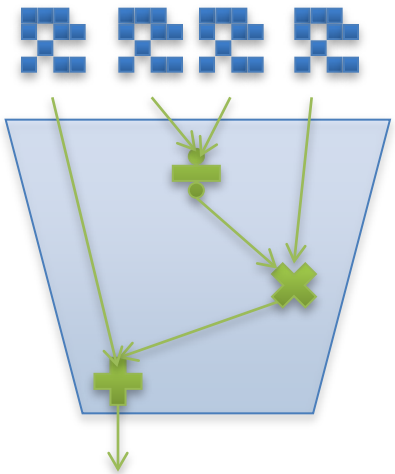
The Ct JIT Automates This Transformation

Vector Processing



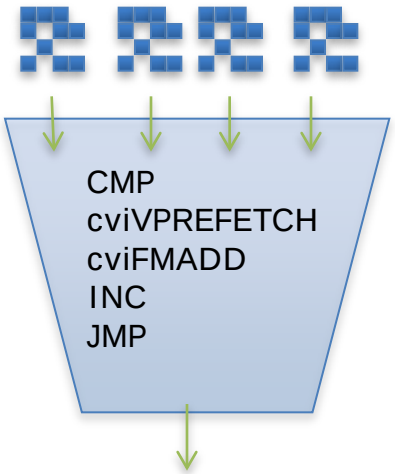
```
dense<f32> A, B, C, D;
A += B/C * D;
```

Scalar Processing



```
f32 kernel(f32 a, b, c, d) {
        return a + (b/c)*d;
}
...
dense <f32> A, B, C, D;
A = map(kernel)(A, B, C, D);
```

Native/Intrinsic Coding




```
native(ndense<f32> a, b, c, d) {
        __asm__ {
                ...
        };
}
...
ndense<f32> A, B, C, D;
A = nmap(native)(A, B, C, D);
```

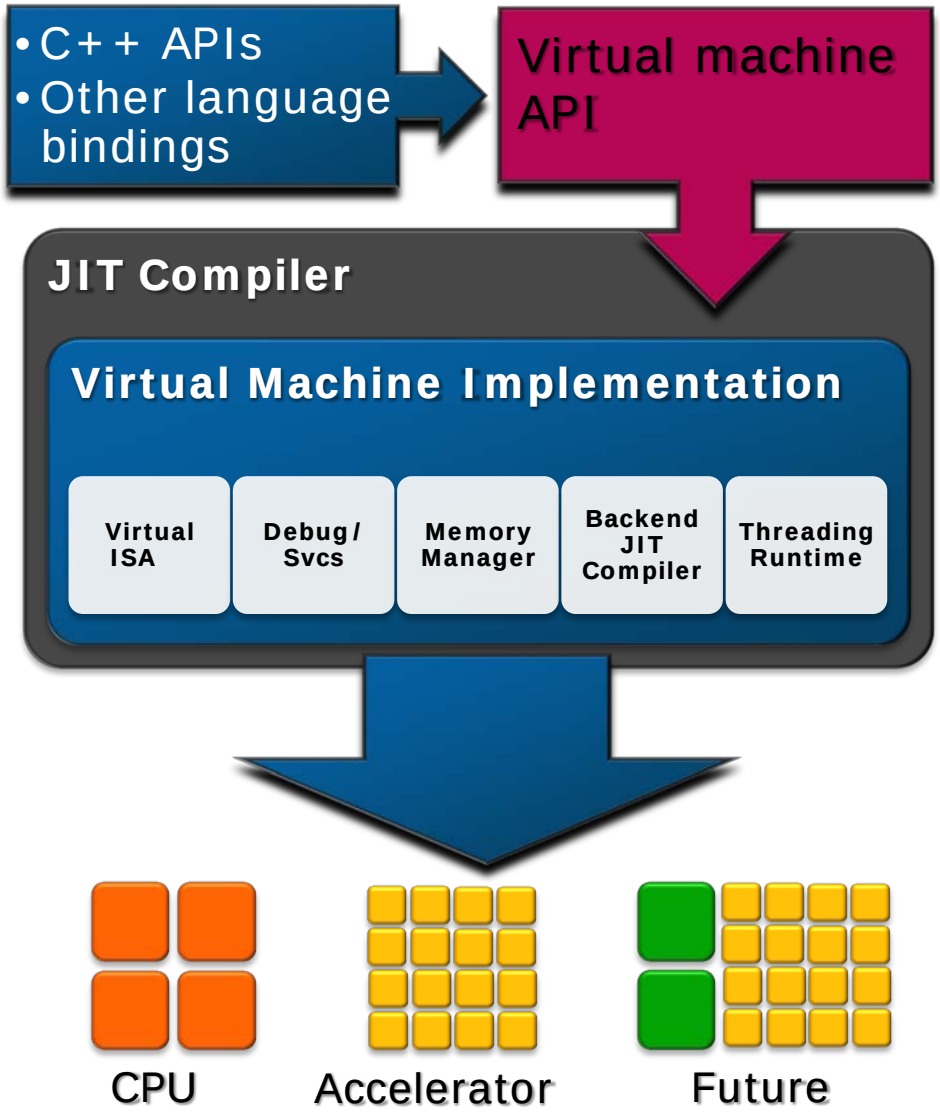
Or Programmers Can Choose Desired Level of Abstraction

The Ct Runtime

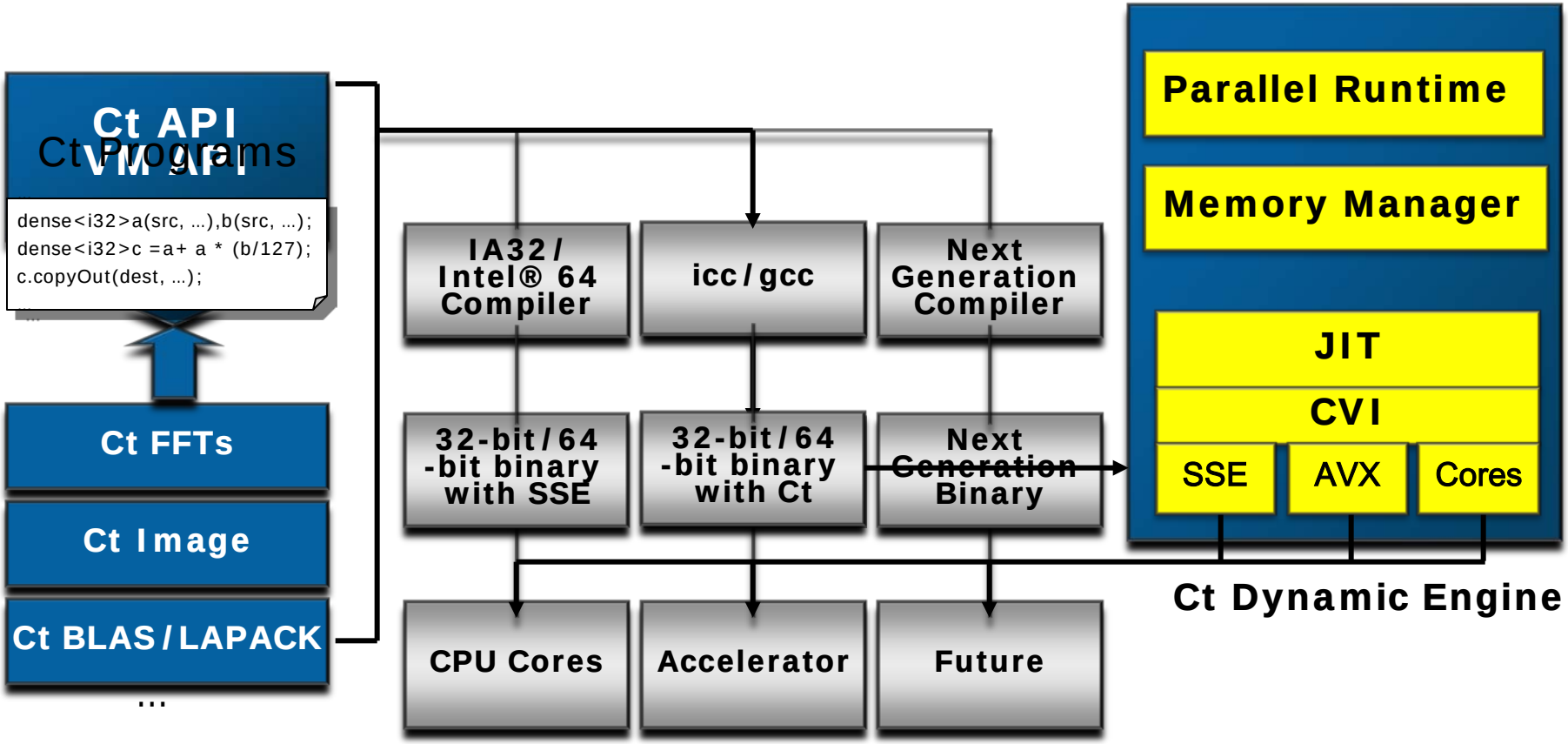
- Intel Ct Technology offers a standards compliant C++ library…

…backed by a runtime

- Runtime generates and manages threads and vector code, via
  - Machine independent optimization
  - Offload management
  - Machine specific code generation and optimizations
  - Scalable threading runtime (based on Intel® Threading Building Blocks)



Dynamic Ct: Supporting All Intel Platforms



Once compiled, run on all Intel platforms

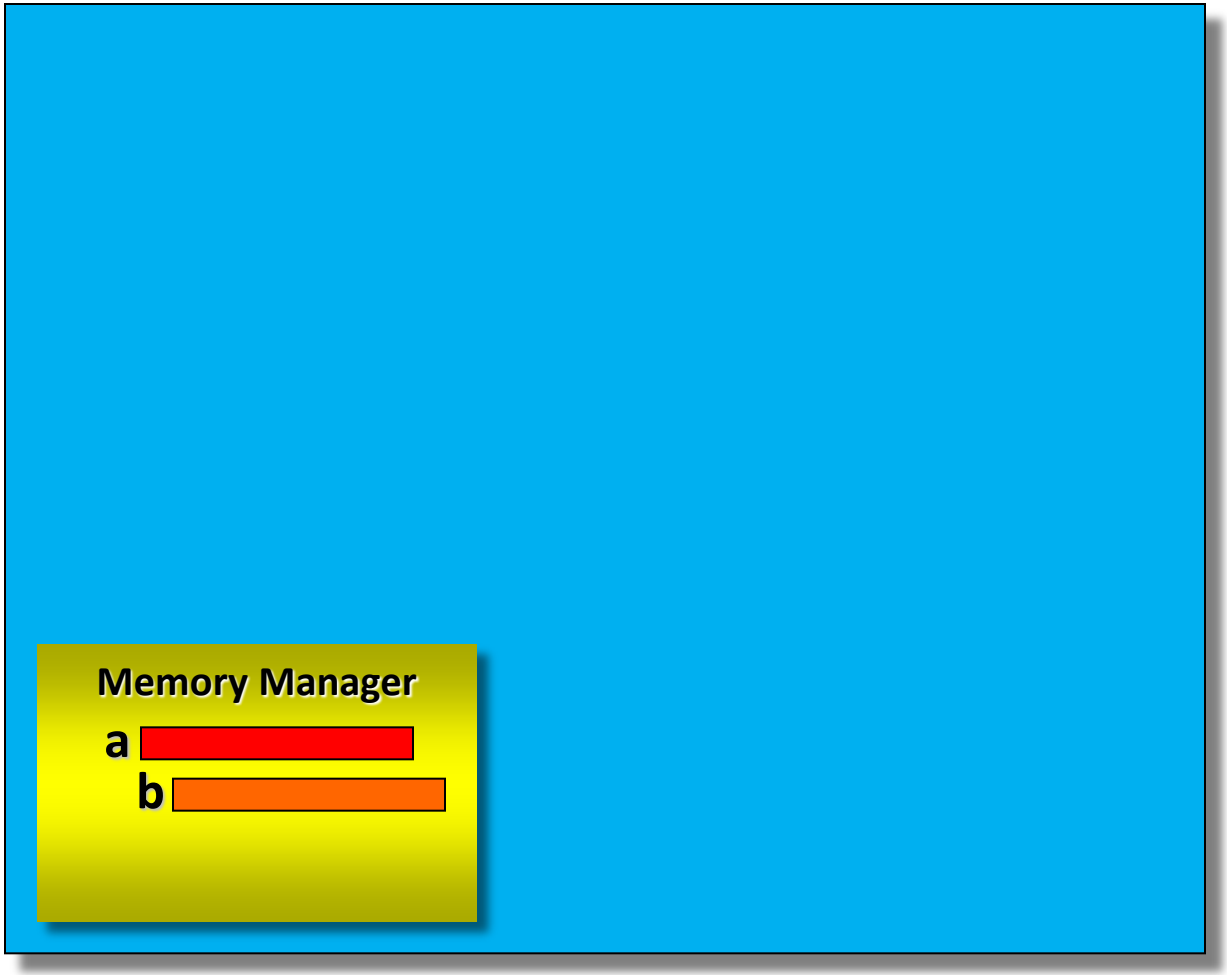
Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, …);
dense<i32> B(ar_b,…);
call( work ) ( A, B );
```

Ct Dynamic Engine

Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, …);
dense<i32> B(ar_b,…);
call( work ) ( A, B );
```

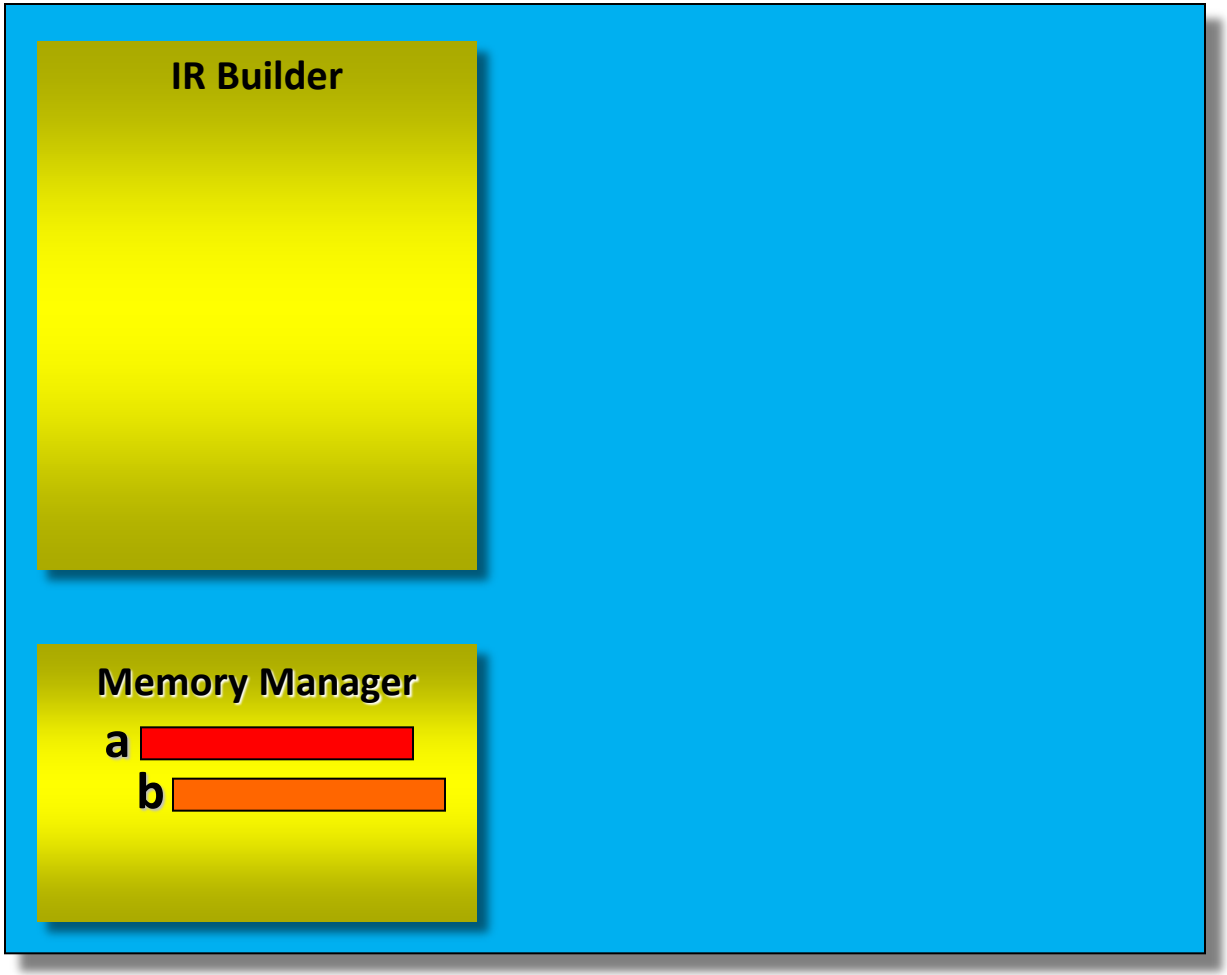


Ct Dynamic Engine

Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, ...);
dense<i32> B(ar_b,...);
call( work ) ( A, B );
```

```
void work(dense<i32> a,
      dense<i32> & b )
{
    b = a + 1;
}
```

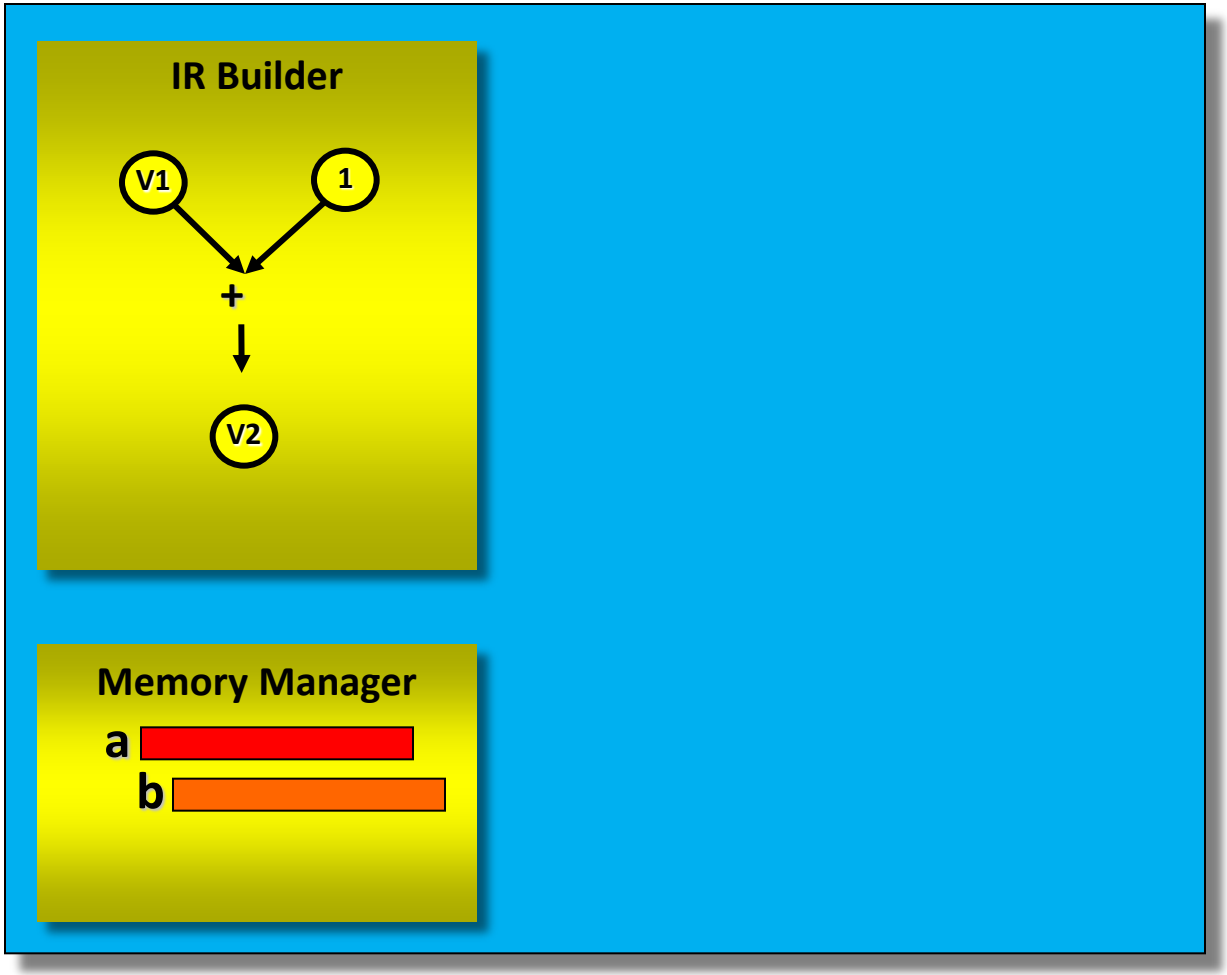


Ct Dynamic Engine

Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, ...);
dense<i32> B(ar_b,...);
call( work ) ( A, B );
```

```
void work(dense<i32> a,
      dense<i32>& b )
{
    b = a + 1;
}
```

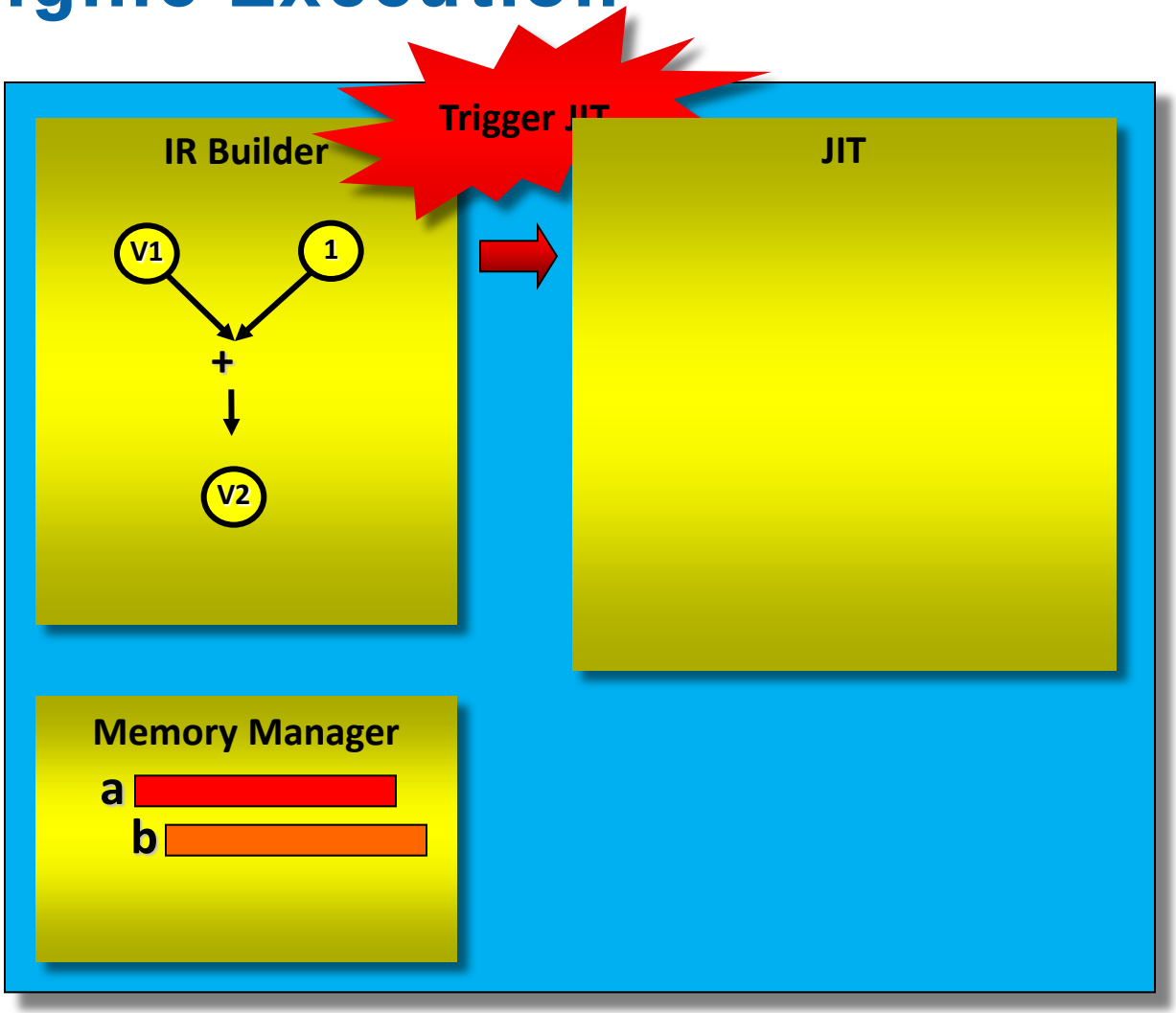


Ct Dynamic Engine

Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, …);
dense<i32> B(ar_b,…);
call( work ) ( A, B );
```

```
void work(dense<i32>a,
      dense<i32>& b )
{
   b = a + 1;
}
```

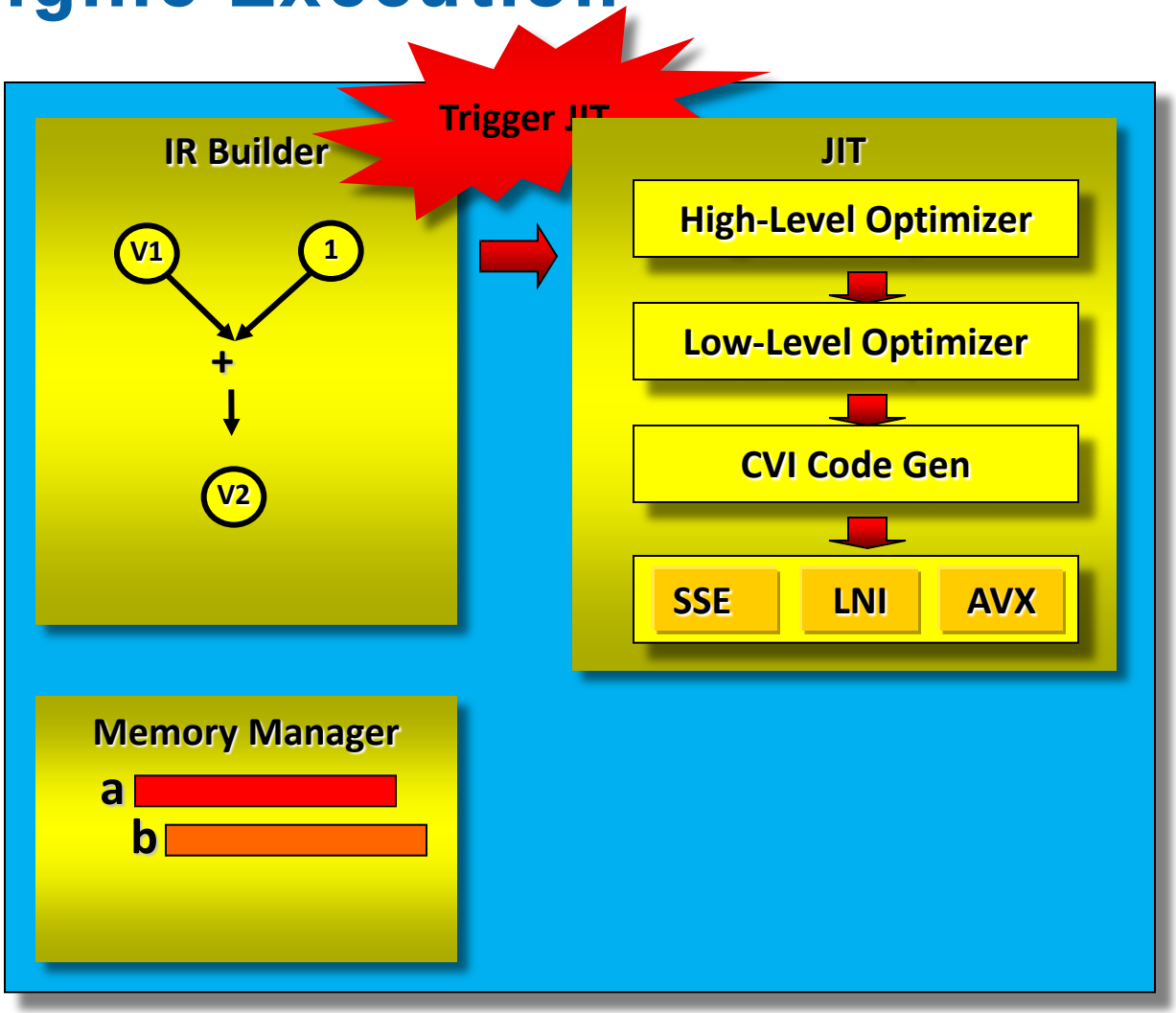


Ct Dynamic Engine

Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, …);
dense<i32> B(ar_b,…);
call( work ) ( A, B );
```

```
void work(dense<i32> a,
      dense<i32> & b )
{
   b = a + 1;
}
```

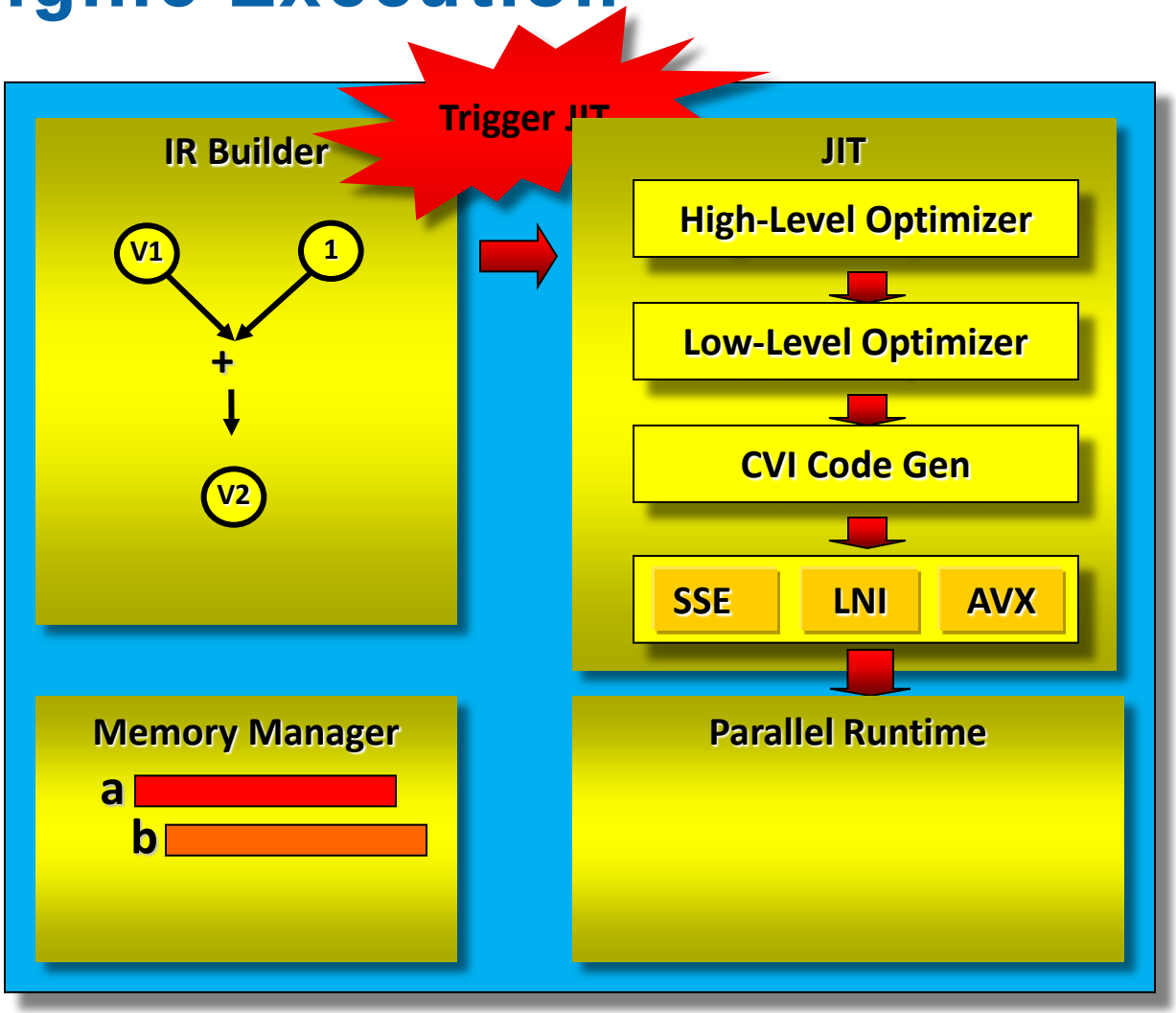


Ct Dynamic Engine

Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, …);
dense<i32> B(ar_b,…);
call( work ) ( A, B );
```

```
void work(dense<i32> a,
    dense<i32> & b )
{
   b = a + 1;
}
```

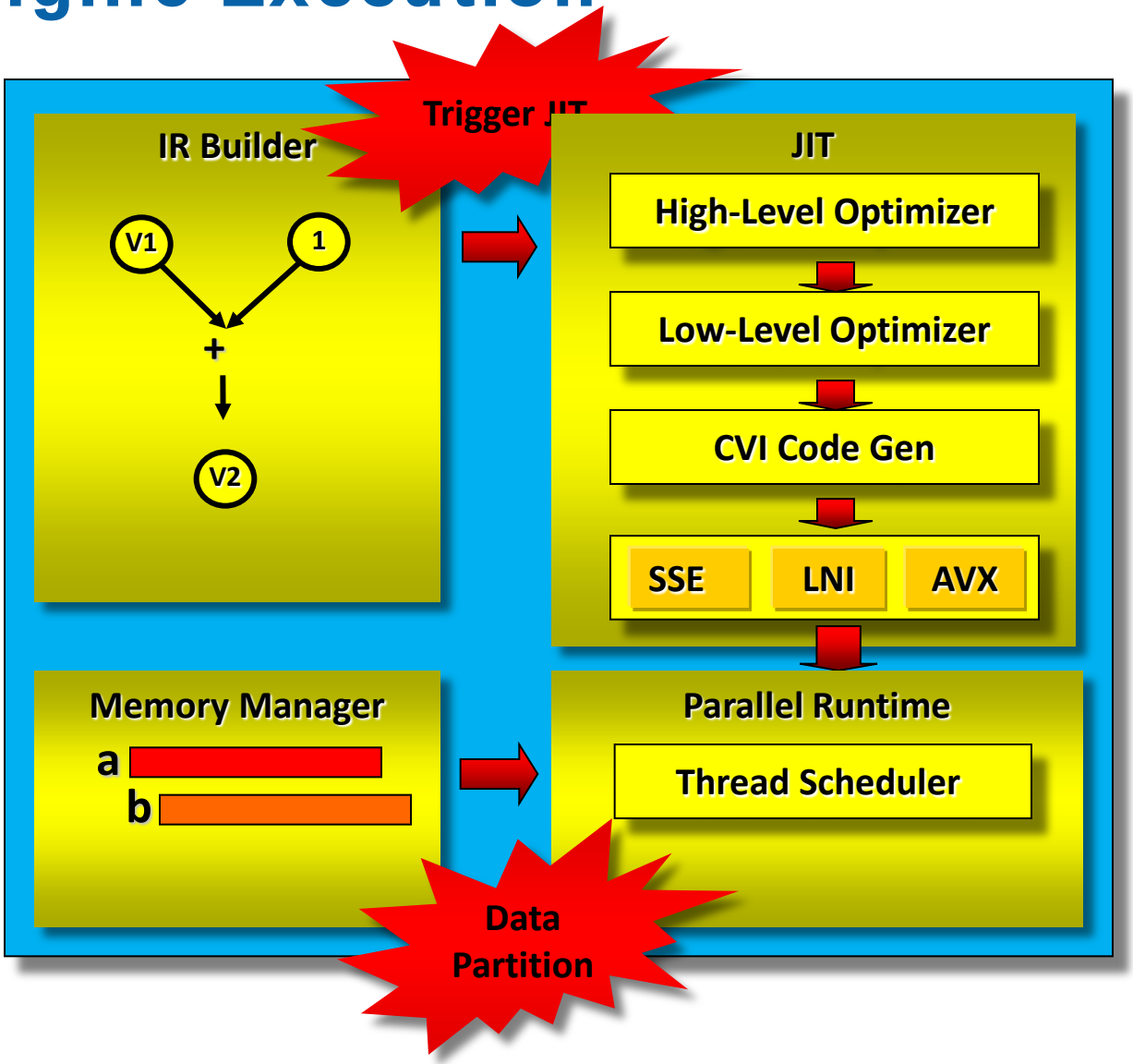


Ct Dynamic Engine

Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, …);
dense<i32> B(ar_b,…);
call( work ) ( A, B );
```

```
void work(dense<i32> a,
      dense<i32> & b )
{
   b = a + 1;
}
```

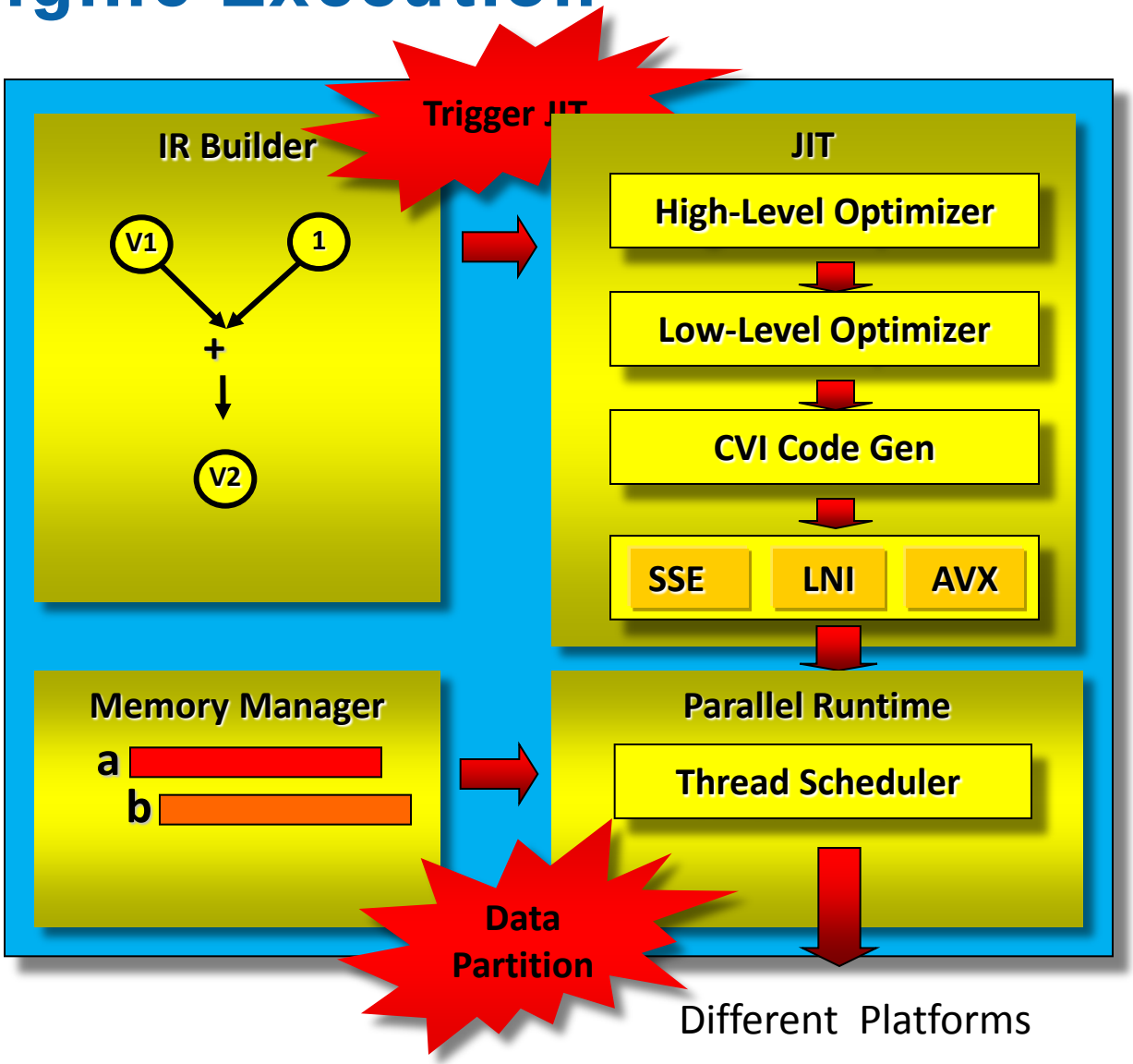


Ct Dynamic Engine

Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, …);
dense<i32> B(ar_b,…);
call( work ) ( A, B );
```

```
void work(dense<i32> a,
      dense<i32> & b )
{
   b = a + 1;
}
```



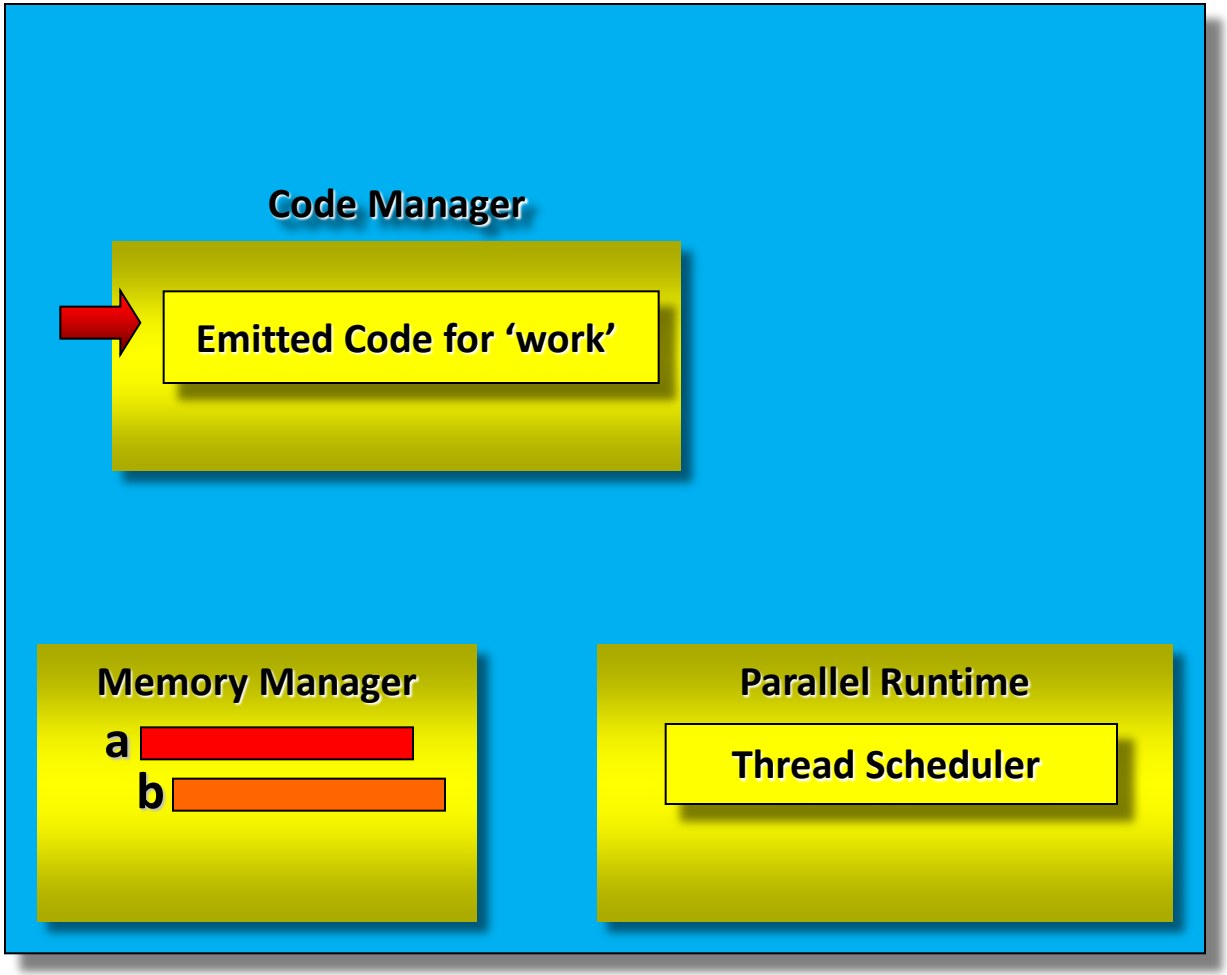
Ct Dynamic Engine

Different Platforms

Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, …);
dense<i32> B(ar_b,…);
call( work ) ( A, B );
```

Compute Kernel Again



Ct Dynamic Engine

Ct Dynamic Engine Execution

```
int ar_a[1024], ar_b[1024]

dense<i32> A(ar_a, …);
dense<i32> B(ar_b,…);
call( work ) ( A, B );
```

```
void
```

Compute Kernel Again

Code Manager

Emitted Code for 'work'

Code Cache

Memory Manager

a

b

Parallel Runtime

Thread Scheduler

Data Partition

Ct Dynamic Engine

Different Platforms

Ct Dynamic Engine Implications

- **Second stage of compilation at runtime**

  Allows run-time specialization
- **Ct kernel: exists as C function, but**
  - Executes only once => for dynamic compilation
  - Symbolic “capture” of computation
  - Takes a snapshot of current bindings and state
- **Ct call: execution is a remote function call**
  - Data flow dependencies inferred
  - Automatic (and always safe) synchronization
  - Enables remote execution and data storage

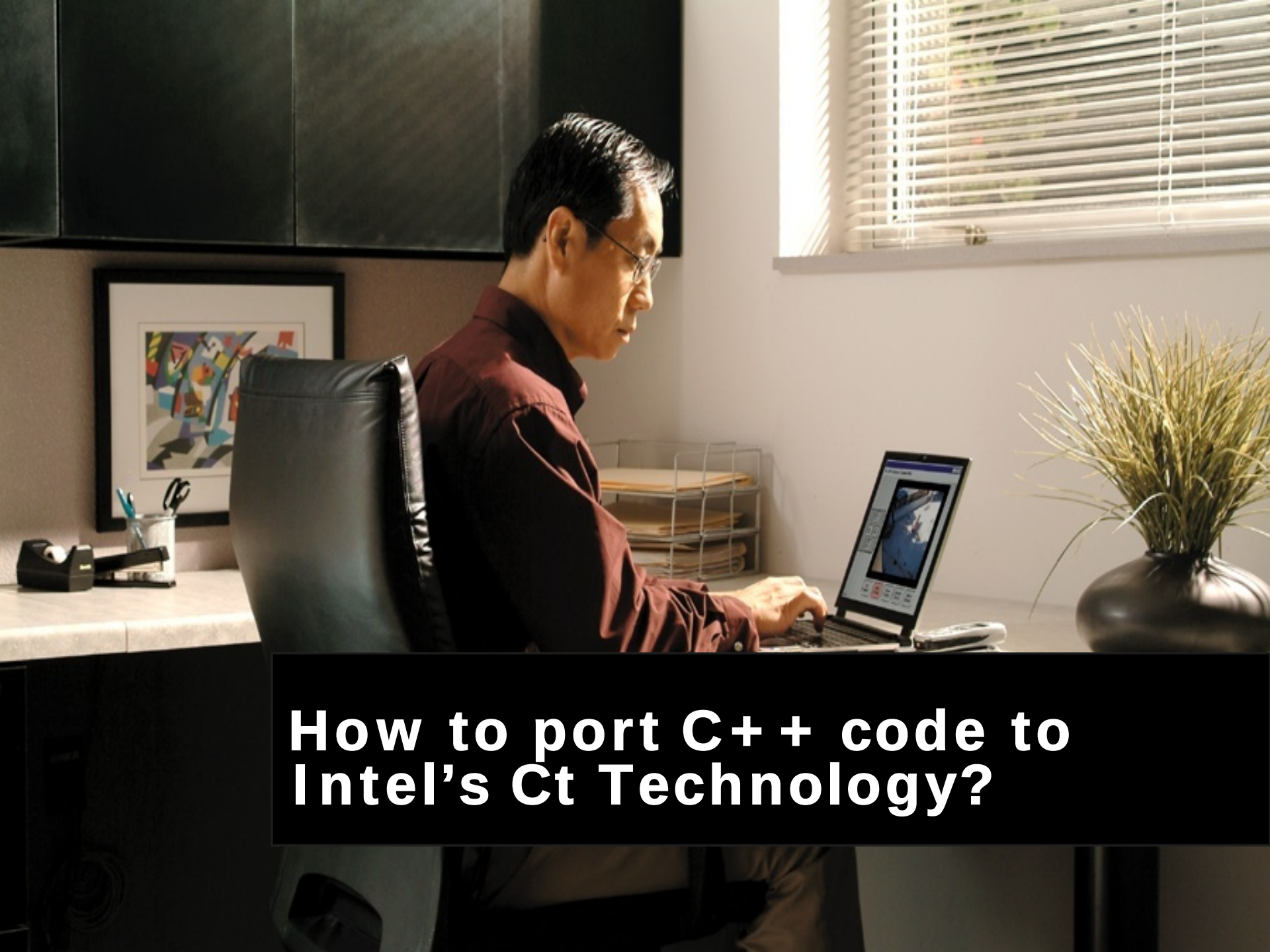
Why Does this Matter?

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

- Virtual function calls, function pointers, and control flow dependent on parameters not known until runtime can limit performance
- Modularity inherently spreads computation across methods
- In extreme cases, the application is effectively an interpreter for a computation specified at runtime

→ ***Dynamic compilation can "compile out" the overhead of such "late binding"***

→ ***This advantage is MULTIPLICATIVE with parallelism***



How to port C++ code to Intel's Ct Technology?

Ct Programming Common Steps

- Identify the computation logic to be written in Ct
- Figure out the signature of the kernel
- Prepare C data buffer for input/output
- Set up C=>Ct kernel bridge by rcall
- Implement kernel

Step 1: Figure Out Kernel Signature

```
int ar_a[1024];
int ar_b[1024];
...
...
...
...
for( i=0; i<1024; i++) {
    ar_b[i] = ar_a[i] + 1;
}
```

```
#include “ct.h”
using namespace Ct;
...
...
int ar_a[1024];
int ar_b[1024];
...
dense<i32> A( ar_a, 1024 );
dense<i32> B( ar_b, 1024 );
call( work ) ( A, B );
...
void work(dense<i32>a, dense<i32> &b){
    b = a + 1;
}
```

Step 2: Prepare Data

```
int ar_a[1024];
int ar_b[1024];
...
...
...
...
for( i=0; i<1024; i++) {
    ar_b[i] = ar_a[i] + 1;
}
```

```
#include “ct.h”
using namespace Ct;
 ...
 ...
 int ar_a[1024];
 int ar_b[1024];
 ...
 dense<i32> A( ar_a, 1024 );
 dense<i32> B( ar_b, 1024 );
 call( work ) ( A, B );
  ...
void work(dense<i32> a,dense<i32>&b ){
    b = a + 1;
 }
```

Step 3: Set Up Bridge

```
int ar_a[1024];
int ar_b[1024];
...
...
...
...
for( i=0; i<1024; i++) {
    ar_b[i] = ar_a[i] + 1;
}
```

```
#include “ct.h”
using namespace Ct;
 ...
 ...
 int ar_a[1024];
 int ar_b[1024];
 ...
 dense<i32> A( ar_a, 1024 );
 dense<i32> B( ar_b, 1024 );
 call( work ) ( A, B );
  ...
void work(dense<i32> a, dense<i32>&b ){
    b = a + 1;
 }
```

Step 4: Implement Kernel

```
int ar_a[1024];
int ar_b[1024];
...
...
...
...
for( i=0; i<1024; i++) {
    ar_b[i] = ar_a[i] + 1;
}
```

```
#include “ct.h”
using namespace Ct;
 ...
 ...
 int ar_a[1024];
 int ar_b[1024];
 ...
 dense<i32> A( ar_a, 1024 );
 dense<i32> B( ar_b, 1024 );
 call( work ) ( A, B );
...
void work(dense<i32>a, dense<i32>&b ){
    b = a + 1;
}
```

Example: Black-Scholes for Options Pricing

Black-Scholes Using C Loops

```
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));
}
```

Black-Scholes Using Ct

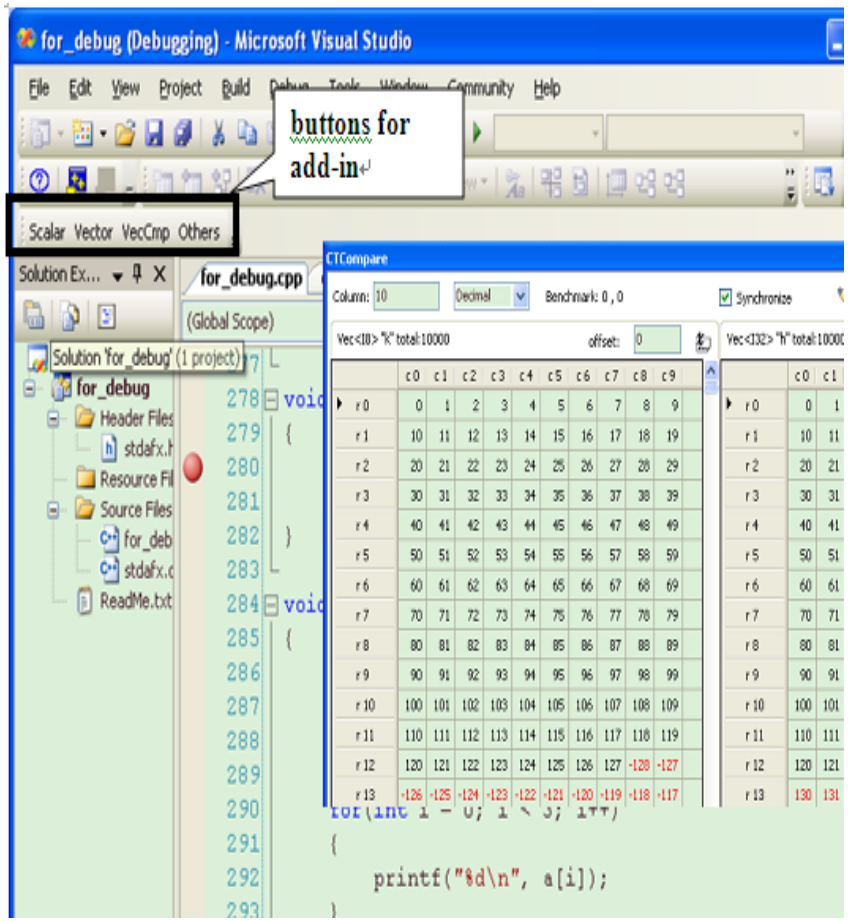
```
#include <ct.h>
using namespace OpenCt;
float s[N], x[N], r[N], v[N], t[N];
float result[N];
dense<f32> S(s, N), X(x, N), R(r, N), V(v, N), T(t, N);

dense<f32> d1 = S / ln(X);
d1 += (R + V * V * 0.5f) * T;
d1 /= sqrt(T);
dense<f32> d2 = d1 – sqrt(T);

dense<f32> tmp = X * exp(R * T) *
( 1.0f - CND(d2)) + (-S) * (1.0f - CND(d1));

tmp.copyOut(result, sizeof(result));
```

1. #include <ct.h> and using namespace
2. Vector operations subsumes loop
3. The Ct code is almost the same as the original loop body
4. copyOut at the end (if you're done!)

Vector-style computation is a natural approach here.

Ct Debugging

- Link with the Emulation Library and debug with a C++ debugger
- Ct provides an Add-in to help watch Ct variables in Visual Studio*
- Dump the Intermediate Representation at various levels
- Use environments to control the Intermediate Representation level output level:
  - CT_DUMPJIT=y
  - HLO_TRACE_LEVEL =n
  - HLO_DISABLE_OPT=dump_cpp=f



CT Add-in in Visual Studio*

Summary

- Ct enables you to write simple parallel algorithms in standard C++
- Ct can get you performance on Intel Architecture today
- Ct Technology today will be forward-scalable to new many core and compute co-processors
- Ct will intermix with other parallel programming models and tools

Ct extends the many choices that Intel provides for Parallel Computing

Call to Action

Sign up for the opportunity to participate in Ct technology beta!

http://www.intel.com/software/data_parallel/

Additional sources of information on this topic:

- Visit our demo and talk to us at Booth #1116:
  Medical imaging on deformable registration

- More web based information on Ct technology:
  http://software.intel.com/en-us/data-parallel/

- Data parallelism whitepapers and tutorials:
  http://software.intel.com/en-us/articles/data-parallelism-whitepapers-and-tutorials/

Session Presentations - PDFs

The PDF for this Session presentation is available from our IDF Content Catalog at the end of the day at:

intel.com/go/idfsessionsBJ

URL is on top of Session Agenda Pages in Pocket Guide

Q&A

Please Fill out the Session Evaluation Form

Give the completed form to the room monitors as you exit!

Thank You for your input, we use it to improve future Intel Developer Forum events

Legal Disclaimer

- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.
- Intel may make changes to specifications and product descriptions at any time, without notice.
- All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
- Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.
- Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.
- Intel, and the Intel logo are trademarks of Intel Corporation in the United States and other countries.
- *Other names and brands may be claimed as the property of others.
-

Risk Factors

The above statements and any others in this document that refer to plans and expectations for the first quarter, the year and the future are forward-looking statements that involve a number of risks and uncertainties. Many factors could affect Intel's actual results, and variances from Intel's current expectations regarding such factors could cause actual results to differ materially from those expressed in these forward-looking statements. Intel presently considers the following to be the important factors that could cause actual results to differ materially from the corporation's expectations. Demand could be different from Intel's expectations due to factors including changes in business and economic conditions; customer acceptance of Intel's and competitors' products; changes in customer order patterns including order cancellations; and changes in the level of inventory at customers. Intel operates in intensely competitive industries that are characterized by a high percentage of costs that are fixed or difficult to reduce in the short term and product demand that is highly variable and difficult to forecast. Additionally, Intel is in the process of transitioning to its next generation of products on 32nm process technology, and there could be execution issues associated with these changes, including product defects and errata along with lower than anticipated manufacturing yields. Revenue and the gross margin percentage are affected by the timing of new Intel product introductions and the demand for and market acceptance of Intel's products; actions taken by Intel's competitors, including product offerings and introductions, marketing programs and pricing pressures and Intel's response to such actions; defects or disruptions in the supply of materials or resources; and Intel's ability to respond quickly to technological developments and to incorporate new features into its products. The gross margin percentage could vary significantly from expectations based on changes in revenue levels; product mix and pricing; start-up costs, including costs associated with the new 32nm process technology; variations in inventory valuation, including variations related to the timing of qualifying products for sale; excess or obsolete inventory; manufacturing yields; changes in unit costs; impairments of long-lived assets, including manufacturing, assembly/test and intangible assets; the timing and execution of the manufacturing ramp and associated costs; and capacity utilization;. Expenses, particularly certain marketing and compensation expenses, as well as restructuring and asset impairment charges, vary depending on the level of demand for Intel's products and the level of revenue and profits. The majority of our non-marketable equity investment portfolio balance is concentrated in companies in the flash memory market segment, and declines in this market segment or changes in management's plans with respect to our investments in this market segment could result in significant impairment charges, impacting restructuring charges as well as gains/losses on equity investments and interest and other. Intel's results could be impacted by adverse economic, social, political and physical/infrastructure conditions in countries where Intel, its customers or its suppliers operate, including military conflict and other security risks, natural disasters, infrastructure disruptions, health concerns and fluctuations in currency exchange rates. Intel's results could be affected by the timing of closing of acquisitions and divestitures. Intel's results could be affected by adverse effects associated with product defects and errata (deviations from published specifications), and by litigation or regulatory matters involving intellectual property, stockholder, consumer, antitrust and other issues, such as the litigation and regulatory matters described in Intel's SEC reports. An unfavorable ruling could include monetary damages or an injunction prohibiting us from manufacturing or selling one or more products, precluding particular business practices, impacting our ability to design our products, or requiring other remedies such as compulsory licensing of intellectual property. A detailed discussion of these and other risk factors that could affect Intel's results is included in Intel's SEC filings, including the report on Form 10-Q.

Rev. 1/14/10