Intel® Array Building Blocks

Hans Pabst
Software Engineer TCE
Software and Services Group
Intel

Introduction to Intel® Array Building Blocks

Introduction: Objectives

- Understand the motivation for Intel® Array Building Blocks
  - Also known as Intel® ArBB
- Understand the Intel® ArBB C++ API-as-a-language
- Understand the basic syntax of the Intel® ArBB API
- Review the available operators
- Be able to write a first "Hello World" application w/ ArBB
- Work through a few example applications

Optimization Notice

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2010. Intel Corporation.

Optimization Notice

Optimization Notice

Optimization Notice

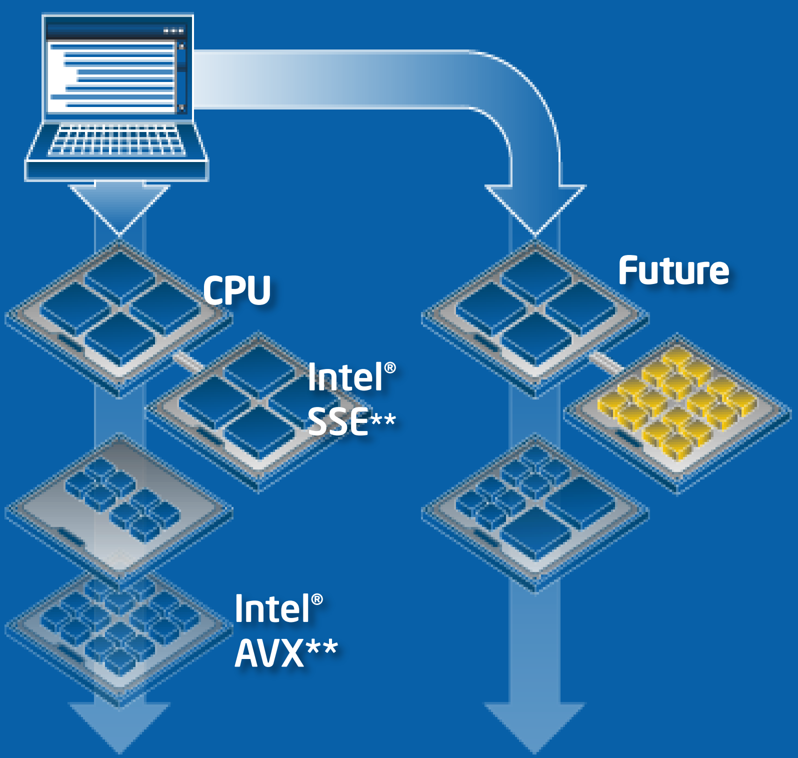
Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

What's Wrong with Parallel Programming?

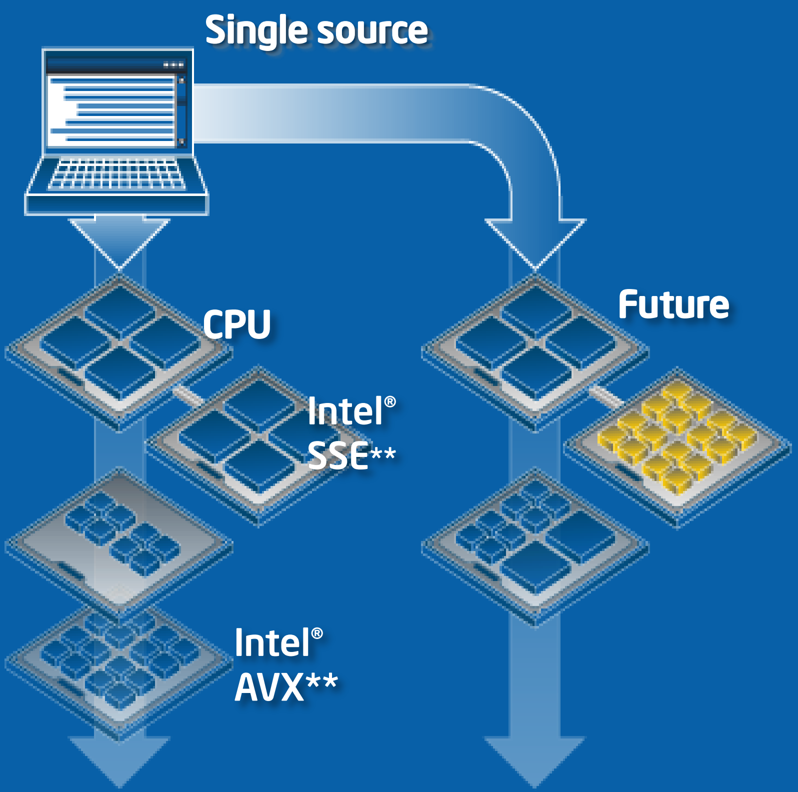


- **Parallel programming is hard**
  - Deadlocks
  - Data races
  - Synchronization
  - Load imbalance
- **Errors inhibit productivity**
- **No uniform programming model for**
  - Intel SSE, Intel AVX
  - Multi-threading
  - IA manycore
- **Parallel programmers lose single code base for their applications**

** Intel® Streaming SIMD Extensions
Intel® Advanced Vector Extensions

Optimization Notice

Intel® Array Building Blocks



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

** Intel® Streaming SIMD Extensions
Intel® Advanced Vector Extensions

Optimization Notice

Productivity

- Integrates
  with existing IDEs, tools, and compilers: no new compiler needed
- Interoperates
  with other Intel parallel programming tools and libraries
- Incremental
  allows selective and targeted modification of existing code bases
- Expressive
  syntax oriented to application experts
- Safe by default
  deterministic semantics avoid race conditions and deadlock by construction
- Easy to learn
  serially consistent semantics and simple interface leverage existing skills
- Widely applicable
  Generalized data parallel model applicable to many types of computations

Performance

- Scalable to large problems
  manages data to directly address memory bottlenecks
- Unified thread and vector parallelization
  single specification targets multiple mechanisms
- Elimination of modularity overhead
  automatically fuses multiple operations
- Wide *and* deep
  developers can choose level of abstraction
  can drill down to detail if needed

Optimization Notice

Portability

- High-level
  avoids dependencies on particular hardware mechanisms or architectures
- ISA extension independent
  common binary can exploit different ISA extensions transparently
- Allows choice of deployment hardware today
  including scaling to many cores
- Allows migration and forward-scaling
  will support future hardware roadmap

ISA: Instruction Set Architecture

Optimization Notice

Productivity via a High Level of Abstraction

"Specify what to do, not how to do it!"

Mathematical structure
Data organization
Where's my data race?
What caused that deadlock?
Why do I get different answers every time I run this?
How many threads should I use?
How big is my cache?
How do I deal with different ISAs and vector widths?
Where's the guy who originally wrote this thing – I can't figure out what the code is supposed to be computing!

Mathematical structure
Data organization

Goal: increasing the efficiency of the expert application developer

Intel® ArBB vs. Intel® SSE intrinsics

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

  dense<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;
}

templat… en… >
void ct… schol… n(dense<T> …
        …se<T> x,
        …se<T> r,


    …dense<T>& res…

  …nse<T> sqrt_value … rt(t);
    dense<T> d1 = ln(s / x) + (r + v * v * 0.5f ) * t) / sqrt_value;
    dense<T> d2 = d1 - sqrt_value;

    result = x * exp(0f - r * t) * (1.0f - CND(d2)) + (-s) * (1.0 - CND(d1));
}


template <typename T>
void ctBlackScholes(T *option_price,   int num_options,
         T *stkprice,   T *strike,     T *rate,   T *volatility,
         T *time)
{
    dense<T> s(stkprice, num_options);
    dense<T> x(strike, num_options);
    dense<T> r(rate, num_options);
    dense<T> v(volatility, num_options);
    dense<T> t(time, num_options);
    dense<T> result(option_price, num_options);
    call( ctBlackScholesKern<T> )( s,x,r,v,t,result );
}
```

ArBB

42 lines
- vectorized
- threaded
- machine independent

```
#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 Inp…
    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 decimal
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)0.319381530));
  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));

  x… te, xPowerTerm);
  x… M_SUB(xRiskFreeRate, xPowerTerm);

  x… M_MUL(xD1, xTime);
  x… );

  x… M_ADD(xD1, xLogTerm);
  x… M_ADD(xD2, xLogTerm);

  x… 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

186 lines
- vectorized
- *not threaded*
- *machine dependent*

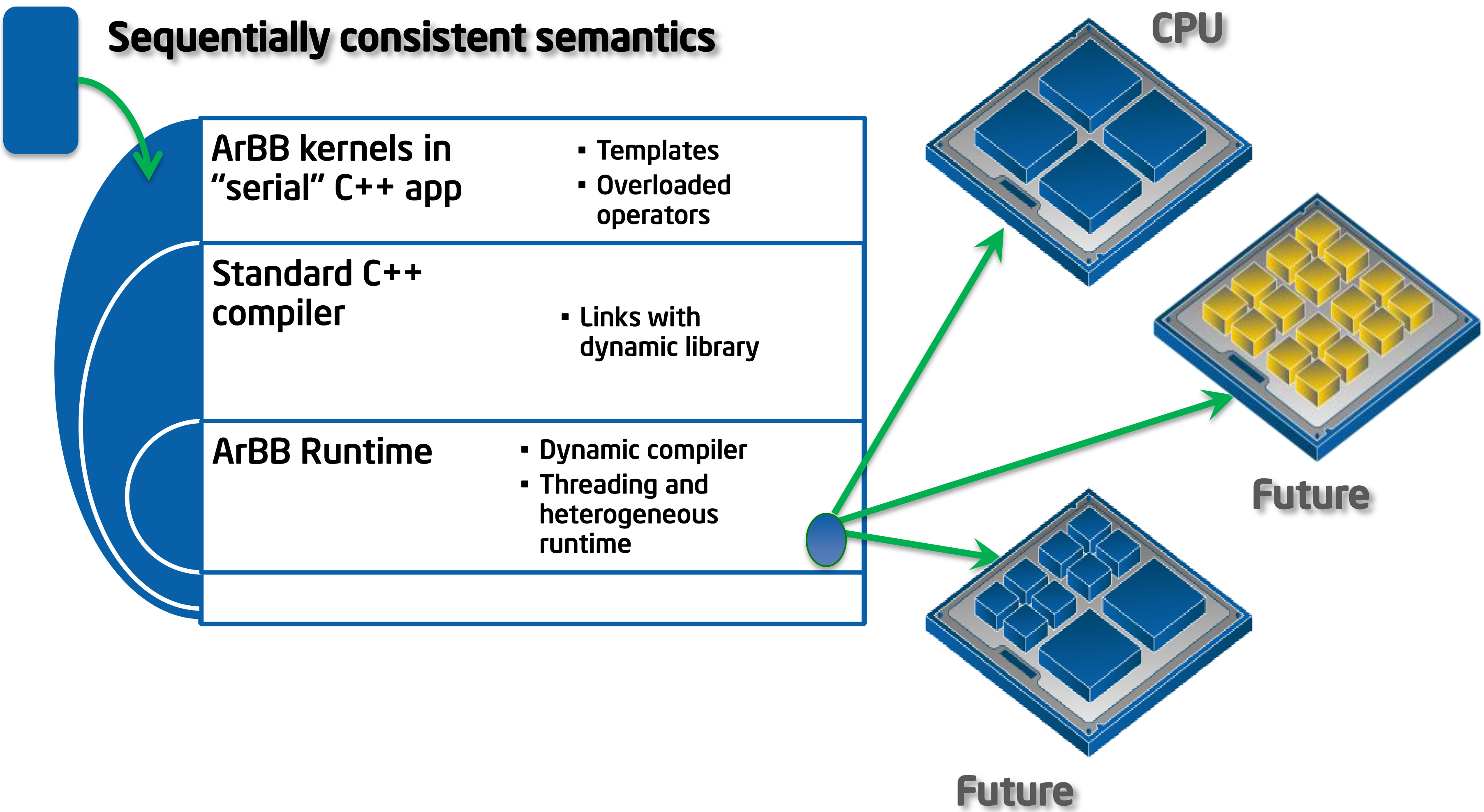
Optimization Notice

Intel's Family of Parallel Models

Intel® Parallel Building Blocks (PBB)		Fixed Function Libraries	Established Standards	Research and Exploration
Intel® Threading Building Blocks (TBB)	Intel® Cilk Plus	Intel® Math Kernel Library (MKL)	MPI	Intel® Concurrent Collections
Intel® Array Building Blocks (ArBB)		Intel® Integrated Performance Primitives (IPP)	OpenMP*	OpenCL*

Optimization Notice

How does it work?

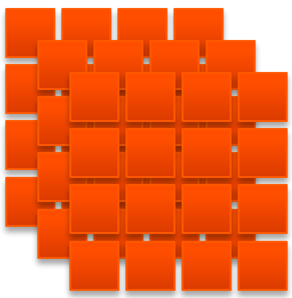


Optimization Notice

Containers

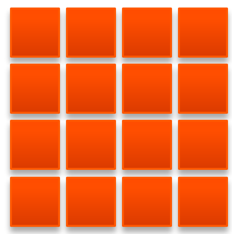
regular containers

dense<T>

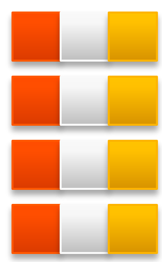


dense<T,3>

array<…>
struct user_type {..};
class user_type { };

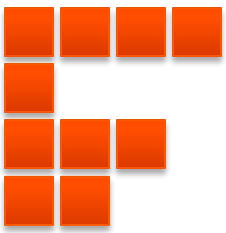


dense<T, 2>



dense<array<…>>
dense<user_type>

irregular containers

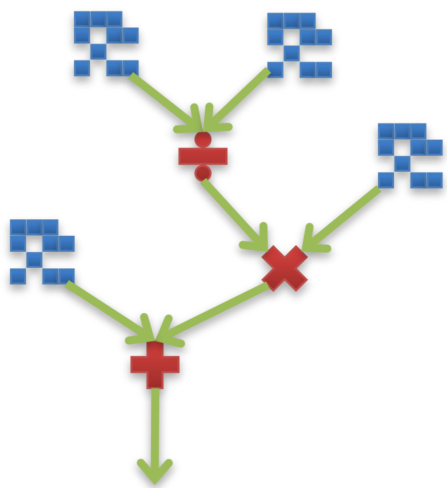


nested<T>

Optimization Notice

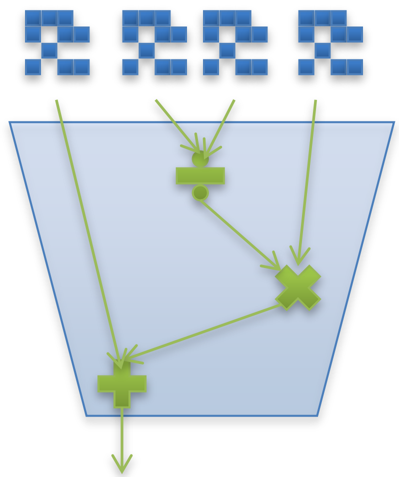
Vector Processing *or* Scalar Processing

Vector Processing



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

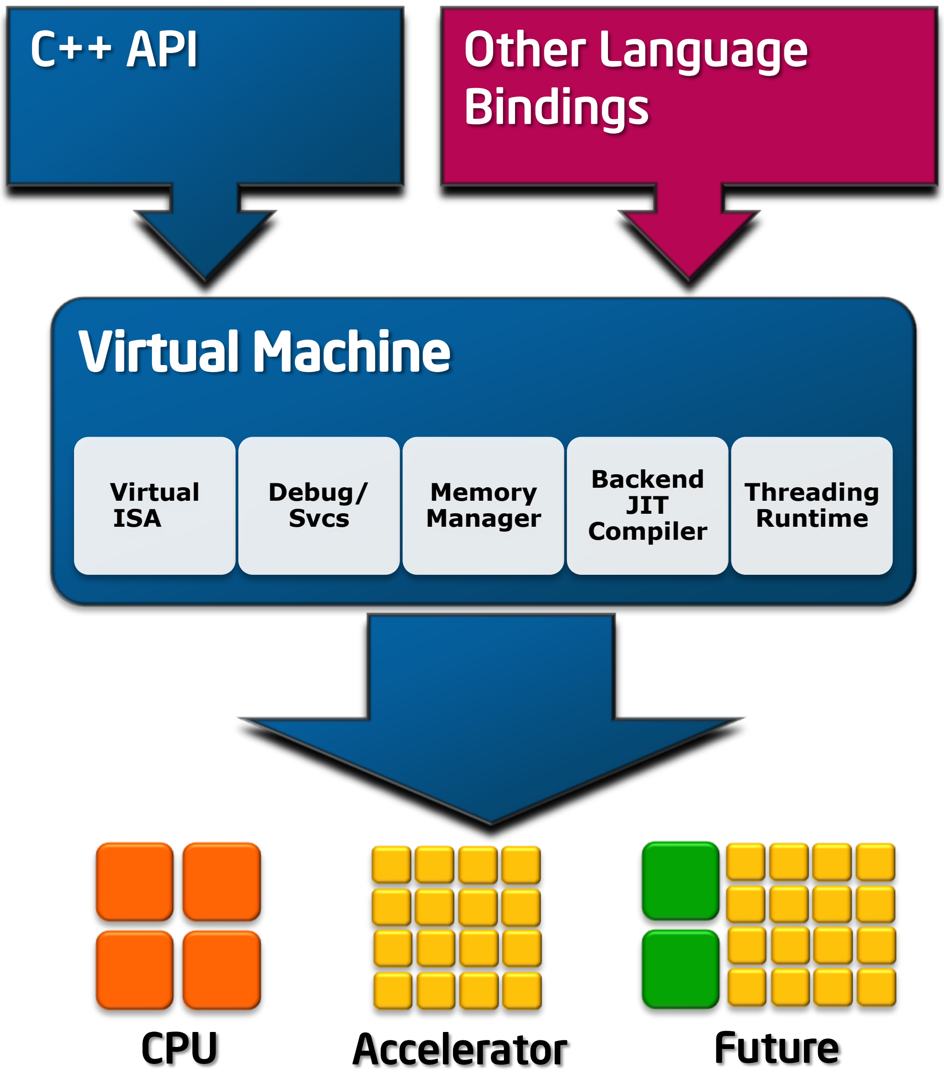
Scalar Processing



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

Intel® ArBB Virtual Machine

- Generalized data-parallel programming model
- Supports wide variety of patterns and collections
- Supports explicit dynamic generation and management of code
- Implementation targets both threads and vector code
  - Machine independent optimization
  - Offload management
  - Machine specific code generation and optimizations
  - Scalable threading runtime



Optimization Notice

Interface: The API as a Language

Syntax and semantics that extend C++

Adds parallel collection objects and methods to C++

- Uses standard C++ features (classes, simple templates, and operator overloading) to create new types and operators
- Sequences of API calls are fused and optimized by a JIT compiler

Works with standard C++ compilers

- Intel® C++ Compiler
- Microsoft* Visual* C++ Compiler
- GNU Compiler Collection*

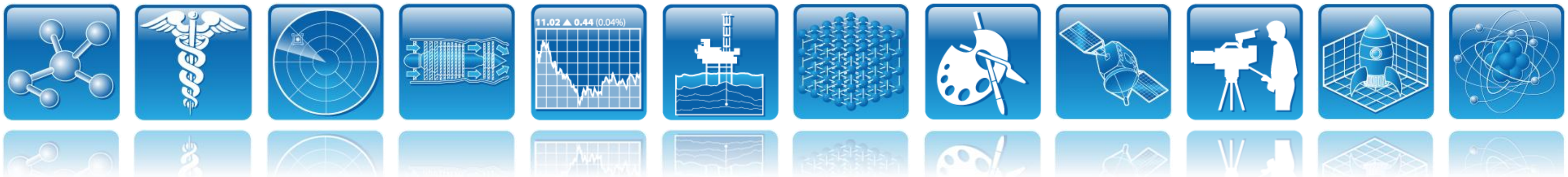
Express algorithms using mathematical notation

- Developers focus on *what to do,* **not** *how to do it*

Uses sequential semantics

- Developers do not use *threads, locks* or other lower-level constructs and can avoid the associated *complexity*
- ***Programmers can reason and debug as if the program were serial.***

What can it be used for?



Bioinformatics

- Genomics and sequence analysis
- Molecular dynamics

Engineering design

- Finite element and finite difference simulation
- Monte Carlo simulation

Financial analytics

- Option and instrument pricing
- Risk analysis

Oil and gas

- Seismic reconstruction
- Reservoir simulation

Medical imaging

- Image and volume reconstruction
- Analysis and computer aided detection (CAD)

Visual computing

- Digital content creation (DCC)
- Physics engines and advanced rendering
- Visualization
- Compression/decompression

Signal and image processing

- Computer vision
- Radar and sonar processing
- Microscopy and satellite image processing

Science and research

- Machine learning and artificial intelligence
- Climate and weather simulation
- Planetary exploration and astrophysics

Enterprise

- Database search
- Business information

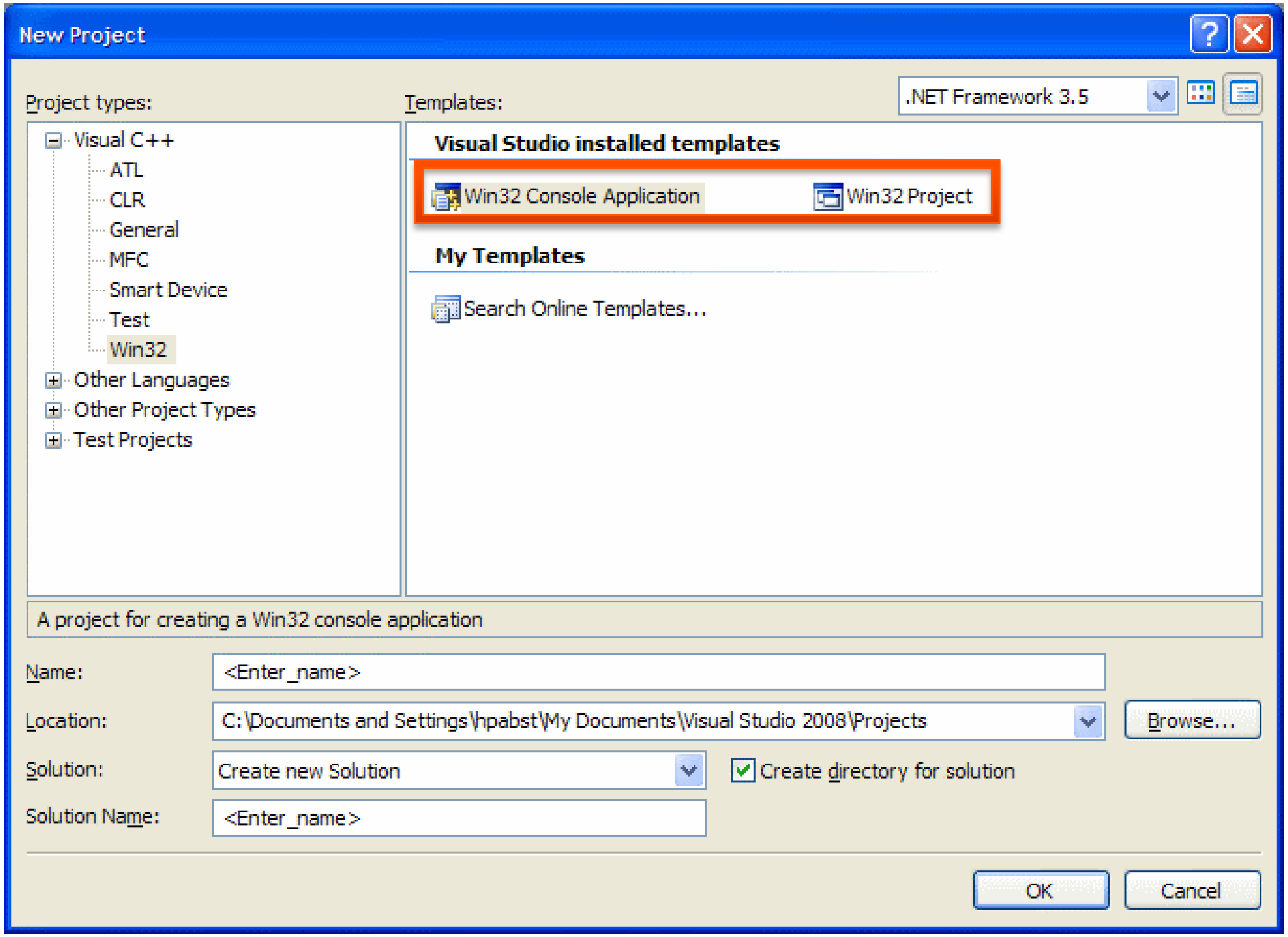
Optimization Notice

Introduction to Intel® Array Building Blocks

How to add it to your project…

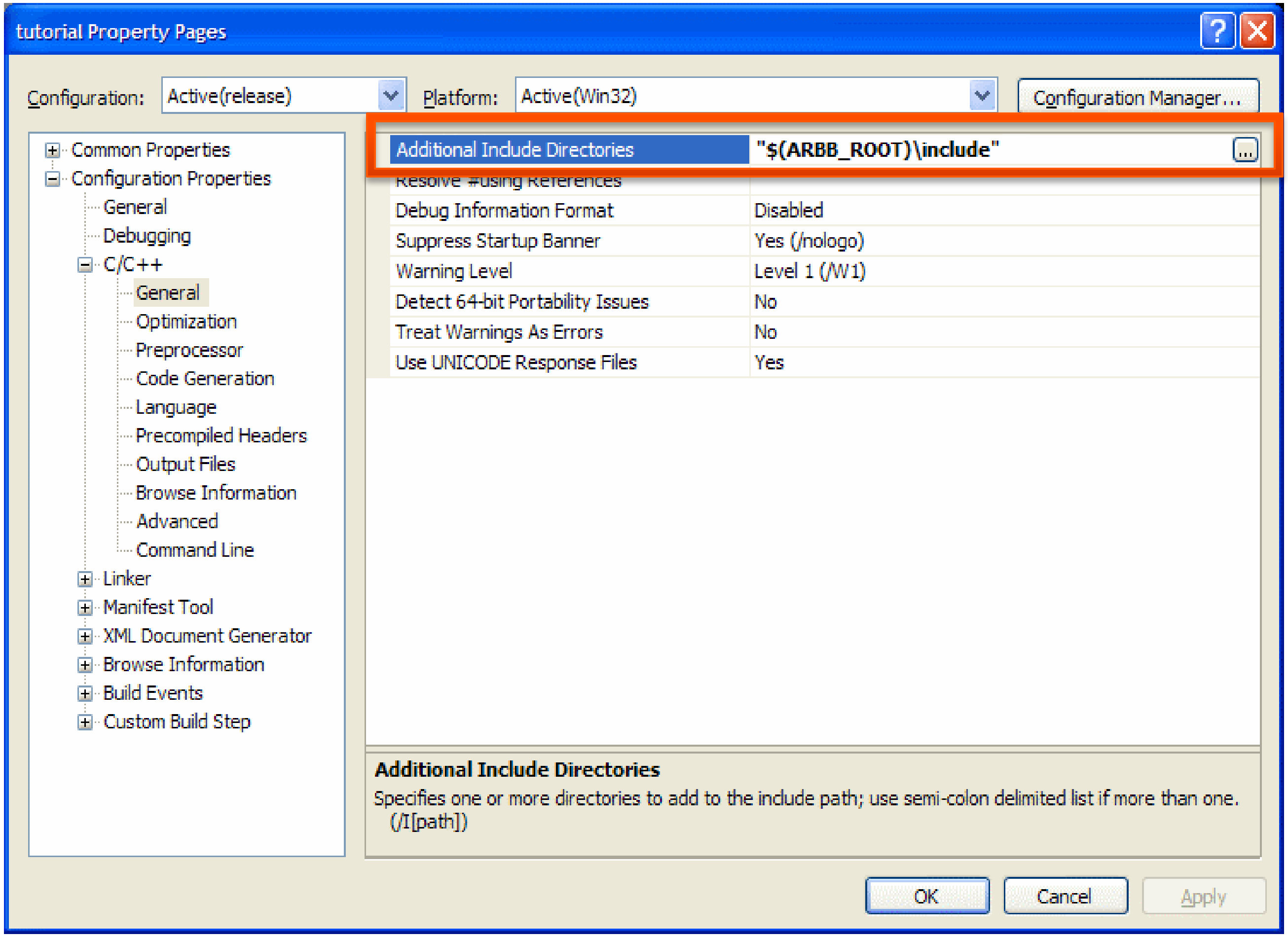
Optimization Notice

Intel® ArBB in a Visual Studio* Project



Screenshots taken from Microsoft* Visual Studio 2008*

Intel® ArBB in a Visual Studio* Project

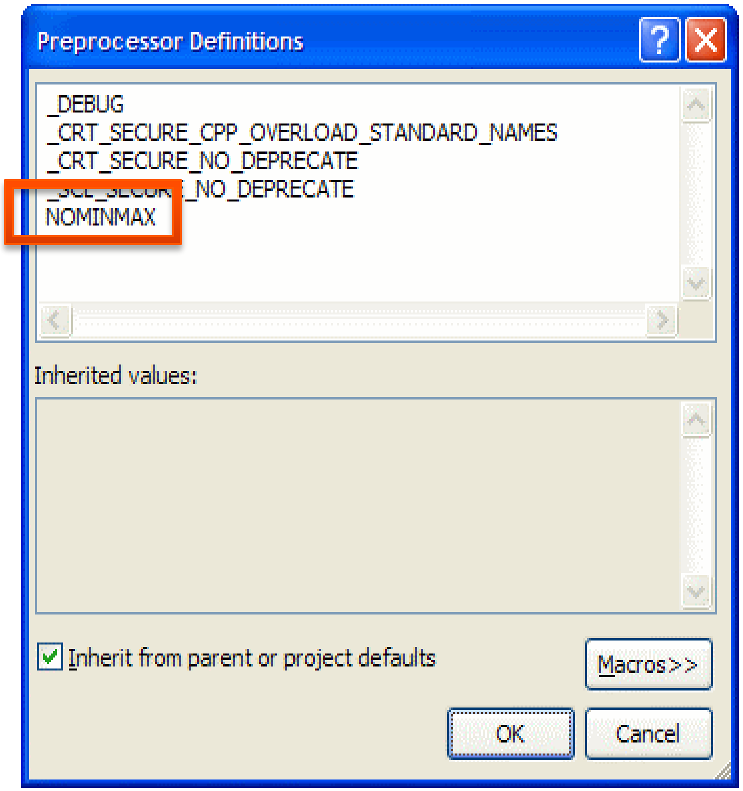


Screenshots taken from Microsoft* Visual Studio 2008*

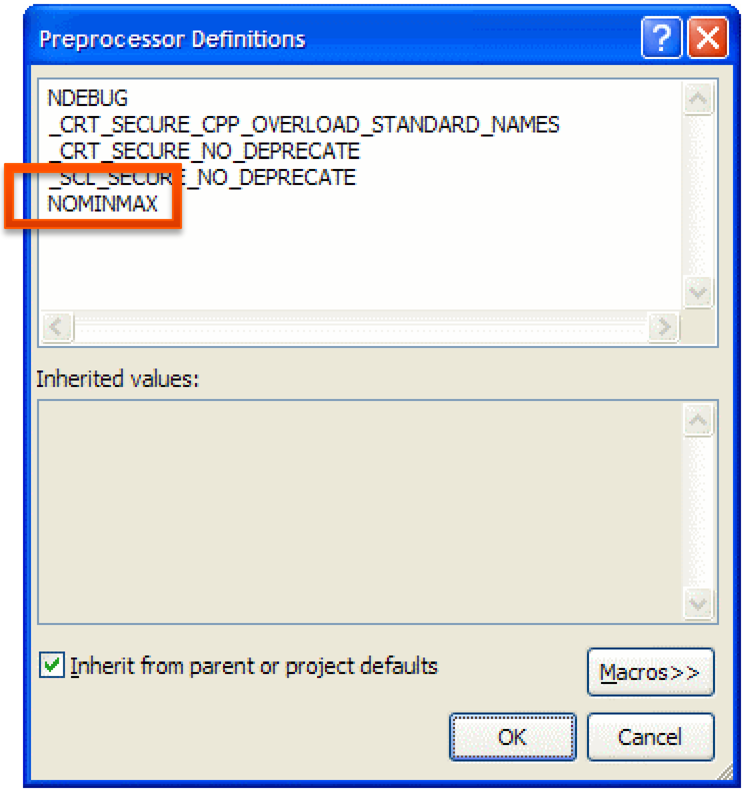
Optimization Notice

Including ArBB in a Visual Studio* Project

Debug Mode

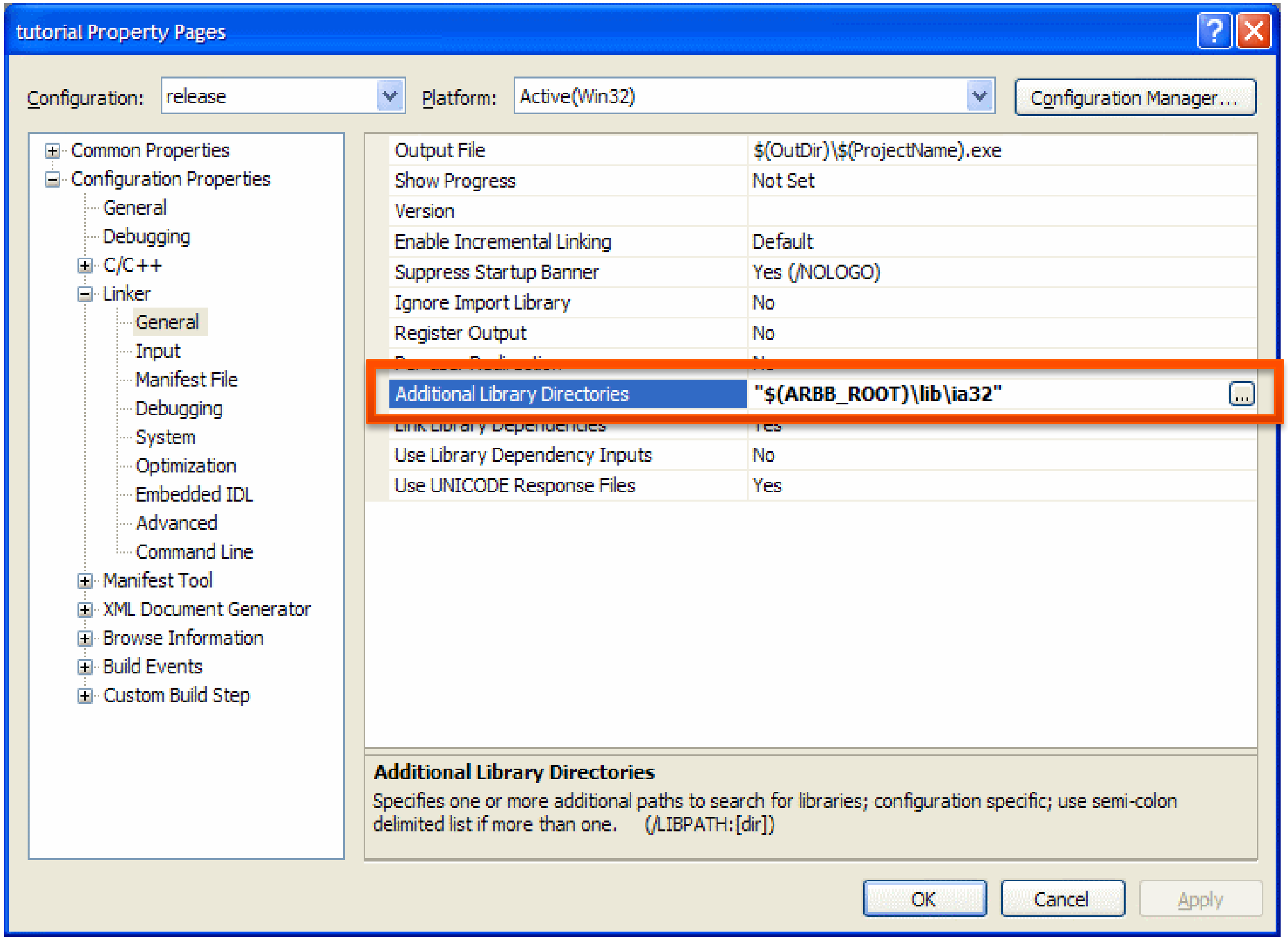


Release Mode



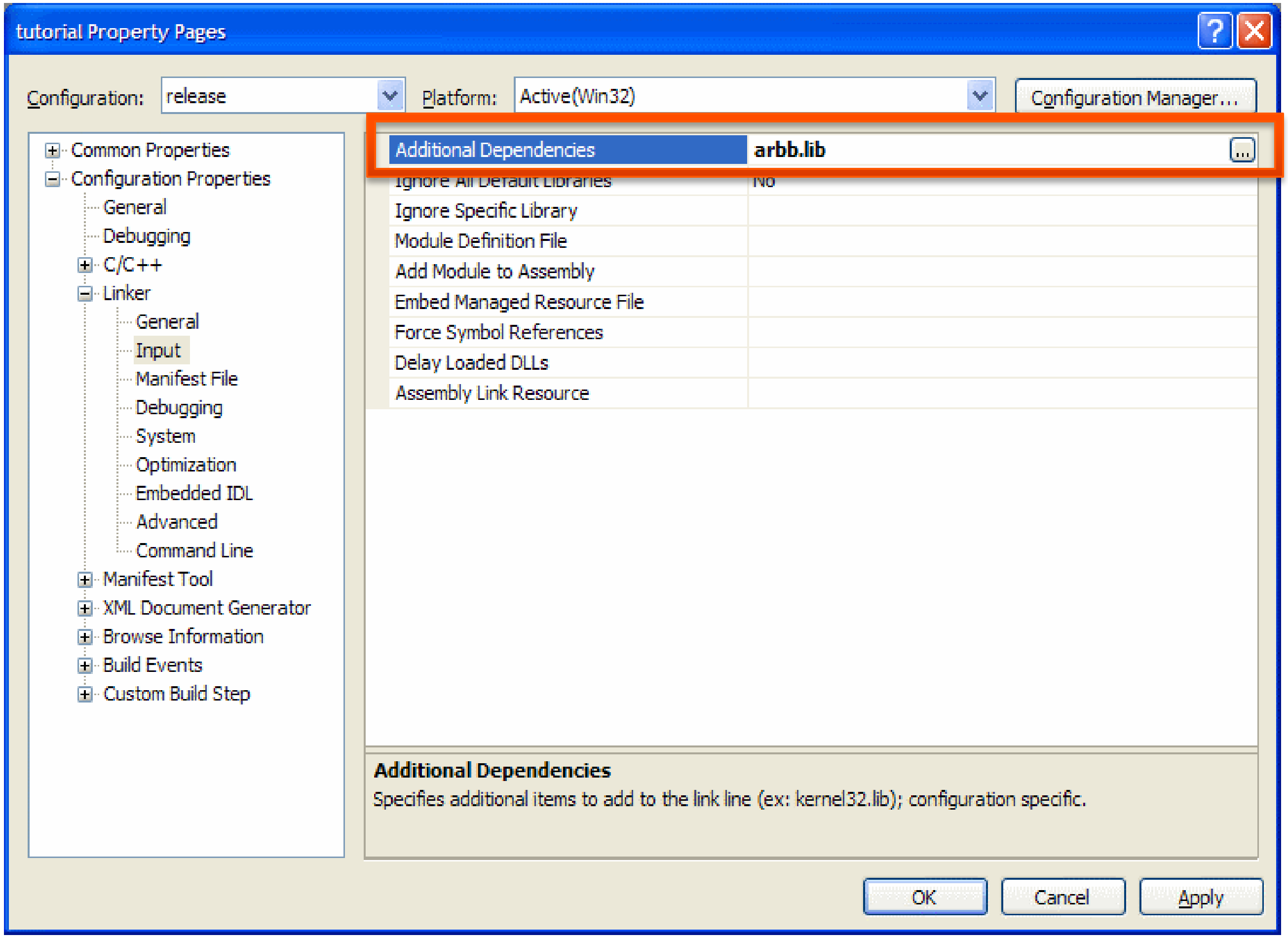
Screenshots taken from Microsoft* Visual Studio 2008*

Intel® ArBB in a Visual Studio* Project



Screenshots taken from Microsoft* Visual Studio 2008*

Intel® ArBB in a Visual Studio* Project



Screenshots taken from Microsoft* Visual Studio 2008*

Optimization Notice

Intel® ArBB in a Makefile-based Project

- **Make available ArBB include (header) files:**

  `-I/opt/intel/arbb/include`

  (modify compiler search path for include files)

- **Make available ArBB libraries**

  `-L/opt/intel/arbb/lib/{ia32,intel64}`

  (modify linker search path for libraries)

- **Include ArBB libraries in linker process**

  `-larbb -ltbb`

Optimization Notice

Using the Intel® ArBB API

- Include the definitions

```
#include <arbb.hpp>
```

- Import the namespace or specific identifiers

```
using namespace arbb;
using namespace arbb::add_reduce;
```

- Good practice:
  - To not pollute the name spaces, restrict scope of "using" statement as much as possible, especially in headers
  - Selectively include ArBB names only if used

Optimization Notice

Code Skeleton for Intel® ArBB Applications

- Use the following code skeleton for ArBB applications

```
int main(int argc, char* argv[]) {
    int ret_code;
    try {
        // call into ArBB code
        ret_code = EXIT_SUCCESS;
    }
    catch(const std::exception& e) {
        ret_code = EXIT_FAILURE;
    }
    catch(...) {
        cerr << "Error: Unknown exception caught!" << endl;
        ret_code = EXIT_FAILURE;
  }
  return ret_code;
}
```

- ArBB indicates runtime errors through standard C++ exceptions
- Existing top-level entry points do not need to change if they already catch std::exception

Optimization Notice

Introduction to Intel® Array Building Blocks

Programming Constructs and Data Types

Optimization Notice

Overall Syntax Conventions

- **All Identifiers are lower-case with underscores**
  - some_type
  - some_class::some_member_function()
- **Chosen to align with C++ standard library conventions**

Intel® ArBB Constructs

- **Scalar types**
  - Equivalent to primitive C++ types
- **Vector types**
  - Parallel collections of (scalar) data
- **Operators**
  - Scalar operators
  - Vector operators
- **Functions**
  - User-defined code fragments
- **Control flow constructs**
  - Conditionals, iteration, etc.
  - These are for *serial* control flow *only*
  - Vector operations and "map" are used for expressing parallelism

Optimization Notice

Scalar types

- **Scalar types provide equivalent functionality to the scalar types built into C/C++**

Types	Description	C++ equivalents
f32, f64	32/64 bit floating point number	float, double
i8, i16, i32, i64	8/16/32 bit signed integers	char, short, int
u8, u16, u32, u64	8/16/32 bit unsigned integers	unsigned char/short/int
boolean	Boolean value (true or false)	bool
usize, isize	Signed/unsigned integers sufficiently large to store addresses	size_t (eqv. usize)

Scalar Types

Use scalar types for ArBB scalar computation

```
i32 int_scalar;                     // a scalar 32-bit integer value
f32 fp_scalar = (f32)int_scalar;    // cast a scalar to new type
```

Casting to/from C/C++ types

```
float f = (float)fp_scalar;         // NOT supported
f32 fp_scalar2(f);                  // immediate copy
f32 fp_scalar3 = f;                 // immediate copy
float x = value(fp_scalar);         // retrieve value
```

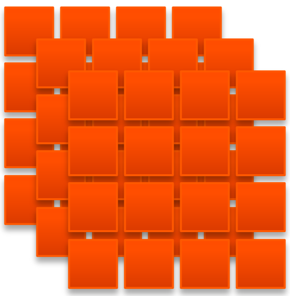
Constant values are supported (types must match)

```
f32 fp_scalar = (f32)int_scalar + 0.5f;
f32 r = 2.0f;
fp_scalar = 3.14f * r * r;
```

Containers

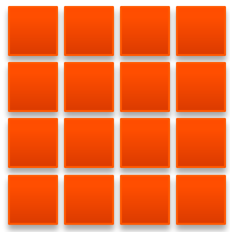
regular containers

dense<T>

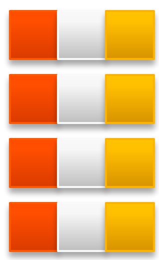


dense<T,3>

array<...>
struct user_type {..};
class user_type { };

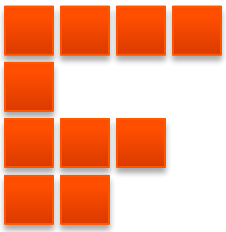


dense<T, 2>



dense<array<...>>
dense<user_type>

irregular containers



nested<T>

Optimization Notice

Dense Containers

```
template<typename T, std::size_t D = 1>
class dense;
```

- **This is the equivalent to `std::vector` or C arrays**
- **Dimensionality is optional, defaults to 1**

Property	Restrictions	Can be set at
Element type	Must be an ArBB scalar or user-defined type	Compile time
Dimensionality	1, 2, or 3	Compile time
Size	Only restricted by free memory	Runtime

Optimization Notice

Declaration and Construction

Declaration	Element type	Dimensionality	Size
dense<f32> a1;	f32	1	0
dense<f32, 1> a2;	f32	1	0
dense<i32, 2> b;	i32	2	0, 0
dense<f32> c(1000);	f32	1	1000
dense<f32> d(c);	f32	1	1000
dense<i8, 3> e(5, 3, 2);	i8	3	5, 3, 2

Operations on dense Containers

- **All scalar operations can be applied element-wise**
  - Arithmetic and bit operations, transcendentals, etc.
- **Additionally provides container operations:**
  - Indexing, e.g. operator[]
  - Reordering, e.g. shift(), section()
  - Reductions, e.g. sum(), any(), all()
  - Prefix sums, packs, and other data-parallel primitives
  - Property access, e.g. num_rows()
- **Most of these operations run in parallel**
  - For example, if you add two dense containers together, all the individual additions can run in parallel

Moving Data into and out of Containers

- **Dense containers provide two ways to access data:**
  - Iterators
    - read_only_range — iterator to read from the container
    - write_only_range — iterator to write into the container
    - read_write_range — iterator to write/read a container
  - Binding
    - On construction, dense containers can be *bound* (associated) to a particular data location
    - Moves data into and out of that location when required

Creating “dense” Containers

Declaration of a dense container:

```
// create an empty container whose values will be assigned later
dense<f32> temp;
```

vector objects of different base types cast into each other:

```
dense<i32> vi = …;
dense<f32> v = (dense<f32>)vi;
```


Optimization Notice

Filling "dense" Containers

```
// request write-only access to container
dense<f32> a(1024);
range<f32> range_a = a.write_only_range();
std::fill(range_a.begin(),
          range_a.end(),
          static_cast<f32>(1));

// request read/write access to container
dense<f32> b(1024);
range<f32> range_b = b.read_write_range();
std::fill(range_b.begin(),
          range_b.end(),
          static_cast<f32>(2));
```

Fixed-size Arrays

- Typical usages: pairs of data, RGBA data, CYMK data, etc.
- Use std::array look-a-like
  - std::array is a C++ TR1/C++0x type
  - Will support std::array operations
  - You can manipulate with element-wise, horizontal, swizzling, and other utility operations

```
array<f32, 3> p1, p2, p3;

f32 r = p1[0];                  // std::array operations
p1 = p2 + p3;                   // element-wise operations
f32 sum_p1 = sum(p1);           // horizontal operations
p1 = cat(p2, p3);               // utility operations
```


Optimization Notice

Structured Types

- C++ classes and structures can be used relatively normally within ArBB
  - Requires that primitive types be classes in ArBB types (f32, etc.)
  - Supports member functions, class members, overloaded operators, etc.
  - However, virtual functions and pointers are resolved during "capture time" only
  - Overloaded operators are automatically lifted over collections
  - Lifting member functions over collections requires an additional declaration (a macro is provided to help with this)

Structure/Class Example

```
class my_class {
public:
  my_class(f32 location, i32 count);
  my_class operator+(const my_class& other) {
    return my_class(location + other.location,
                    max(count, other.count));
  }
  // other code…
private:
  f32 m_location;
  i32 m_count;
};

dense<my_class> A, B, C;
A = B + C;           // This will use the user-defined operator+!
my_class m = A[5];  // Other interactions work naturally.
```


Optimization Notice

A First Example: Vector Addition

Plain C version

```
void vecsum(float* a, float* b, float* c, int size) {
    for (int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}

int main(int argc, char** argv) {
#define SIZE = 1024;
    float a[SIZE]; float b[SIZE];
    float c[SIZE];

    vecsum(a, b, c, SIZE);
}
```

Add two vectors `a` and `b` of length `SIZE` into vector `c`.

Step 1: Figure out Kernel Signature

```
void vecsum(float* a, float* b,
            float* c, int size)
{
    for (int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}

int main(int argc, char** argv) {
#define SIZE = 1024;
    float a[SIZE]; float b[SIZE];
    float c[SIZE];

    vecsum(a, b, c, SIZE);
}
```

```
void vecsum(dense<f32> a,
            dense<f32> b,
            dense<f32>& c) {

}

int main(int argc, char** argv) {
#define SIZE = 1024;
    float a[SIZE]; float b[SIZE];
    float c[SIZE];

}
```


Optimization Notice

Step 2: Allocate, Size, and Bind Containers

```
void vecsum(float* a, float* b,
            float* c, int size)
{
    for (int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}

int main(int argc, char** argv) {
#define SIZE = 1024;
    float a[SIZE]; float b[SIZE];
    float c[SIZE];



    vecsum(a, b, c, SIZE);
}
```

```
void vecsum(dense<f32> a,
            dense<f32> b,
            dense<f32>& c) {



}

int main(int argc, char** argv) {
#define SIZE = 1024;
    float a[SIZE]; float b[SIZE];
    float c[SIZE];

    dense<f32> va; bind(va, a, SIZE);
    dense<f32> vb; bind(vb, b, SIZE);
    dense<f32> vc; bind(vc, c, SIZE);


}
```

Step 3: Invoke Kernel Through Call

```
void vecsum(float* a, float* b,
            float* c, int size)
{
    for (int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}

int main(int argc, char** argv) {
#define SIZE = 1024;
    float a[SIZE]; float b[SIZE];
    float c[SIZE];

    vecsum(a, b, c, SIZE);
}
```

```
void vecsum(dense<f32> a,
            dense<f32> b,
            dense<f32>& c) {

}

int main(int argc, char** argv) {
#define SIZE = 1024;
    float a[SIZE]; float b[SIZE];
    float c[SIZE];

    dense<f32> va; bind(va, a, SIZE);
    dense<f32> vb; bind(vb, b, SIZE);
    dense<f32> vc; bind(vc, c, SIZE);

    call(vecsum)(va, vb, vc);
}
```


Optimization Notice

Step 4: Implement Kernel

```
void vecsum(float* a, float* b,
            float* c, int size)
{
    for (int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}

int main(int argc, char** argv) {
#define SIZE = 1024;
    float a[SIZE]; float b[SIZE];
    float c[SIZE];

    vecsum(a, b, c, SIZE);
}
```

```
void vecsum(dense<f32> a,
            dense<f32> b,
            dense<f32>& c) {

    c = a + b;

}

int main(int argc, char** argv) {
#define SIZE = 1024;
    float a[SIZE]; float b[SIZE];
    float c[SIZE];

    dense<f32> va; bind(va, a, SIZE);
    dense<f32> vb; bind(vb, b, SIZE);
    dense<f32> vc; bind(vc, c, SIZE);

    call(vecsum)(va, vb, vc);
}
```


Optimization Notice

Introduction to Intel® Array Building Blocks

Operators

Optimization Notice

Intel® ArBB Operators

- ArBB operators can be categorized into the following classes:
  - Element-wise:
    - Apply the same operation to all elements of a vector, or to the corresponding elements of a set of vectors
  - Vector-scalar:
    - Promote a scalar to a vector by replication, then apply element-wise operations
  - Collectives:
    - Output depends on the entire input vector
    - Ex: reduce a vector to a single, scalar value, e.g. via summation.
  - Permutation operators:
    - Reorganize elements of a vector
  - Facility functions:
    - Generic data access

Element-wise & Vector-scalar Operators

- Arithmetic operators:
  - +, +=, ++ (prefix and postfix), — addition, increment
  - -, -=, -- (prefix and postfix), — subtraction, decrement
  - *, *=, — multiplication
  - /, /=, — division
  - %, %= — modulo
- Bitwise operators:
  - &, &=, — bitwise AND
  - |, |=, — bitwise OR
  - ^, ^=, — bitwise XOR
  - ~, ~=, — bitwise NOT
  - <<, <<=, — shift left
  - >>, >>= — shift right
- Logical / comparison operators:
  - ==, !=, — equals
  - >, >=, — greater than
  - <, <=, — less than
  - &&, ||, ! — logical AND/OR/NOT

Optimization Notice

Element-wise & Vector-scalar Operators

- **Unary operators:**

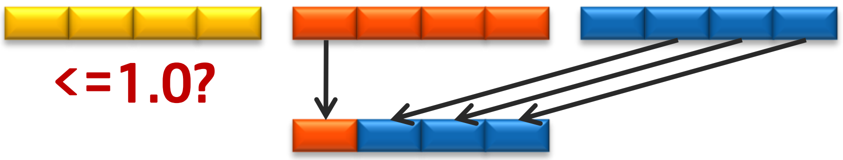
Operator	Description	Operator	Description
abs	absolute value	log	natural logarithm
acos	arccosine	rcp	reciprocal
asin	arcsine	round	round to nearest integer
atan	arctangent	rsqrt	reciprocal square root
ceil	round towards infinity	sin	sine
cos	cosine	sinh	hyperbolic sine
cosh	hyperbolic cosine	sqrt	square root
exp	exponent	tan	tangent
floor	round towards neg. infinity	tanh	hyperbolic tangent
log10	common logarithm		

Optimization Notice

Element-wise & Vector-scalar Operators

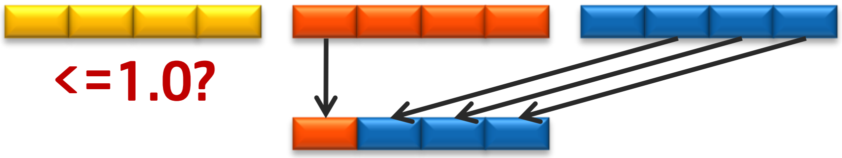
Operator	Description
atan2	arctangent
clamp	compare and cut at lower/upper bound
max	element-wise maximum
min	element-wise minimum
pow	power
select	"cond ? x : y" for each element of x and y

a = select(b <= 1.0, c, d)



Element-wise & Vector-scalar Operators

a = select(b <= 1.0, c, d)



Note:
We have to use this syntax since the current C++ standard does not allow overloading of the ?: operator.

Collective Operators

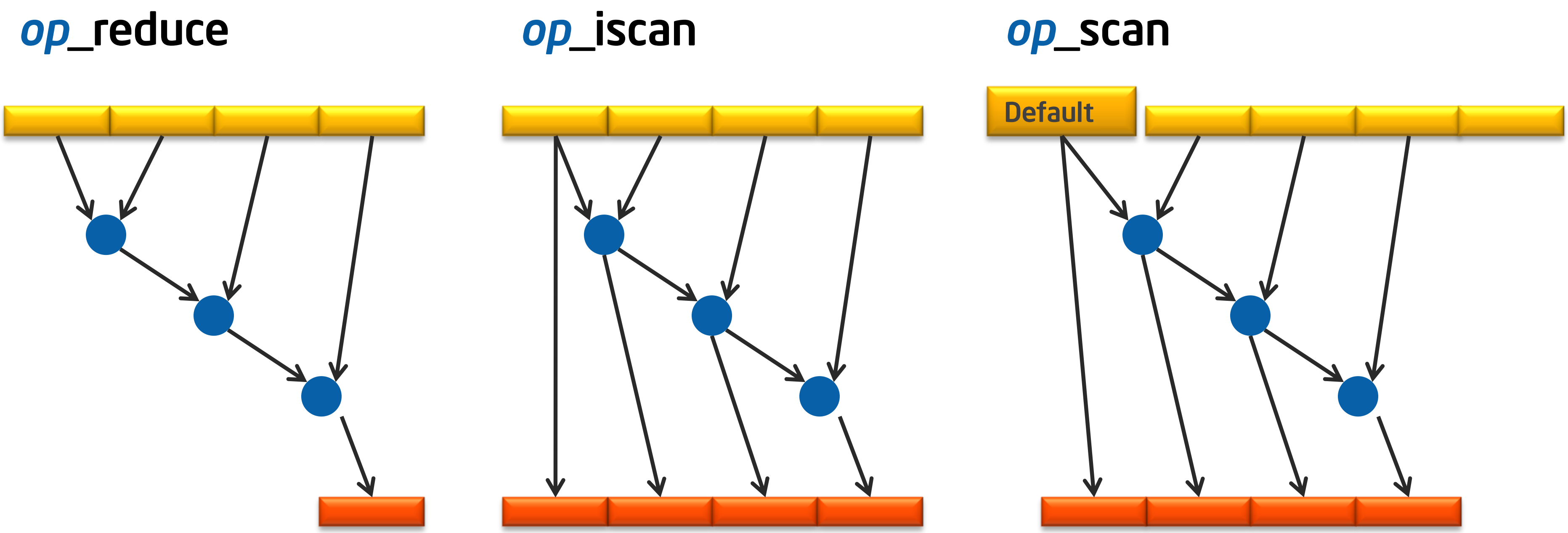
- Computations over entire vectors.
  - The output(s) can in theory depend on all the inputs
- 2 kinds of collective primitives:
  - Reductions apply an operator over an entire vector to compute a distilled value (or values depending on the type of vector):

    `add_reduce([1 0 2 -1 4])` yields
    `6`
  - Scans compute reductions on all prefixes of a collection, either *inclusively* or *exclusively*:

```
add_iscan([1 0 2 -1 4]) yields
[1 (1+0) (1+0+2) (1+0+2+(-1)) (1+0+2+(-1)+4)]
[1  1     3       2              6]
add_scan([1 0 2 -1 4]) yields
[0  1     (1+0)   (1+0+2)        (1+0+2+(-1))]
[0  1     1       3              2]
```


Optimization Notice

Collective Operators



Collective Operators

Reductions

Operator	Description
add_reduce	add all elements
sum	add over all dimensions
and_reduce	logical AND all elements
all	AND over all dimensions
mul_reduce	multiply all elements
ior_reduce	logical OR on all elements
any	OR over all dimensions
max_reduce	maximum of all elements
min_reduce	minimum of all elements
xor_reduce	XOR on all elements

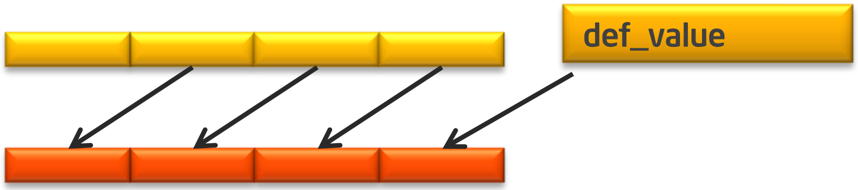
NOTE: "*_reduce" operations on multidimensional collections operate a dimension at a time.

Scans

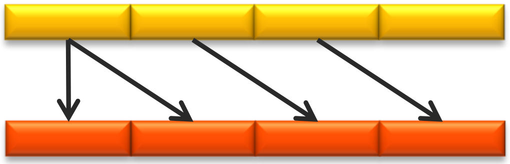
Operator	Description
add_scan	prefix sum
add_iscan	inclusive prefix sum
and_scan	prefix logical and
and_iscan	inclusive prefix logical and
ior_scan	prefix logical or
ior_iscan	inclusive prefix logical or
max_scan	prefix maximum
max_iscan	inclusive prefix maximum
min_scan	prefix minimum
min_iscan	inclusive prefix minimum
mul_scan	prefix multiply
mul_iscan	inclusive prefix multiply
xor_scan	prefix exclusive-or
xor_iscan	inclusive prefix exclusive-or

Permutation Operators

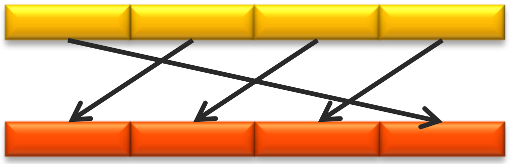
```
a = shift(b, -1, def_value);
```
// right if positive; left if negative




```
a = shift_sticky(b, 1);
```
// shift with duplicated boundary value



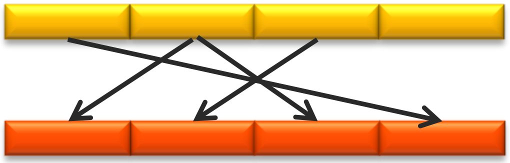
```
a = rotate(b, -1);
```
// shift with rotated boundary values



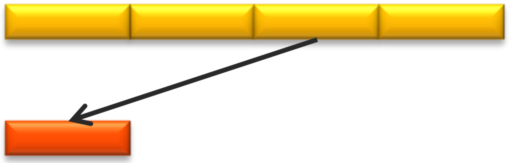
Optimization Notice

Permutation Operators

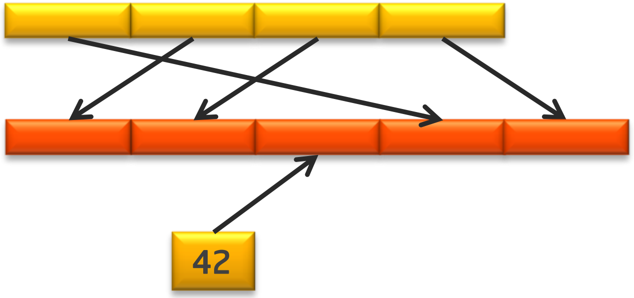
```
a = b[ {1,2,1,0} ];
a = gather(b, {1,2,1,0})
```



```
x = b[2];
x = gather(b, 2);
```

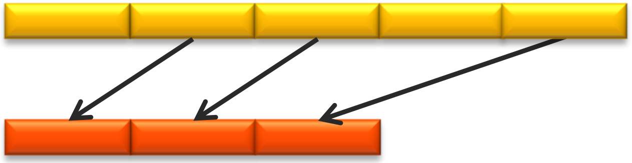


```
a = scatter(b, {3,0,1,4}, 5, 42);
```

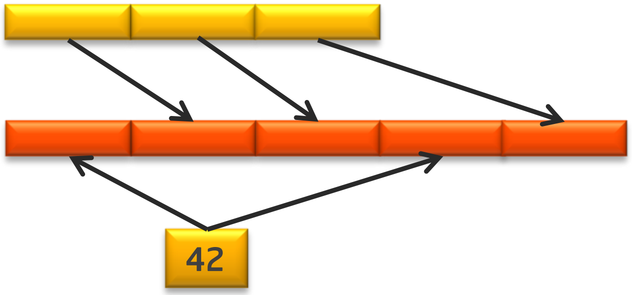


Permutation Operators

```
a = pack(b, {0, 1, 1, 0, 1});
```



```
a = unpack(b, {0, 1, 1, 0, 1}, 42);
```



Facility Functions

- **Facility function provide data processing features**

Operator	Dim	Description
cat	1, 2, 3	concatenate dense containers
()	1, 2, 3	positional access of a scalar value
row	2, 3	retrieve row of a dense container
col	2, 3	retrieve column of a dense container
section	1, 2, 3	retrieve sub-section of a dense container
replace	1, 2, 3	replace sub-section of a dense container
replace_row	2, 3	replace row of a dense container
replace_col	2, 3	replace column of a dense container
page	3	retrieve slice of a dense container
replace_page	3	replace slice of a dense container

Introduction to Intel® Array Building Blocks

Control Flow Constructs

Optimization Notice

Loops

For loop

```
_for (begin, end, step) {    // note use of commas, not semicolons!
    /* code */
} _end_for;                  // note use of termination keyword
```

Example

```
_for (i32 i=0, i<=N, i++) {
    /* code */
} _end_for;
```

All loop constructs in ArBB, including _for, are used to describe *serial* control flow that depend on dynamically computed data (that is, values computed by ArBB types).

THEY DO NOT THEMSELVES EXPRESS PARALLELISM

Loops

While loop

```
_while (condition) {
    /* code */
} _end_while;
```

Supporting statements:

- Exit loop with `_break`
- Skip remainder of current iteration with `_continue`
- Return from Intel® ArBB function with `_return`

Conditionals

if statement

```
_if (condition){
      /* code */
} _end_if;
```

if statement with else

```
_if (condition){
      /* code */
}
_else {
      /* code */
} _end_if;
```

if statement with “else if”

```
_if (condition1){
      /* code */
}
_else_if (condition2) {
      /* code */
}
_else {
      /* code */
} _end_if;
```

Function Calls

Function calls

```
call(function_ptr)(arg1, arg2, ..., argn);
```

- Regular function call
- Transfers control from the caller to the callee

Applying functions to every element of a collection

```
map(function_ptr)(arg1, arg2, ..., argn);
```

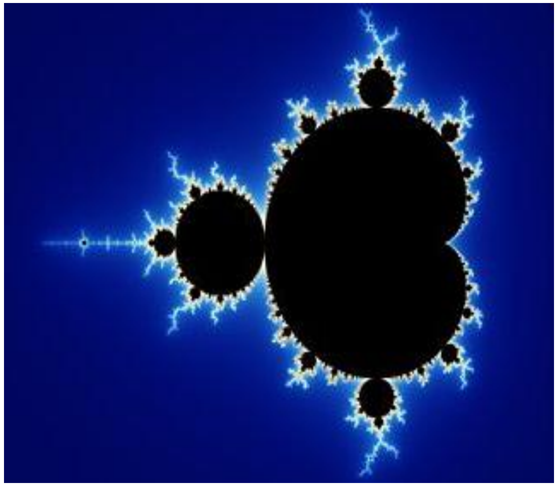
- Arguments should match formal type exactly OR be a collection with an element type that matches exactly
- Converts a scalar function into a parallel operation

Introduction to Intel® Array Building Blocks

Example: Mandelbrot using an Elemental Function

Example: Mandelbrot Set

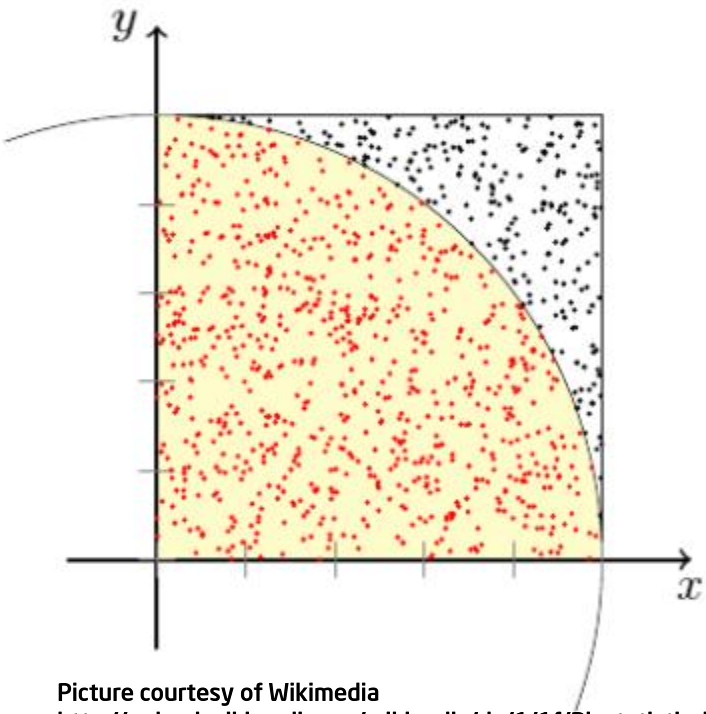
```
int max_count = . . .;
void mandel(i32& d, std::complex<f32> c) {
  i32 i;
  std::complex<f32> z = 0.0f;
  _for (i = 0, i < max_count, i++) {
    _if (abs(z) >= 2.0f) {
      _break;
    } _end_if;
    z = z*z + c;
  } _end_for;
  d = i;
}
void doit(dense<i32,2>& D, dense<std::complex<f32>,2> C)
{
  map(mandel)(D,C);
}

call(doit)(dest, pos);
```



Introduction to Intel® Array Building Blocks

Example: Monte Carlo with Vector Computation

Monte Carlo Computation of Pi



Picture courtesy of Wikimedia
http://upload.wikimedia.org/wikipedia/de/1/1f/Pi_statistisch.png

```
double computepi() {
  int cnt = 0;
  for(int i = 0; i < NEXP; i++) {
    float x = float(rand()) /
              float(RAND_MAX);
    float y = float(rand()) /
              float(RAND_MAX);
    float dst = sqrtf(x*x + y*y);
    if (dst <= 1.0f) {
      cnt++;
    }
  }
  return 4.0 *
          ((double) cnt) / NEXP;
}
```

Monte Carlo Computation of Pi (C/C++)

```
double computepi() {
  int cnt = 0;
  for(int i = 0; i < NEXP; i++) {
    float x = float(rand()) / float(RAND_MAX);
    float y = float(rand()) / float(RAND_MAX);
    float dst = sqrtf(x*x + y*y);
    if (dst <= 1.0f) {
      cnt++;
    }
  }
  return 4.0 * ((double) cnt) / NEXP;
}
```

Run NEXP experiments.

Pick a random x coordinate [0,1].

Pick a random y coordinate [0,1].

If distance from (0,0) is less than 1, count the needle as within the unit circle.

Monte Carlo Computation of Pi (Intel® ArBB)

```
void computepi(f64& pi) {
    random_generator rng;

    dense<f32> x = rng.randomize(NEXP);
    dense<f32> y = rng.randomize(NEXP);

    dense<f32> dist = sqrt(x*x + y*y);

    dense<boolean> mask = (dist <= 1.0f);

    dense<i32> cnt = select(mask, 1, 0);

    pi = 4.0 * add_reduce(cnt) / NEXP;
}
```

User-defined random number generator.

Initialize NEXP experiments and store coordinates in vectors.

Compute distance of (x,y) vectors.

Generate mask vector.

Map mask to vector with 1s and 0s.

“Count” elements with value 1.

Intel® Array Building Blocks Execution Engine

Objectives

- Understand the semantics of the ArBB execution model
- Understand how ArBB generates code
- Understand how to control the code generation process
- Understand ArBB's way of calling functions

Intel® Array Building Blocks Execution Engine

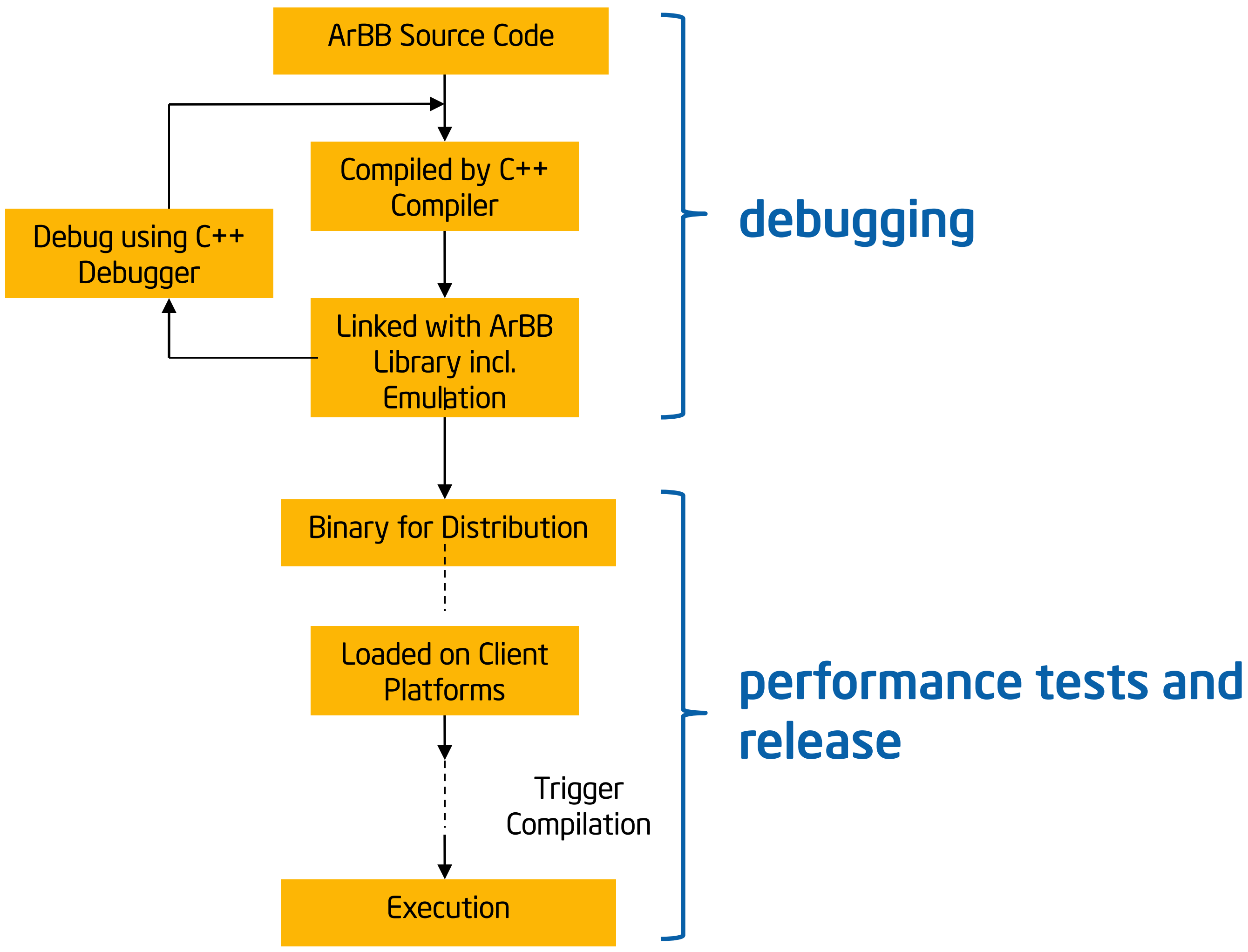
Execution Model

Intel® ArBB Execution Model

- **Container objects represent collections of data**
- **Vector operations and elemental functions represent a set of data-parallel operations that operate on these containers**
  - Container objects are passed by value or by const reference
  - Operator application logically returns a new container (single assignment)
  - Assignment always behaves “as if” data was copied into destination
    - but unnecessary copies are optimized away internally
- **ArBB programs can be compiled by any ISO-compatible C++ compiler**
  - object code is linked with the ArBB library
  - debugging through a standard C++ debugger

- **Binaries distributed as normal IA32/Intel64 applications**

Intel® ArBB Execution Model



ArBB Execution Model

- **Binaries are loaded on to the client's platform**
  – ArBB dynamic runtime triggers 2nd stage compilation (aka adaptive compilation)
  – Compilation is dependent on characteristics of target architecture

- **The ArBB dynamic execution model provides advantages**
  – ***Performance transparency***
    – Predictable performance to varying degrees of accuracy.
    – Translation of seemingly sequential and scalar based codes into highly efficient, SIMD-ized and parallelized codes, depending on the low-level architecture.
  – ***Forward scalability***
    – Architectural portability closely related to the requirements of forward-scaling multi-core applications.
    – Increase core count -> necessary for portable program execution models to deliver this advantage.

Intel® Array Building Blocks Execution Engine

The Runtime System

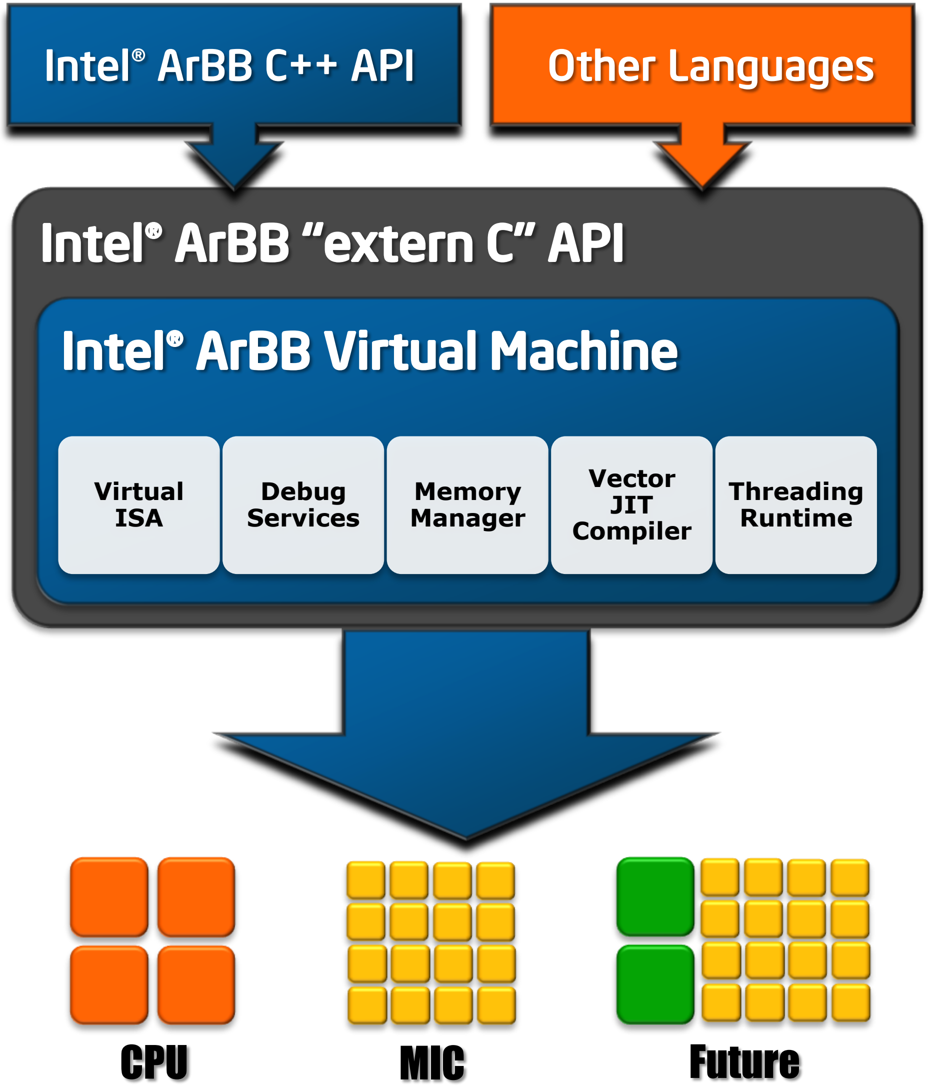
Intel® ArBB VM

Intel® ArBB has a high-level, standards compliant C++ interface to a Virtual Machine (VM)

Can be used with a broad range of ISO standard C++ compilers

VM both manages threads and dynamically generates optimized vector code

Code is *portable* across different SIMD widths and different core counts, *even in binary form*



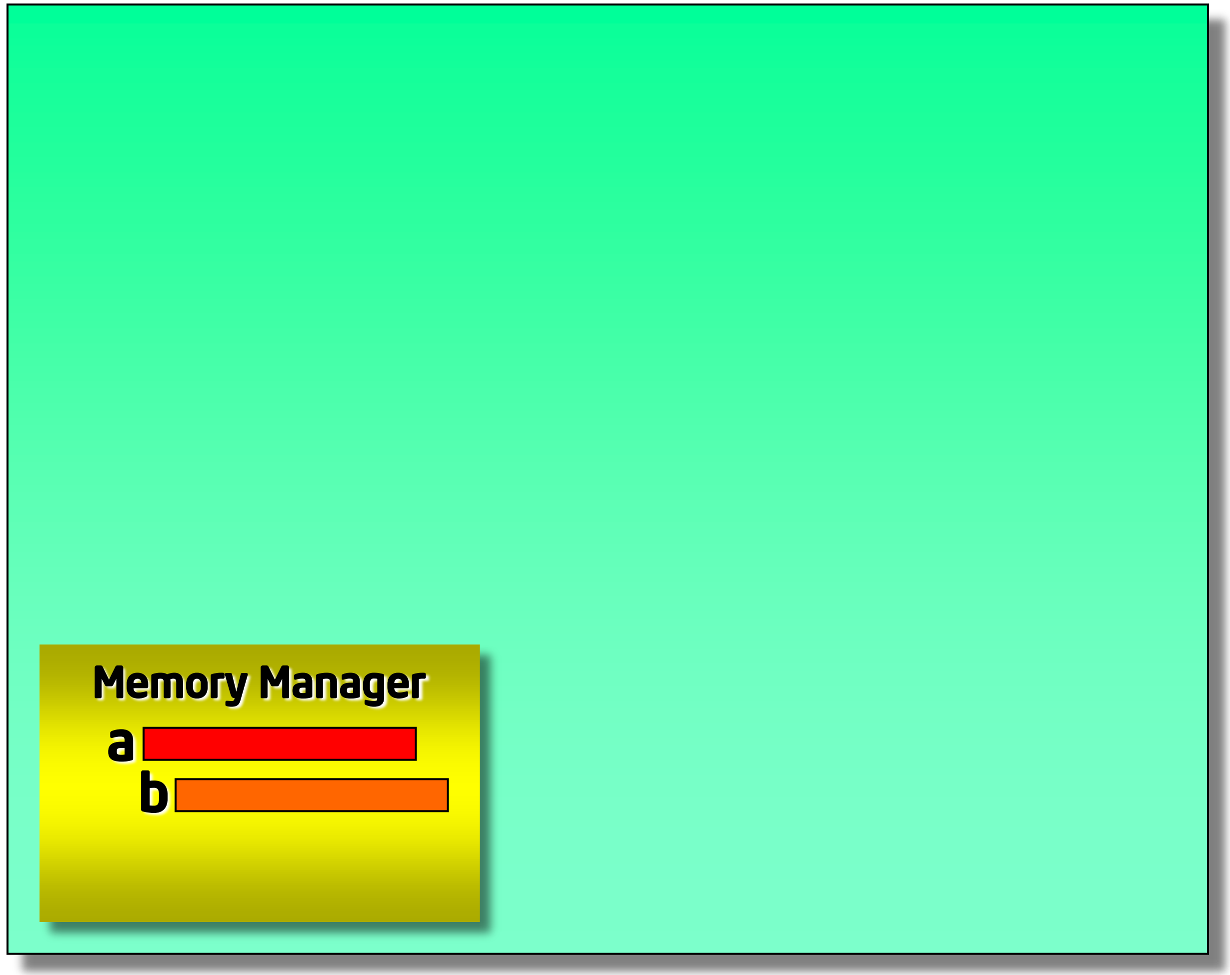
Intel® ArBB Dynamic Engine Execution

```
int ar_a[1024],
    ar_b[1024]
dense<i32> a();
bind(a, ar_a, 1024);
dense<i32> b();
bind(b, ar_b, 1024);
call(work)(a, b);
```

ArBB
Dynamic
Engine

Intel® ArBB Dynamic Engine Execution

```
int ar_a[1024],
    ar_b[1024]
dense<i32> a();
bind(a, ar_a, 1024);
dense<i32> b();
bind(b, ar_b, 1024);
call(work)(a, b);
```

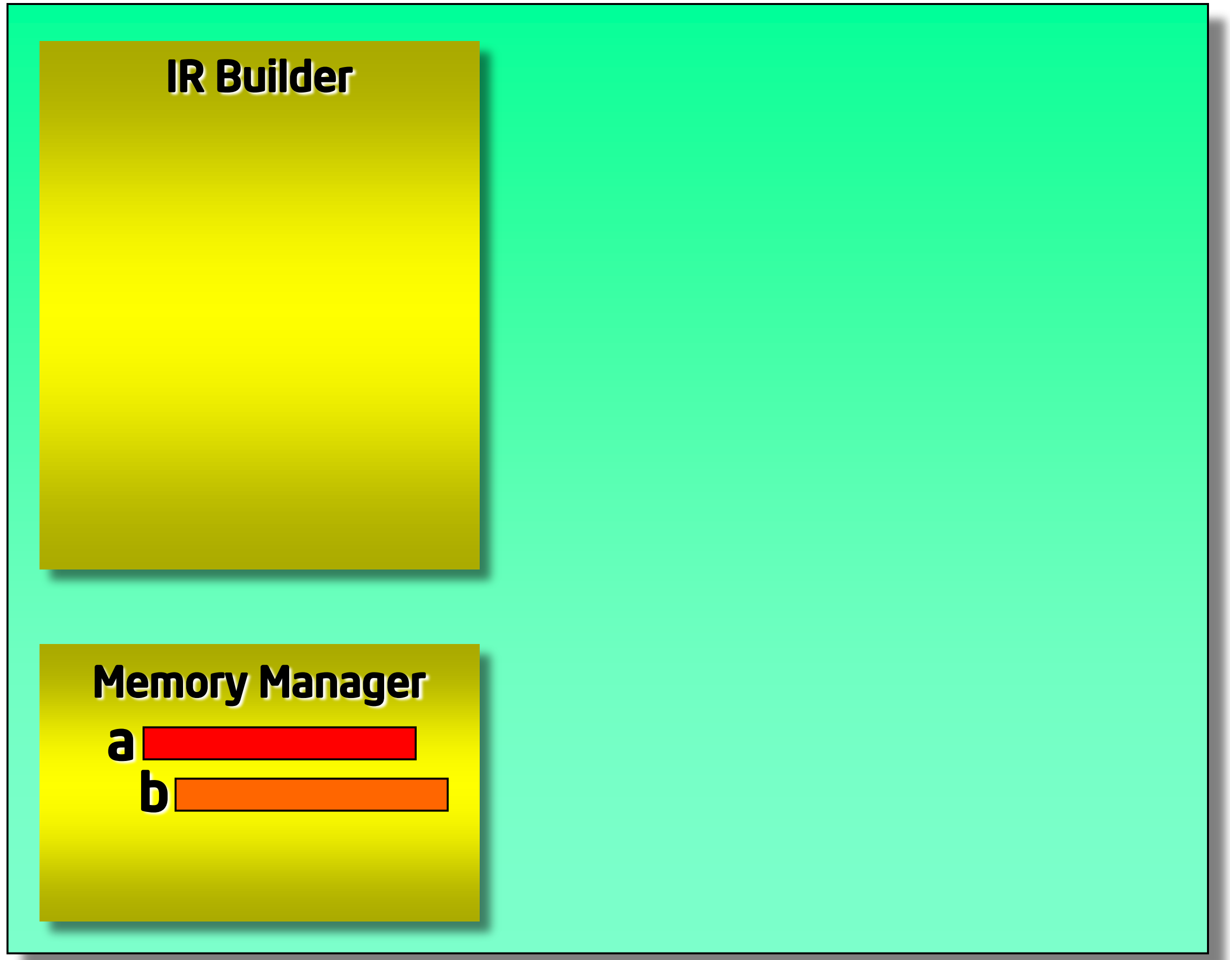


ArBB
Dynamic
Engine

Intel® ArBB Dynamic Engine Execution

```
int ar_a[1024],
    ar_b[1024]
dense<i32> a();
bind(a, ar_a, 1024);
dense<i32> b();
bind(b, ar_b, 1024);
call(work)(a, b);
```

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



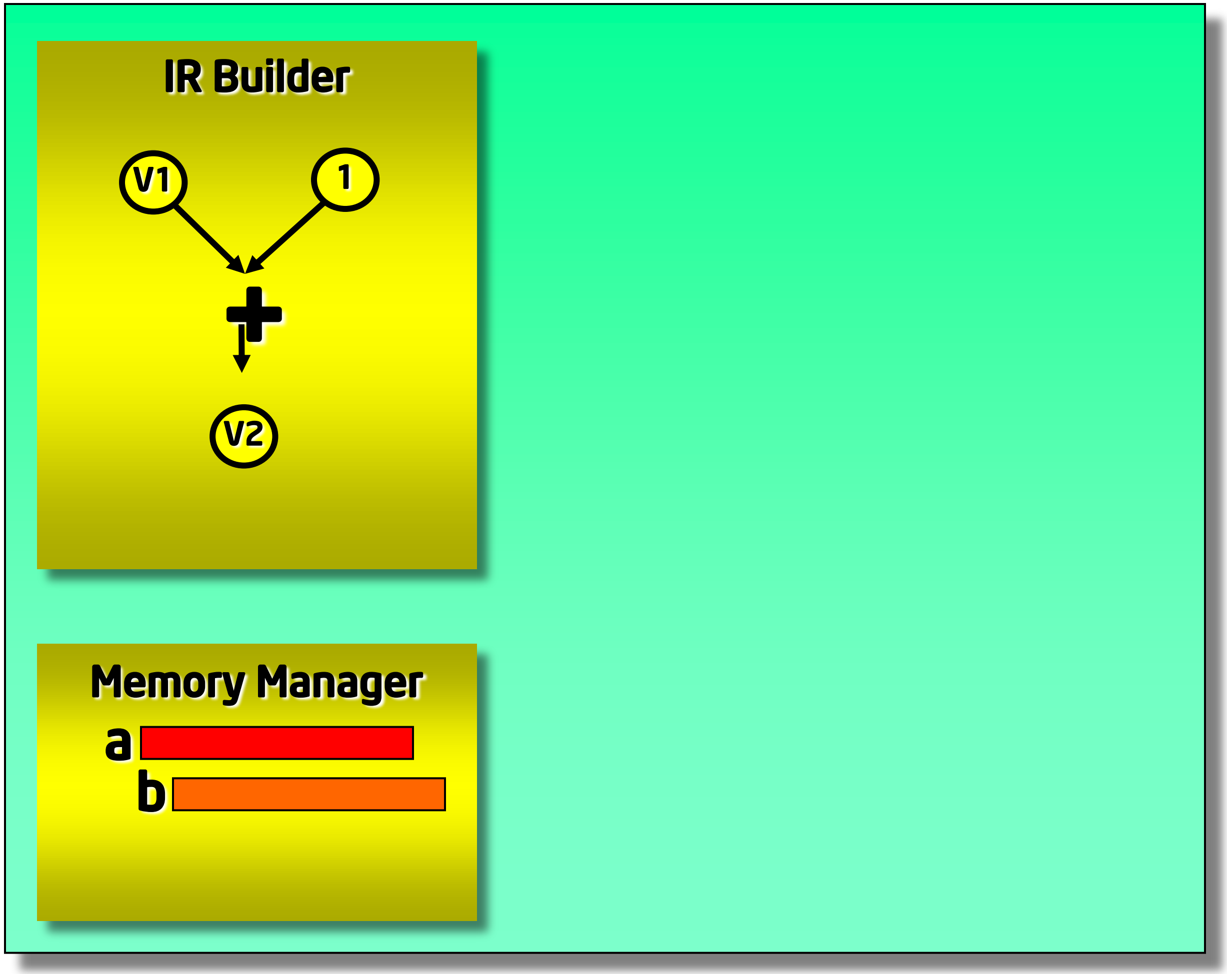
ArBB
Dynamic
Engine

Intel® ArBB Dynamic Engine Execution

```
int ar_a[1024],
    ar_b[1024]
dense<i32> a();
bind(a, ar_a, 1024);
dense<i32> b();
bind(b, ar_b, 1024);
call(work)(a, b);
```

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

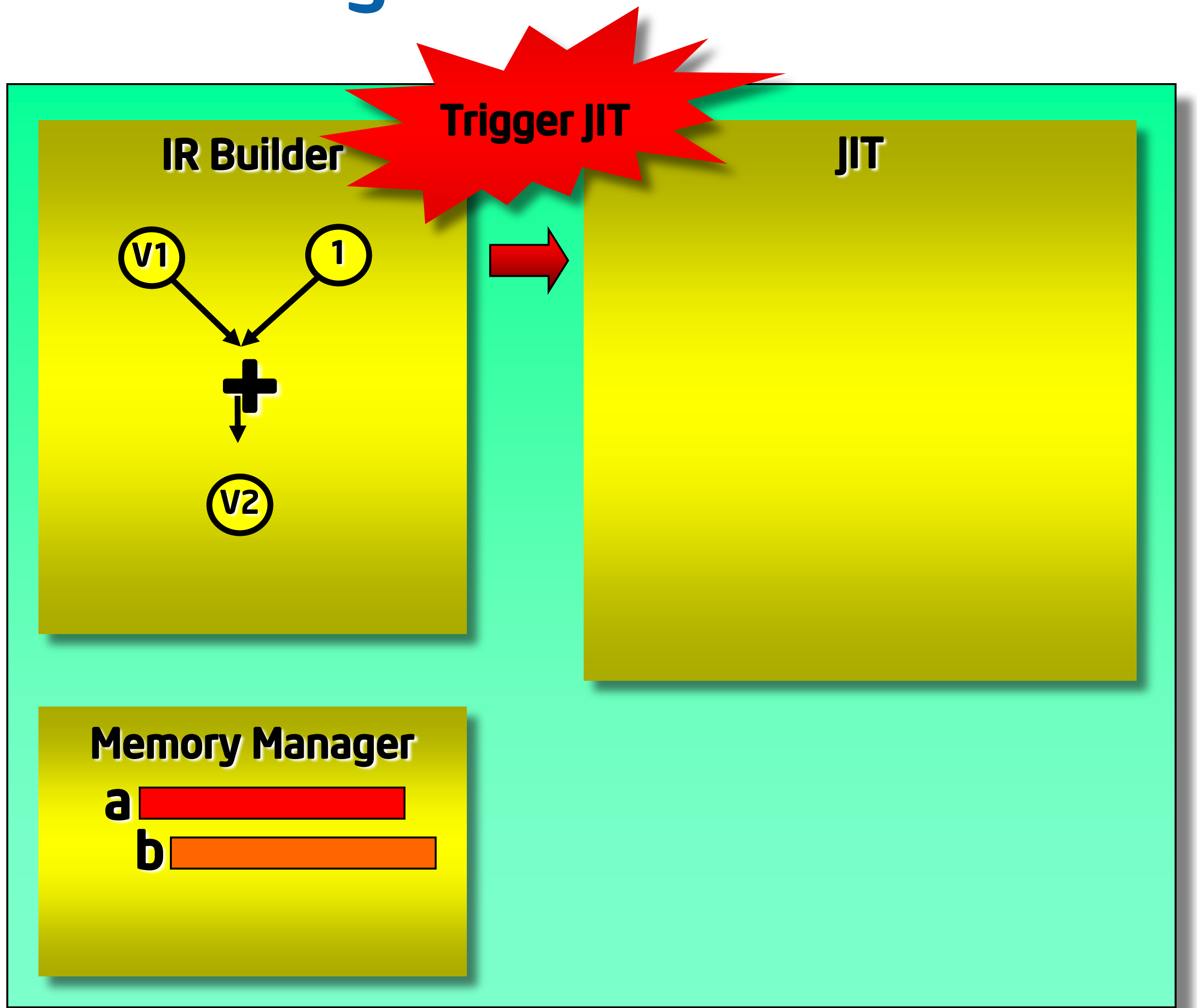
ArBB
Dynamic
Engine



Intel® ArBB Dynamic Engine Execution

```
int ar_a[1024],
    ar_b[1024]
dense<i32> a();
bind(a, ar_a, 1024);
dense<i32> b();
bind(b, ar_b, 1024);
call(work)(a, b);
```

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



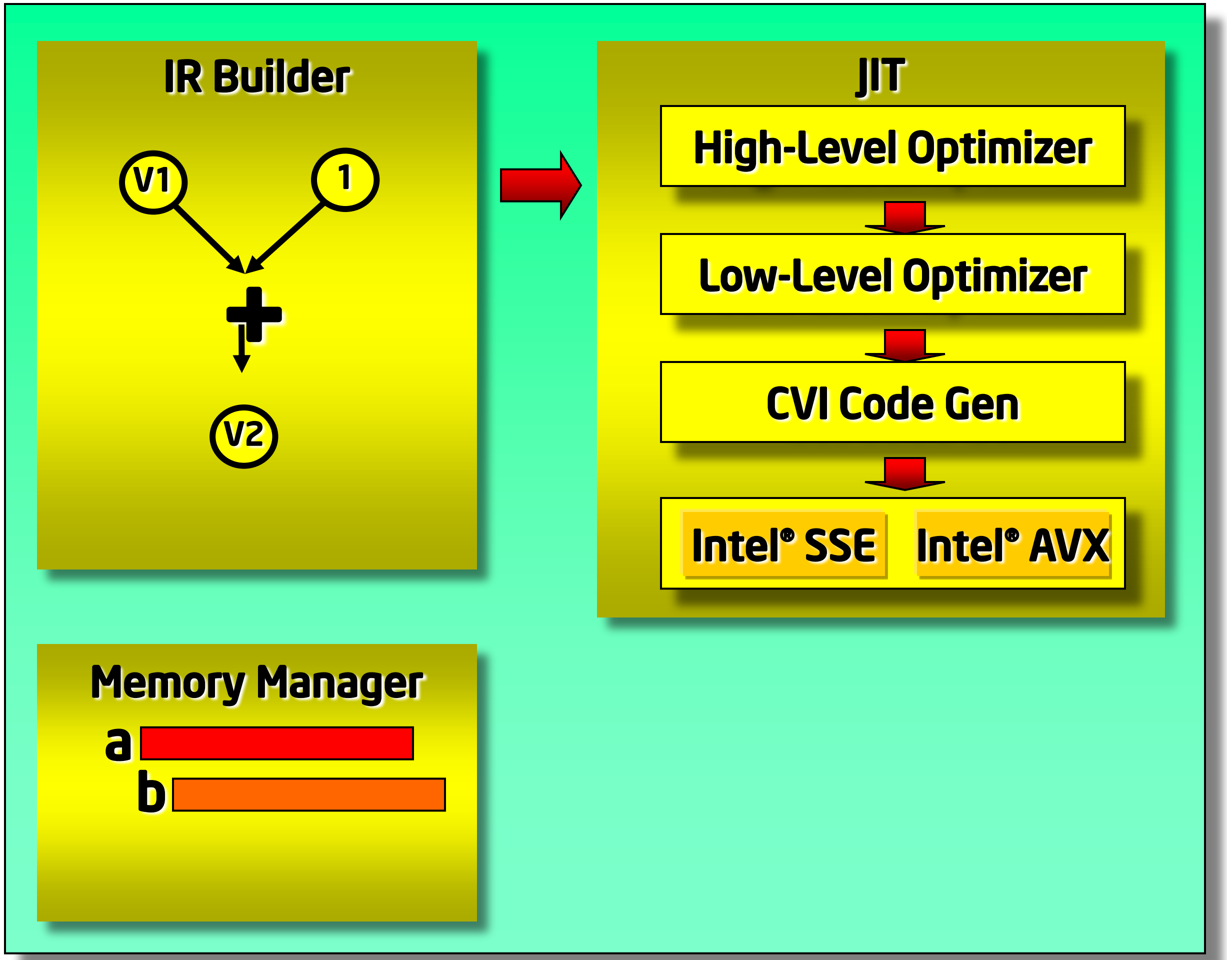
ArBB Dynamic Engine

Intel® ArBB Dynamic Engine Execution

```
int ar_a[1024],
    ar_b[1024]
dense<i32> a();
bind(a, ar_a, 1024);
dense<i32> b();
bind(b, ar_b, 1024);
call(work)(a, b);
```

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

ArBB Dynamic Engine

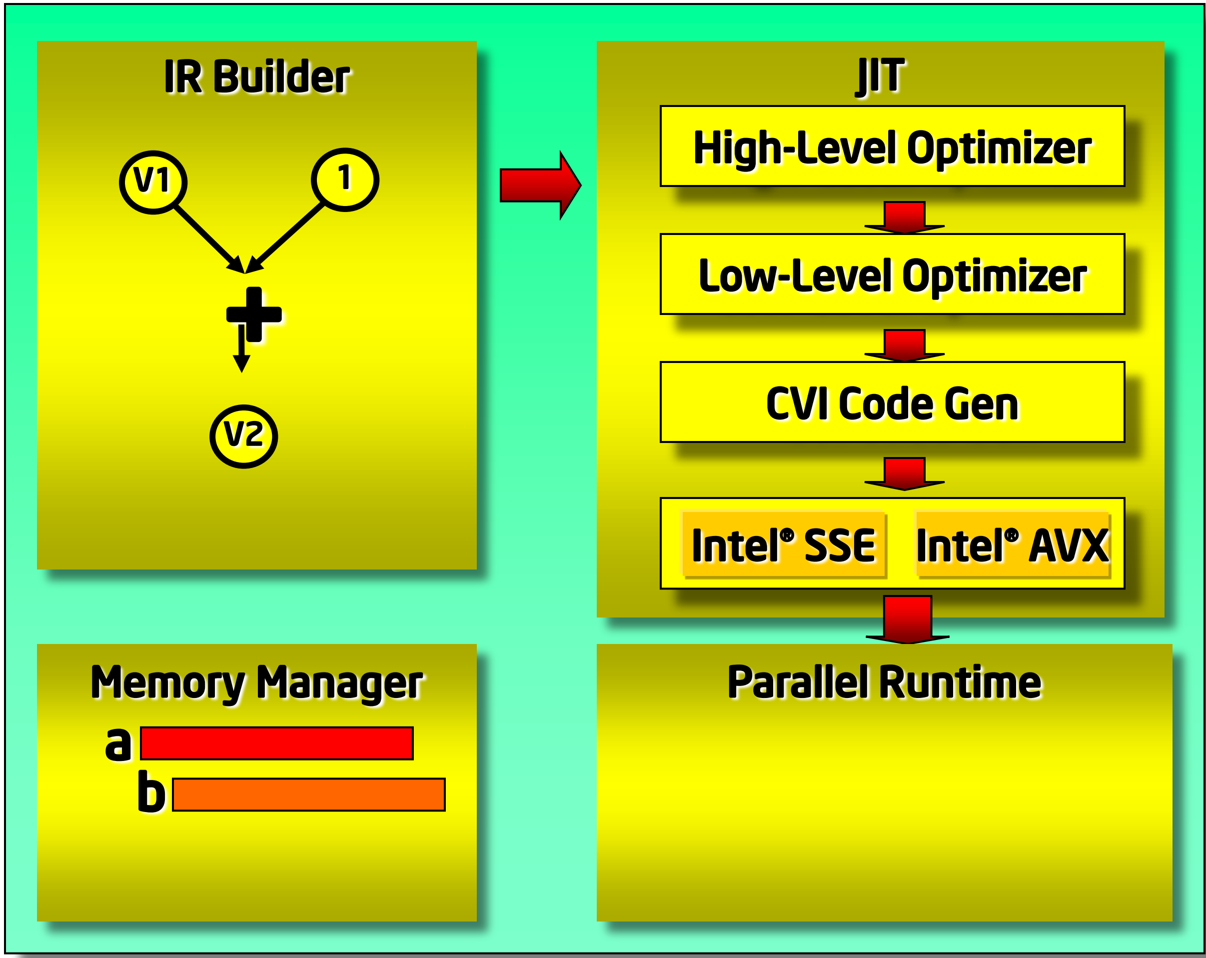


Intel® ArBB Dynamic Engine Execution

```
int ar_a[1024],
    ar_b[1024]
dense<i32> a();
bind(a, ar_a, 1024);
dense<i32> b();
bind(b, ar_b, 1024);
call(work)(a, b);
```

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

ArBB
Dynamic
Engine

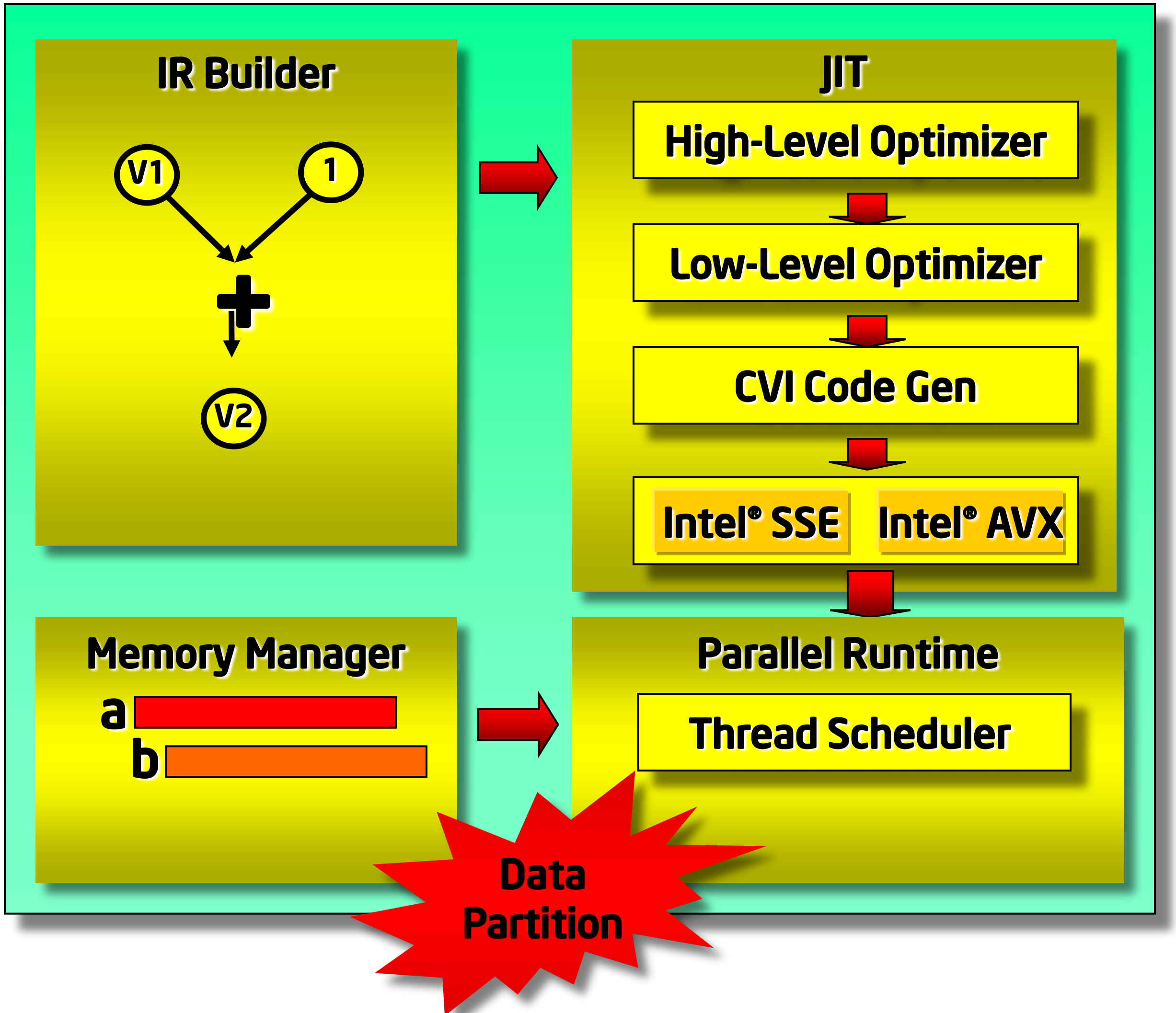


Intel® ArBB Dynamic Engine Execution

```
int ar_a[1024],
    ar_b[1024]
dense<i32> a();
bind(a, ar_a, 1024);
dense<i32> b();
bind(b, ar_b, 1024);
call(work)(a, b);
```

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

ArBB
Dynamic
Engine

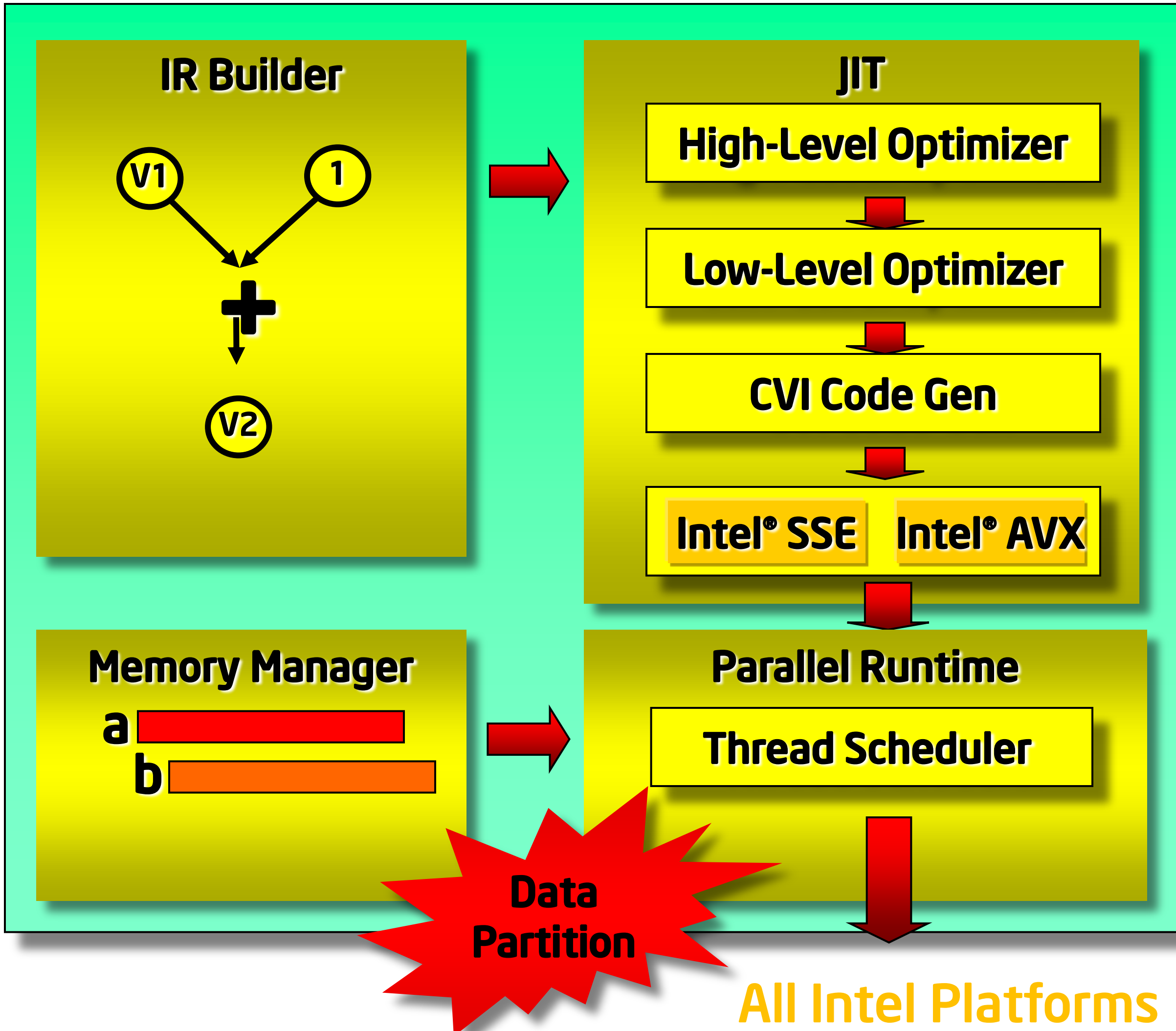


Intel® ArBB Dynamic Engine Execution

```
int ar_a[1024],
    ar_b[1024]
dense<i32> a();
bind(a, ar_a, 1024);
dense<i32> b();
bind(b, ar_b, 1024);
call(work)(a, b);
```

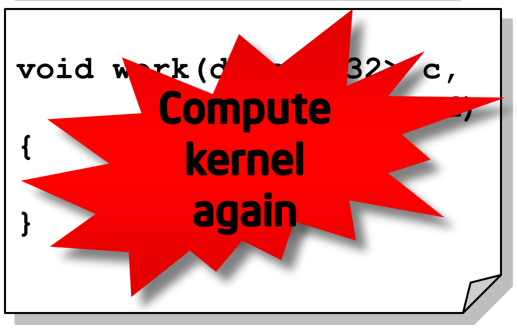
```
void work(dense<i32> c,
          dense<i32>& d)
{
    c = d + 1;
}
```

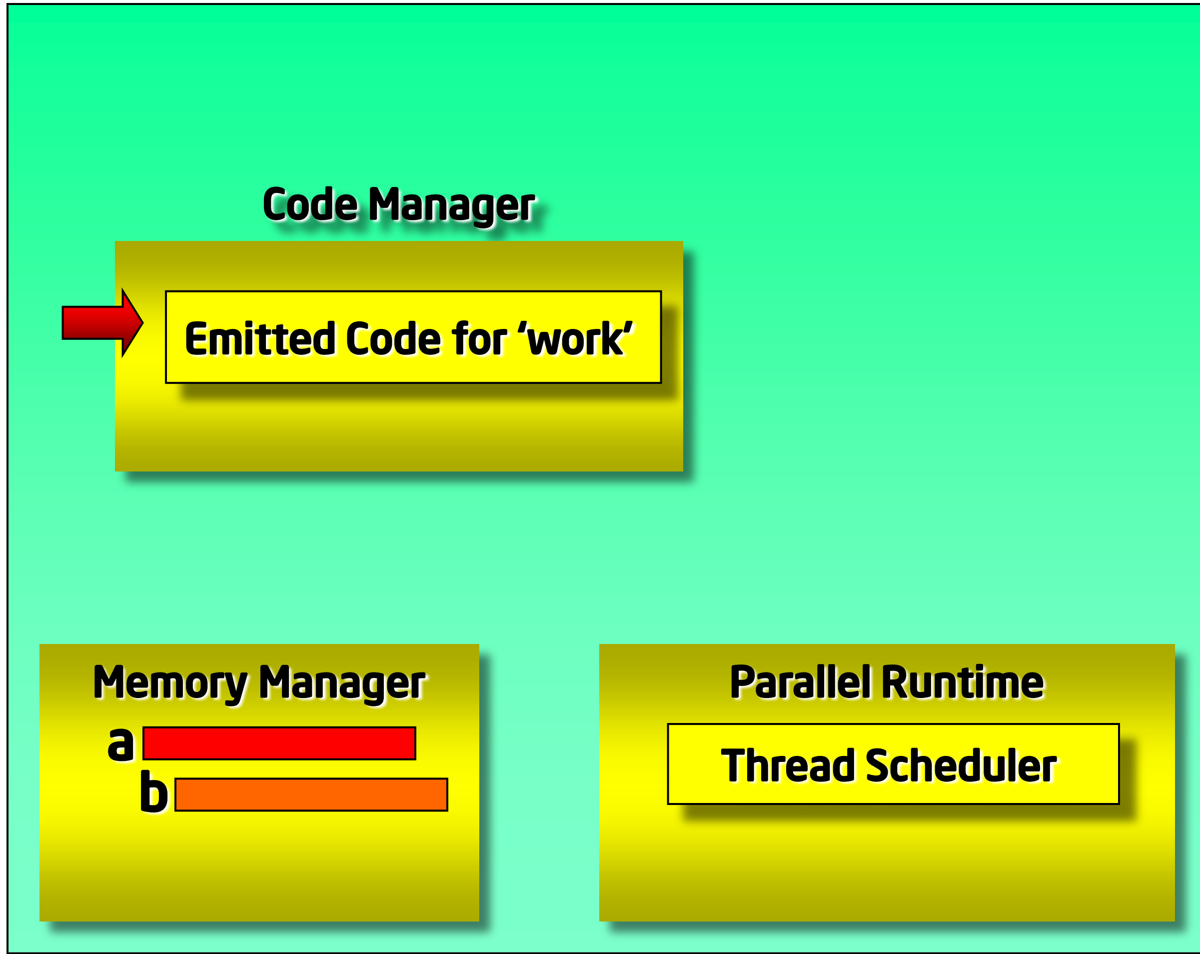
ArBB
Dynamic
Engine



Intel® ArBB Dynamic Engine Execution

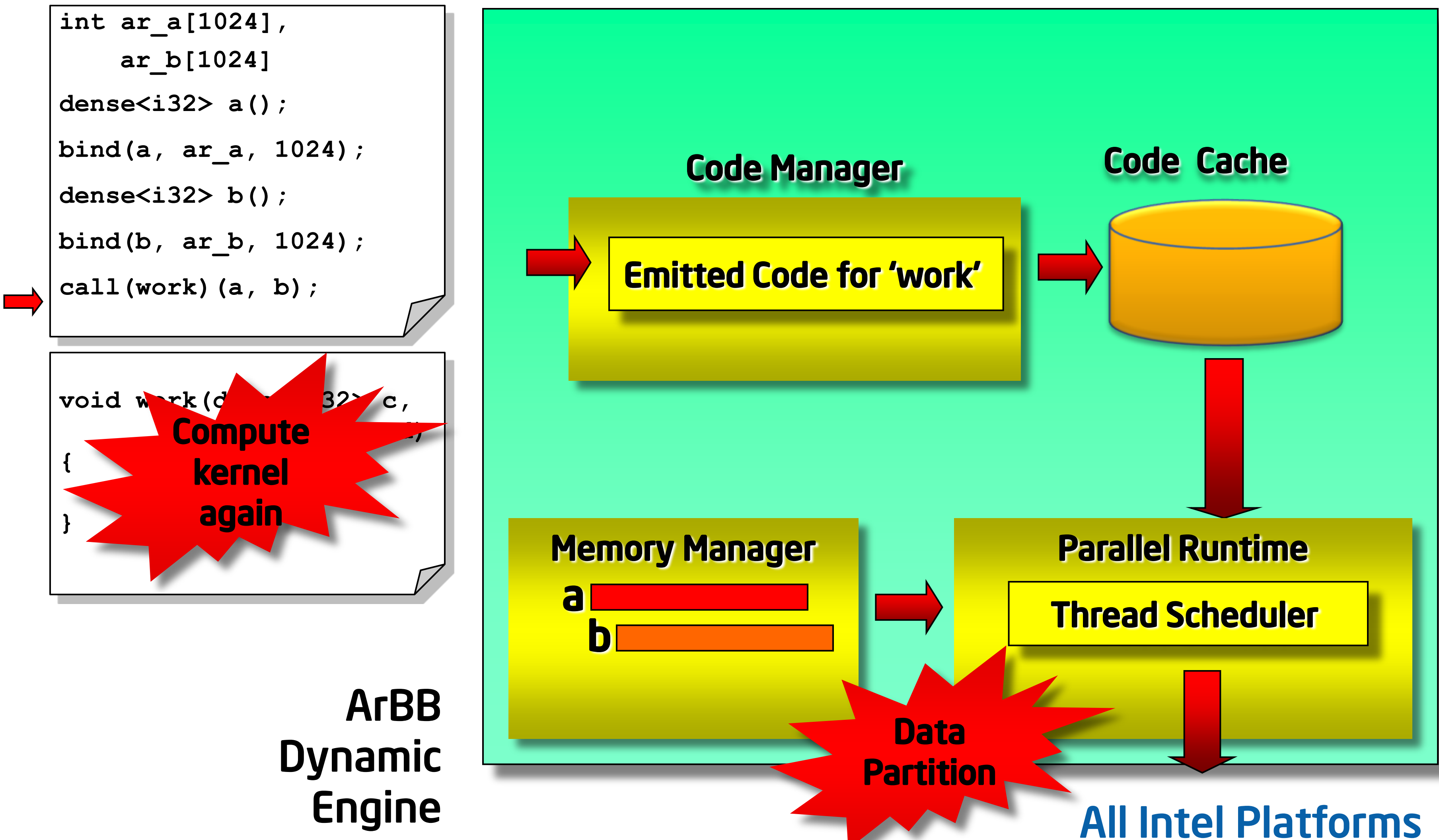
```
int ar_a[1024],
    ar_b[1024]
dense<i32> a();
bind(a, ar_a, 1024);
dense<i32> b();
bind(b, ar_b, 1024);
call(work)(a, b);
```





ArBB
Dynamic
Engine

Intel® ArBB Dynamic Engine Execution



Modifying Runtime Behavior

- **Control behavior through O/S-level environment variables:**

Variable	Values	Description
ARBB_OPT_LEVEL	O0, O2, O3	Set the level of optimization O0 immediate mode O2 enable vectorization O3 enable vectorization and multi-threading
ARBB_VERBOSE	1 (y) or 0 (n)	Instruct JIT compiler and runtime system to emit diagnostic messages during execution.
ARBB_NUM_CORES	Positive integer number	Set the number of threads for multi-threaded execution.
ARBB_DUMPJIT	1 (y) or 0 (n)	Instruct runtime system to emit generated code into working directory.

Intel® Array Building Blocks Execution Engine

Controlling Dynamic Compilation

Capturing

- A `call()` expression works like this:
  - If it's never seen the function passed in before, it *captures* the function into a *closure*, then executes it
  - Otherwise, it executes the previously captured closure

function pointer

```
call(my_function)(arg1, arg2);
```

captures or returns previously capture closure

executes closure returned from call()

Closures

- Concept originating in functional programming languages
  – But extended to include references to mutable objects in imperative languages
- Closures are made of
  – A piece of code to be executed
  – Captured state of bound variables (C++ non-locals)
  – References to mutable non-local variables (ArBB non-locals)
- Once constructed, closures in ArBB are *immutable*
- Similar concepts used in other programming models
  – Threading Building Blocks: thunk classes
  – C++0x: lambda functions

How Capturing Works

- **Capturing simply *executes* the C/C++ function**
  - Any operations on ArBB types will be captured
  - Any non-ArBB C++ operations just execute immediately
- `capture()` **explicitly captures a given function**
  - `call()` only captures a function the first time it sees it
  - `capture()` captures (re-executes) a function every time
- **Using `capture()` is usually not necessary**
  - But understanding the process is helpful
- **Best illustrated with an example...**

Closure Interaction with call

- **template:** `closure<ret_t(arg1_t, arg2_t, …9)>`
  - Represents a function that has been *captured* and can then be called
  - Has a related non-template type, *auto_closure*, that performs run-time type checking instead of static type checking
- **call() returns a closure<…>**
  - **call()** of same function pointer calls capture the first time
  - after that always returns same closure
  - Provides predictable behavior but simple usage for new users

```
void my_function(f32& out, f32 in);
f32 f1, f2;

auto_closure c1 = call(my_function);
auto_closure c2 = call(my_function);
assert(c1 == c2);
call(my_function)(f1, f2);              // works as expected.
c1(f1, f2);                             // equivalent to previous line
```

Capturing: an Example

```
void my_function(f32& result, f32 input) {
   std::cout << “Hello, world!” << std::endl;
   result = input + 1.0f;
}

int main() {
   typedef closure<void (f32&, f32)> mfc;
   mfc a = capture(my_function);
   mfc b = call(my_function);
   mfc c = call(my_function);
}
```

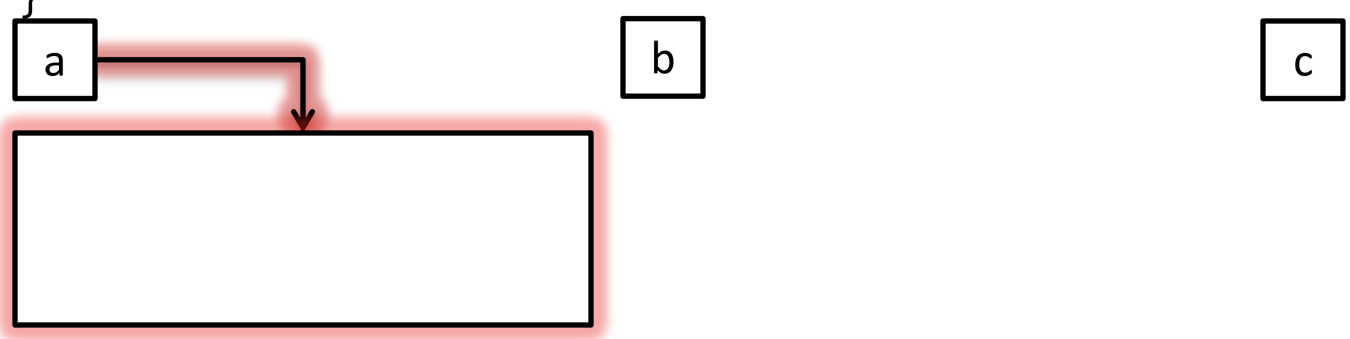
a

b

c

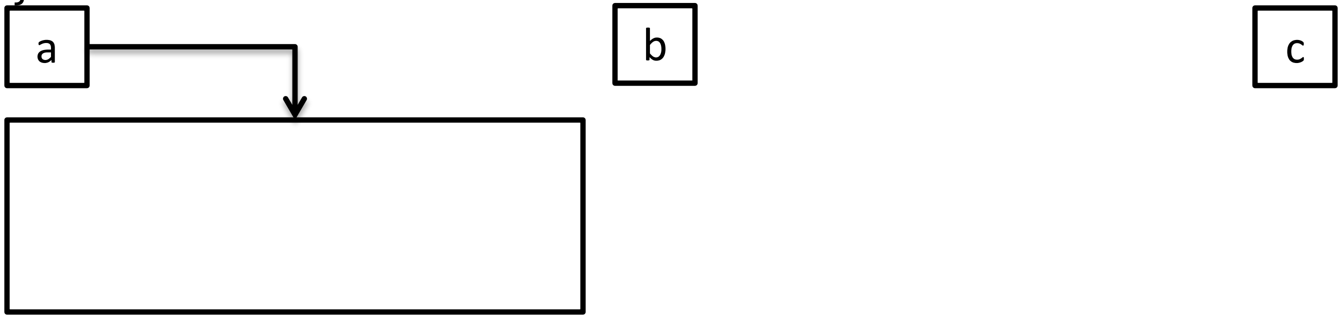
Capturing: an Example

```
void my_function(f32& result, f32 input) {
   std::cout << “Hello, world!” << std::endl;
   result = input + 1.0f;
}

int main() {
   typedef closure<void (f32&, f32)> mfc;
   mfc a = capture(my_function);
   mfc b = call(my_function);
   mfc c = call(my_function);
}
```



Capturing: an Example

```
void my_function(f32& result, f32 input) {
   std::cout << “Hello, world!” << std::endl;
   result = input + 1.0f;
}

int main() {
   typedef closure<void (f32&, f32)> mfc;
   mfc a = capture(my_function);
   mfc b = call(my_function);
   mfc c = call(my_function);
}
```



Capturing: an Example

```
void my_function(f32& result, f32 input) {
   std::cout << "Hello, world!" << std::endl;
   result = input + 1.0f;
}

int main() {
   typedef closure<void (f32&, f32)> mfc;
   mfc a = capture(my_function);
   mfc b = call(my_function);
   mfc c = call(my_function);
}
```

```
Hello, world!
```

a

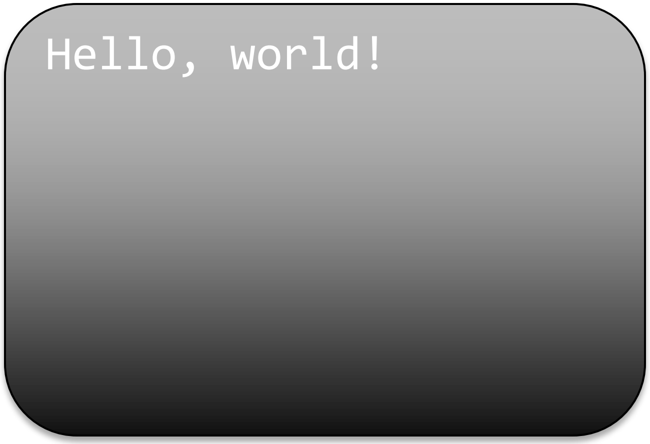
```
tmp = add(input, 1.0);
result = tmp;
```

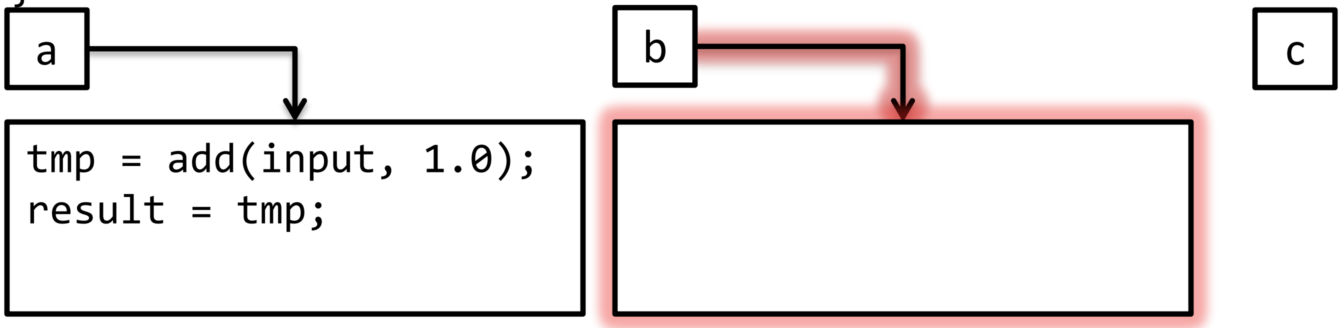
b

c

Capturing: an Example

```
void my_function(f32& result, f32 input) {
   std::cout << “Hello, world!” << std::endl;
   result = input + 1.0f;
}

int main() {
   typedef closure<void (f32&, f32)> mfc;
   mfc a = capture(my_function);
   mfc b = call(my_function);
   mfc c = call(my_function);
}
```





Capturing: an Example

```
void my_function(f32& result, f32 input) {
   std::cout << "Hello, world!" << std::endl;
   result = input + 1.0f;
}

int main() {
   typedef closure<void (f32&, f32)> mfc;
   mfc a = capture(my_function);
   mfc b = call(my_function);
   mfc c = call(my_function);
}
```

```
Hello, world!
Hello, world!
```

a

```
tmp = add(input, 1.0);
result = tmp;
```

b

c

Capturing: an Example

```
void my_function(f32& result, f32 input) {
   std::cout << “Hello, world!” << std::endl;
   result = input + 1.0f;
}

int main() {
   typedef closure<void (f32&, f32)> mfc;
   mfc a = capture(my_function);
   mfc b = call(my_function);
   mfc c = call(my_function);
}
```

```
Hello, world!
Hello, world!
```

a

```
tmp = add(input, 1.0);
result = tmp;
```

b

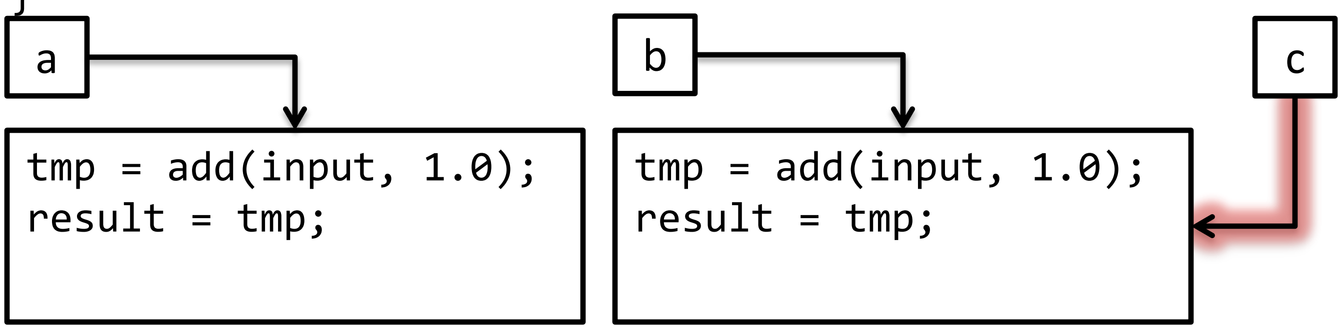
```
tmp = add(input, 1.0);
result = tmp;
```

c

Capturing: an Example

```
void my_function(f32& result, f32 input) {
   std::cout << “Hello, world!” << std::endl;
   result = input + 1.0f;
}

int main() {
   typedef closure<void (f32&, f32)> mfc;
   mfc a = capture(my_function);
   mfc b = call(my_function);
   mfc c = call(my_function);
}
```

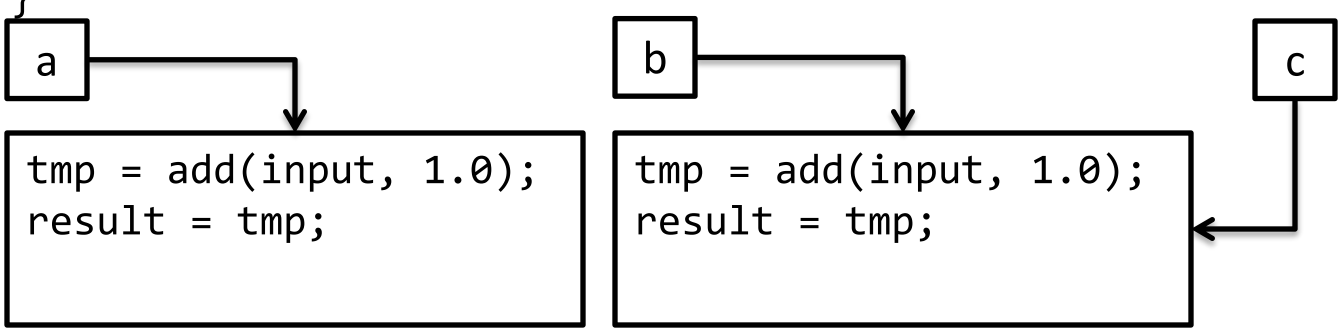
```
Hello, world!
Hello, world!
```



Capturing: an Example

```
void my_function(f32& result, f32 input) {
   std::cout << “Hello, world!” << std::endl;
   result = input + 1.0f;
}

int main() {
   typedef closure<void (f32&, f32)> mfc;
   mfc a = capture(my_function);
   mfc b = call(my_function);
   mfc c = call(my_function);
}
```

```
Hello, world!
Hello, world!
```



C++ Control Flow

- **Regular C++ control flow works *during capture***
  - When a function is captured into a closure, the closure contains the *effect* of the control flow *at the time it was captured*

```
positive = true;
void my_function(f32& result, f32 input) {
    if (positive) {
        result = input + 1.0f;
    } else {
        result = input - 1.0f;
    }
}

int main() {
    closure<void (f32&, f32)> closure_pos = capture(my_function);
    positive = false;
    closure<void (f32&, f32)> closure_neg = capture(my_function);
}
```

Intel® ArBB Control Flow

- **ArBB provides its own control flow constructs**
  - They can be used on ArBB types (e.g. `boolean`)
  - The *control flow* will be captured, not just its effects

```
boolean positive = true;
void my_function(f32& result, f32 input) {
    _if (positive) {
        result = input + 1.0f;
    } _else {
        result = input - 1.0f;
    } _end_if;
}

int main() {
    closure<void (f32&, f32)> closure = capture(my_function);
    positive = false;
    // No need to re-capture
}
```

Intel® Array Building Blocks Execution Engine

Function Calls

Function Calls in Intel® ArBB

C++ Space

- **No call operator**
  - Standard C/C++ function call
  - Compiler decides on inlining, or can use `inline` keyword
- **call operator**
  - `call()` to invoke an C++ function w/ ArBB code
  - Triggers on-demand code compilation

Intel® ArBB Space

- **No operator**
  - Standard C/C++ function call
  - Full inlining of function body
- **call operator**
  - `call()` to invoke an C++ function w/ ArBB code
  - True function call involving a branch
- **map operator**
  - Replicates function over index space of array

Map Operator: How it Works…

- Conceptually, the map function runs in 3 steps:
  1. For each parameter, the argument is *copied in*.
  2. The map function executes *in parallel*

     with one instance for every element in containers passed in.
  3. For each reference parameter, the result is *copied out*.
- This means:
  – Map instances are *completely independent*
  – The effects of one instance is not visible until after all have executed
- Execution follows the “as-if rule”:
  – Normally, copies are avoided completely
    – as long as the semantics are the same
  – Instances of a map execution will be blocked into larger tasks
    – rather than spawning a task for every instance

Map Operator: `map()`

- Syntax of the `map()` operator:

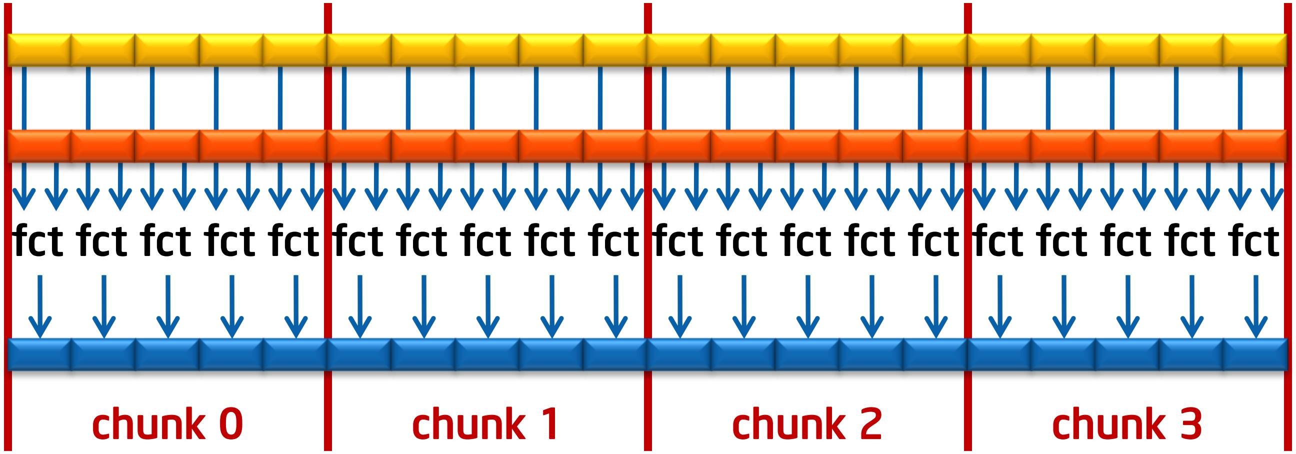
$$\underbrace{\texttt{map(}\overbrace{\texttt{my\_function}}^{\text{function pointer}}\texttt{)}}_{\text{captures or returns previously capture closure}}\underbrace{\texttt{(arg1, arg2);}}_{\text{executes closure returned from map()}}$$

- The map function must be compatible with the dense containers given as arguments
  - A map function takes and returns ArBB scalars and structured types
  - At the map() operator, dense containers must be of the same scalar types or a structured type

Map Operator: `map()`

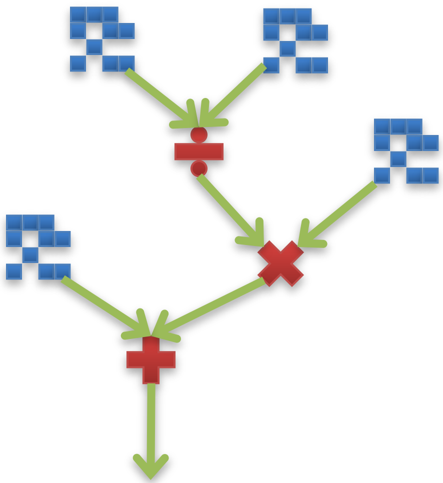
- **Example:**

```
void fct(f32 a, f32 b, f32& c) {
    c = a + b;
}

void func(dense<f32> a, dense<f32> b, dense<f32>& c) {
    map(fct)(a, b, c);
}
```



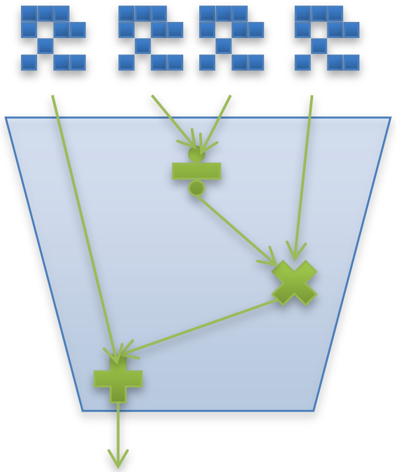
Vector Processing vs. Scalar Processing

Vector Processing



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

Scalar Processing



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

Map Functions: Argument Types

- **Map functions can take two kinds of arguments**
  - Map functions are polymorphic and can be applied to any combination

- **Fixed arguments:**
  - Are values whose type exactly matches those of the parameters to which they are being passed
  - Values are replicated over every instance of the map
- **Varying arguments:**
  - Are containers being passed to parameters corresponding to their element type
  - Map is applied to every element individually

Map Functions: Argument Types

```
void mad(i64 base, i64 offset, i64 scale, i64& result) {
    result = base + offset * scale;
}

void apply_mad() {
    i64 base = 0xDEADBEEF;
    dense<i64> offsets;
    i64 scale = 8;
    dense<i64> result;

    map(mad)(base, offsets, scale, result);
}
```

Color code:

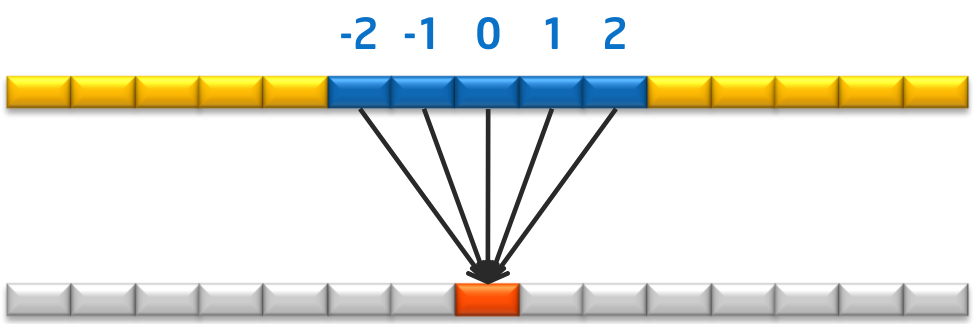
Fixed argument

Varying argument

Stencils in Map Functions

- Use `neighbor()` for stencil codes:

```
void fun3x3<f32 a,
            f32 w0, f32 w1, f32 w2, f32 w3, f32 w4,
            f32& r) {
    r = w0 * a +
        w1 * neighbor(a, -2) + w2 * neighbor(a, -1) +
        w3 * neighbor(a, 1) +  w4 * neighbor(a, 2);
};
```



Intel® Array Building Blocks Execution Engine

Using the Virtual Machine Interface

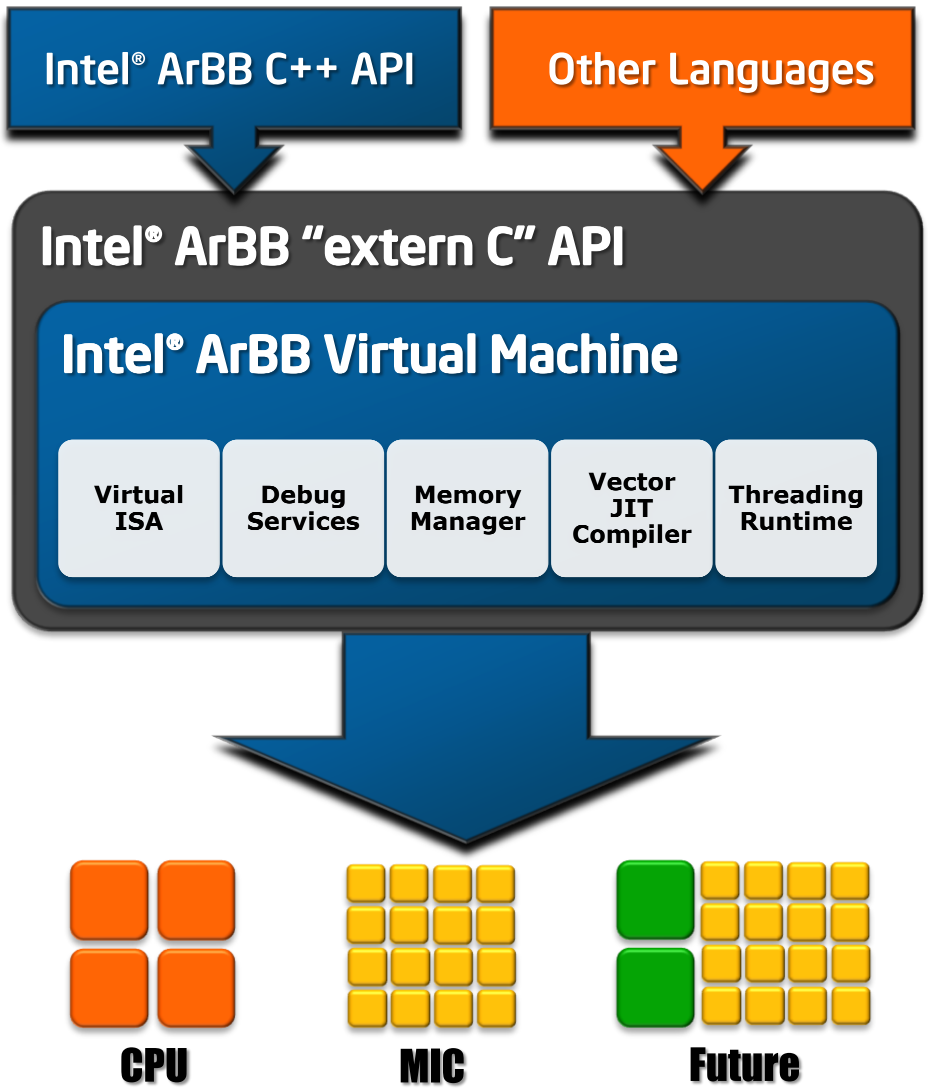
Intel® ArBB VM

Intel® ArBB has a high-level, standards compliant C++ interface to a Virtual Machine (VM)

Can be used with a broad range of ISO standard C++ compilers

VM both manages threads and dynamically generates optimized vector code

Code is *portable* across different SIMD widths and different core counts, *even in binary form*



Virtual Machine Functionality

Data management

- Declare new types
- Allocate containers
- Move/bind data between the application and the VM

Function definition

- Generate functions at runtime
- Using sequences of both scalar and collective operations

Execution management

- Execute functions
- Including remote and parallel execution

Virtual Machine Interface

- **The VM interface is a C-only interface**
  - C calling convention wide-spread
  - Almost all modern programming languages can call into C functions
  - Very generic applicability of the VM interface
- **Requirements**
  - No assumptions about host language
  - No assumptions about memory management
  - Easily bind as DLL or dynamic library
  - Fixed ABI, i.e., simple to update without recompilation
- **API**
  - All internal types are opaque, no data is exposed publicly
  - Consistent error handling through opaque error structure
    - No use of exceptions
    - Suitable for any language frontend
  - Stateless, using context objects for thread-safety

Example: Dot Product

```
void arbb_sprod(dense<f32> a,
                dense<f32> b,
                f32& result) {
  result = add_reduce(a * b);
}
```

Structure of the code

- Get a handle to the ArBB VM
- Create the input/output types needed (scalars, dense container)
- Create a function called “dot”
  - Define the input and output parameters of “dot”
  - Create local variables

Structure of the code

- Create a function called “dot”
  - Do a element-wise multiplication between the input containers
  - Do an add_reduce on the result of the multiplication
  - … store the result in the output argument

- The next slides will show how this can be done with the VM API

```
arbb_function_t generate_dot() {
    arbb_context_t context;
    arbb_get_default_context(&context, NULL);
    arbb_type_t base_type;
    arbb_get_scalar_type(context, &base_type, arbb_f32, NULL);
    arbb_type_t dense_1d_f32;
    arbb_get_dense_type(context, &dense_1d_f32, base_type, 1, NULL);
    arbb_type_t inputs[]  = { dense_1d_f32, dense_1d_f32 };
    arbb_type_t outputs[] = { base_type };
    arbb_type_t fn_type;
    arbb_get_function_type(context, &fn_type, 1, outputs, 2, inputs, NULL);

    arbb_function_t function;
    arbb_begin_function(context, &function, fn_type, "dot", 0, NULL);
      arbb_variable_t a, b, c;
      enum { is_input, is_output };
      arbb_get_parameter(function, &a, is_input,  0, NULL);
      arbb_get_parameter(function, &b, is_input,  1, NULL);
      arbb_get_parameter(function, &c, is_output, 0, NULL);

      arbb_variable_t tmp[1];
      arbb_create_local(function, tmp, dense_1d_f32, 0, NULL);

      arbb_variable_t in[] = { a, b };
      arbb_op(function, arbb_op_mul, tmp, in, 0, NULL);

      arbb_variable_t result[] = { c };
      arbb_op_dynamic(function, arbb_op_add_reduce, 1, result, 1, tmp, 0, NULL);
    arbb_end_function(function, NULL);
    arbb_compile(function, NULL);
    return function;
}
```

Intel® ArBB VM Code for Dot Product

```
arbb_function_t generate_dot() {
    arbb_context_t context;
    arbb_get_default_context(&context, NULL);

    arbb_type_t base_type;
    arbb_get_scalar_type(context, &base_type, arbb_f32, NULL);

    arbb_type_t dense_1d_f32;
    arbb_get_dense_type(context, &dense_1d_f32, base_type, 1, NULL);

    arbb_type_t inputs[]  = { dense_1d_f32, dense_1d_f32 };
    arbb_type_t outputs[] = { base_type };

    arbb_type_t fn_type;
    arbb_get_function_type(context, &fn_type, 1, outputs, 2, inputs, NULL);

    // continue on the next slide
}
```

Intel® ArBB VM Code for Dot Product

```
arbb_function_t generate_dot() {
    // continued from previous slide

    arbb_function_t function;
    arbb_begin_function(context, &function, fn_type, "dot", 0, NULL);

        arbb_variable_t a, b, c;
        enum { is_input, is_output };
        arbb_get_parameter(function, &a, is_input,  0, NULL);
        arbb_get_parameter(function, &b, is_input,  1, NULL);
        arbb_get_parameter(function, &c, is_output, 0, NULL);

        arbb_variable_t tmp[1];
        arbb_create_local(function, tmp, dense_1d_f32, 0, NULL);

    // continue on the next slide
}
```

Intel® ArBB VM Code for Dot Product

```
arbb_function_t generate_dot() {
    // continued from previous slide

      arbb_variable_t in[] = { a, b };
      arbb_op(function, arbb_op_mul, tmp, in, 0, NULL);

      arbb_variable_t result[] = { c };
      arbb_op_dynamic(function, arbb_op_add_reduce,
                      1, result, 1, tmp, 0, NULL);

    arbb_end_function(function, NULL);

    arbb_compile(function, NULL);

    return function;
}
```

Using the Generated “dot” Function

As an example, serialize the generated code

```
void print_generated_code() {
    arbb_function_t function = generate_dot();
    arbb_string_t serialized;
    arbb_serialize_function(function, &serialized, NULL);
    const char *cstring = arbb_get_c_string(serialized);
    printf("%s", cstring);
    arbb_free_string(serialized);
}
```

Generated code

```
function _dot(out $f32 _0, in dense<$f32> _1,
              in dense<$f32> _2) {
  _3 = mul<dense<$f32>>(_1, _2);
  _0 = add_reduce<$f32>(_3);
}
```

Advanced Intel® ArBB Programming

Topics

- Templates
- Generative metaprogramming
- User-defined data types
- User-defined functions
- Debugging
- Optimizing for performance

Advanced Intel® ArBB Programming

Templates

Basic Templates

- C++ functions with ArBB code may be subject to template type arguments
- Programmers can write ArBB code as templated code:

```
template <typename T>
void foo(...) {
   /* ArBB code */
}

call(foo<f32>)(...);
call(foo<f64>)(...);
call(foo<array<i32,4>>)(...);
```

Advanced Template Usages

- Standard C++ templates may often be re-used as ArBB templates
  - if (and only if) template instantiation yields a valid ArBB code sequence
- Example:

```
template<typename T>
void double_it(const T& a, T& result) {
  result = a * 2;
}
```

Advanced Template Usages

```
template<typename T>
void double_it(const T& a, T& result) {
  result = a * 2;
}
```

use as a standard C++ function (non-ArBB):

```
void caller() {
  double a = 2.0;
  double r;
  double_it(a, r);          // T := double
}
```

Advanced Template Usages

```
template<typename T>
void double_it(const T& a, T& result) {
  result = a * 2;
}
```

use as function on dense containers:

```
void caller() {
  dense<f64> a = fill(1.0, 1024);
  dense<f64> r;
  double_it(a, r);          // T := dense<f64>
}
```

Advanced Template Usages

```
template<typename T>
void double_it(const T& a, T& result) {
  result = a * 2;
}
```

use as kernel (mapped function) on containers:

```
void caller() {
  dense<f64> a = fill(1.0, 1024);
  dense<f64> r;
  map(double_it)(a, r);  // T := f64
}
```

Advanced Intel® ArBB Programming

Generative Metaprogramming

Function Calls

- No call operator
  - Standard C/C++ function call
  - Full inlining of function body
- call operator
  - call() to invoke an C++ function w/ ArBB code
  - True function call involving a branch

```
void func() {
    statement_1();
    statement_2();
}
void example() {
    call(func)();
    func();
    func();
    call(func)();
}
```

→

```
void example() {
    call(func)();
    statement_1();
    statement_2();
    statement_1();
    statement_2();
    call(func)();
}
```

Control Flow vs Generative Programming

_if statement

```
boolean flag = true;
void my_function(f32& result,
                 f32 input) {
   _if (flag) {
      result = sin(input);
   } _else {
      result = cos(input);
   } _end_if;
}
```

capture(my_function)

```
void my_function(f32& result,
                 f32 input) {
   _if (flag) {
      result = sin(input);
   } _else {
      result = cos(input);
   } _end_if;
}
```

if statement

```
bool flag = true;
void my_function(f32& result,
                 f32 input) {
   if (flag) {
      result = sin(input);
   } else {
      result = cos(input);
   }
}
```

capture(my_function)

```
void my_function(f32& result,
                 f32 input) {
   result = sin(input);
}
```

Closures and Capturing Example

```
unsigned int unroll_factor;
void foo(dense<f32>& out, dense<f32> in)
{
  _for(index i = 0, i < in.size()/unroll_factor, ++i) {
    for (unsigned int j = 0; j < unroll_factor; ++j) {
      // ...perform some unrolled operation…
    }
  } _end_for;
}

int main() {
  unroll_factor = 1;
  closure<void(dense<f32>&, dense<f32>)> not_unrolled = capture(foo);
  unroll_factor = 4;
  closure<void(dense<f32>&, dense<f32>)> unrolled_4_times = capture(foo);

  dense<f32> input(…), result(…);
  not_unrolled(result, input);          // like call(foo)(result, input),
  unrolled_4_times(result, input);      // but with specialization
}
```

Closures and Capturing Example

```
unsigned int unroll_factor;
void foo(dense<f32>& out, dense<f32> in)
{
  _for(i32 i = 0, i < in.size()/unroll_factor, ++i) {
    for (unsigned int j = 0; j < unroll_factor; ++j) {
      // ...perform some unrolled operation…
    }
  } _end_for;
}
```

`unroll_factor = 1`

`unroll_factor = 4`

Closure “not_unrolled”

```
void foo(dense<f32>& out, dense<f32> in)
{
  _for(i32 i = 0, i < in.size(), ++i) {
      // ...perform some unrolled operation…
  }
}
```

Closure “unrolled_4”

```
void foo(dense<f32>& out, dense<f32> in)
{
  _for(i32 i = 0, i < in.size()/4, ++i) {
    // ...perform some unrolled operation…
    // ...perform some unrolled operation…
    // ...perform some unrolled operation…
    // ...perform some unrolled operation…
  }
}
```

Advanced Intel® ArBB Programming

User-defined Data Types

User-defined Types

- C++ classes and structures can be used (mostly) normally in ArBB, including:
  - class members
  - member functions
  - overloaded operators
  - …
- Requirements
  - primitive types be classes in ArBB types (f32, etc.) or other ArBB structured types
  - Default constructible
  - Copy construction not suppressed
  - Operator implementation according to actual operator usage
  - Virtual functions resolved at capture time

User-defined Data Types: Example

```
template<typename T>
struct interval {
  typedef T value_type;
  interval(): m_data(make_array<2, T>(0)) {}
  interval(const T& a, const T& b) {
    m_data[0] = a; m_data[1] = b;
  }
  interval& operator+=(const interval& rhs) {
    m_data += rhs.m_data;
    return *this;
  }
private:
  array<T, 2> m_data;
};
```

Efficient Handling of Structured Types

- Structured types are automatically converted to allow for vectorization
  - Break up AoS (array of structures)
  - Transform to SoA (structure of arrays)
- Involves copy operations for data layout conversion
  - Needed for efficient vectorization
  - Negligible for large problem sizes and more efficient vectorization compensates copy overhead

```
struct interval {
private:
  array<T, 2> m_data;
};
```

→

```
struct temp {
private:
  dense<T> m_data_elt0;
  dense<T> m_data_elt1;
};
```

Advanced Intel® ArBB Programming

User-defined Functions

User-defined Functions: Example

```
template<typename T>
struct interval {
  typedef T value_type;
  interval(): m_data(make_array<2, T>(0)) {}
  interval(const T& a, const T& b) {
    m_data[0] = a; m_data[1] = b;
  }
  T width() const { return m_data[1] - m_data[0]; }
private:
 array<T, 2> m_data;
};
```

How to invoke member function width for a dense container of interval?

Declaring User-defined Functions

Declarative macros for user-defined functions:

ARBB_ELTWISE_	FUNCTION	1	(	ret_type,		func, arg_types)
	METHOD	…			class,	
	TMETHOD	35			class<T>,	

- Let *N* be the function arity (number of arguments)
- Extend user-defined functions into functions on dense:
  - ARBB_ELTWISE_FUNCTION_*N*

    F(T1, T2, …) → F(dense<T1>, dense<T2>, …)
  - ARBB_ELTWISE_METHOD_*N*

    C::M(T1, T2, …) → M(dense<C>, dense<T1>, dense<T2>, …)
  - ARBB_ELTWISE_TMETHOD_*N*

    C<T>::M(T1, T2, …) → M(dense<C<T>>, dense<T1>, dense<T2>, …)

User-defined Functions: Example

```
template<typename T>
struct interval {
  ...
  T width() const { return m_data[1] - m_data[0]; }
  ...
  ARBB_ELTWISE_TMETHOD_0(T, const interval<T>, width)
};
```

Advanced Intel® ArBB Programming

Debugger Integration

Debugging Intel® ArBB Applications

- Debugging of ArBB code possible through standard debugger, e.g.
  - Visual Studio* debugger
  - GNU Debugger (gdb)
- ArBB supplies a script for debugger integration
  - Introspection of ArBB scalars and dense containers
  - Visualization of values of scalars and data in dense containers
  - Provides insight into ArBB's opaque types in the C++ space
- Debugging mode of ArBB
  - Execution mode ARBB_OPT_LEVEL=O0
  - Debugger integration relies on immediate mode of execution

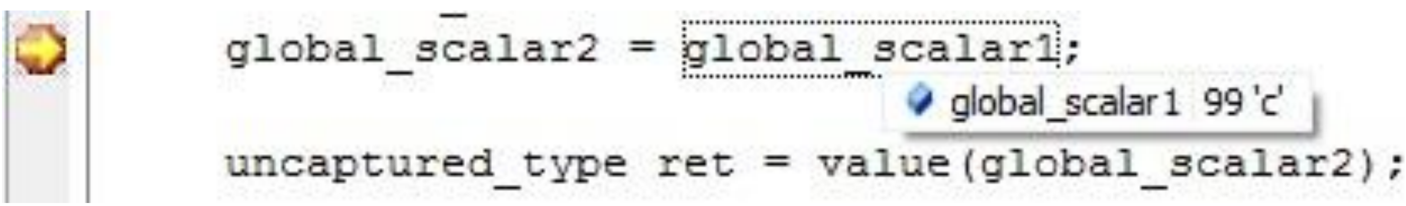
Debugger Integration (Visual Studio*)

- Immediate mode triggers non-JIT execution of ArBB code
  - No IR recording and JIT compilation involved
  - ArBB execution directly happens in C++ space
- Standard debugger features work as expected (e.g. breakpoints)
- Control flow can directly be monitored through Visual Studio debugger commands

- Note: Capturing and closure creation is not (currently) supported

Debugger Integration (Visual Studio*)

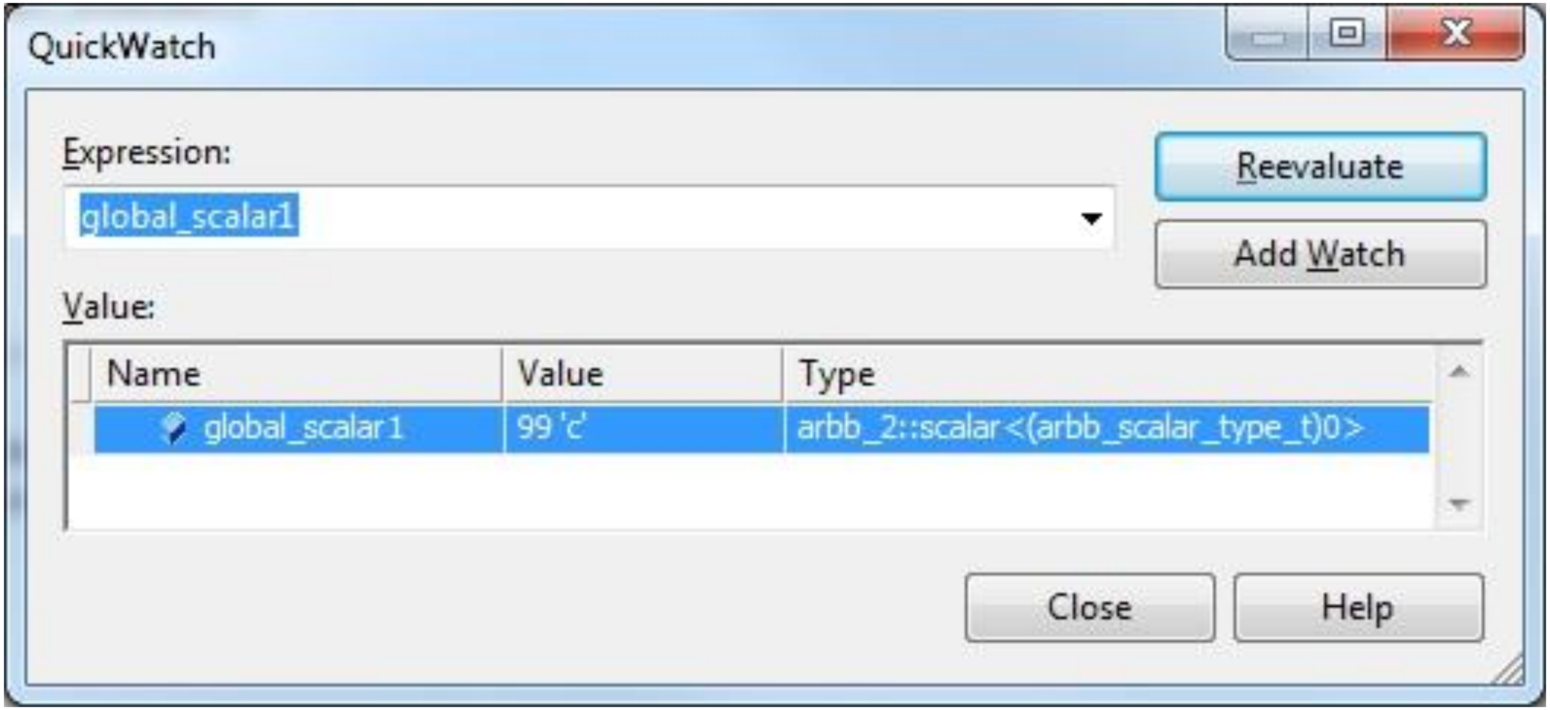
- SmartTags expose current state of ArBB scalars and dense containers:



Screenshots taken from Microsoft* Visual Studio 2008*

Debugger Integration (Visual Studio*)

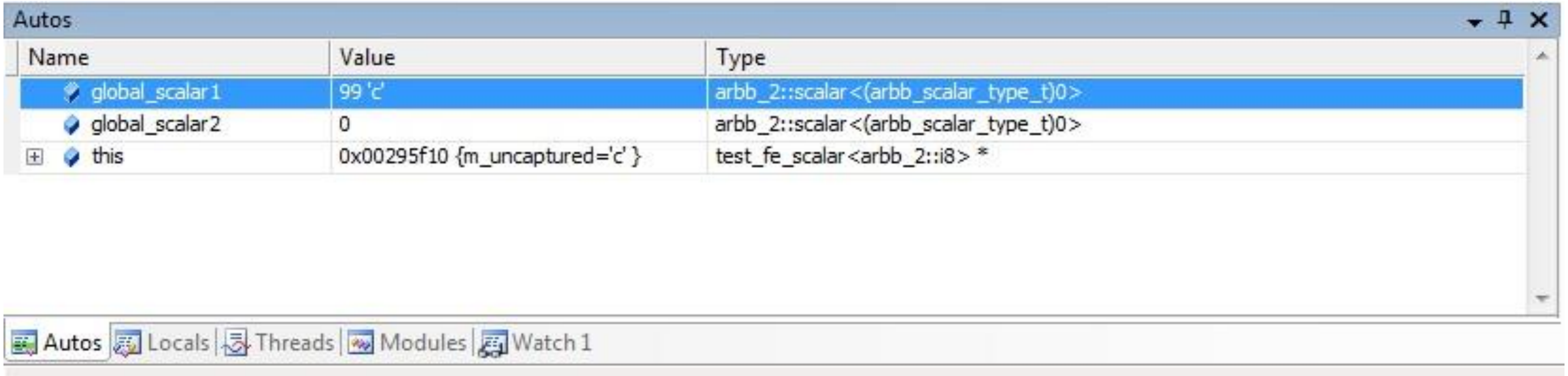
- **Quick watches allow permanent monitoring of ArBB objects**



Screenshots taken from Microsoft* Visual Studio 2008*

Debugger Integration (Visual Studio*)

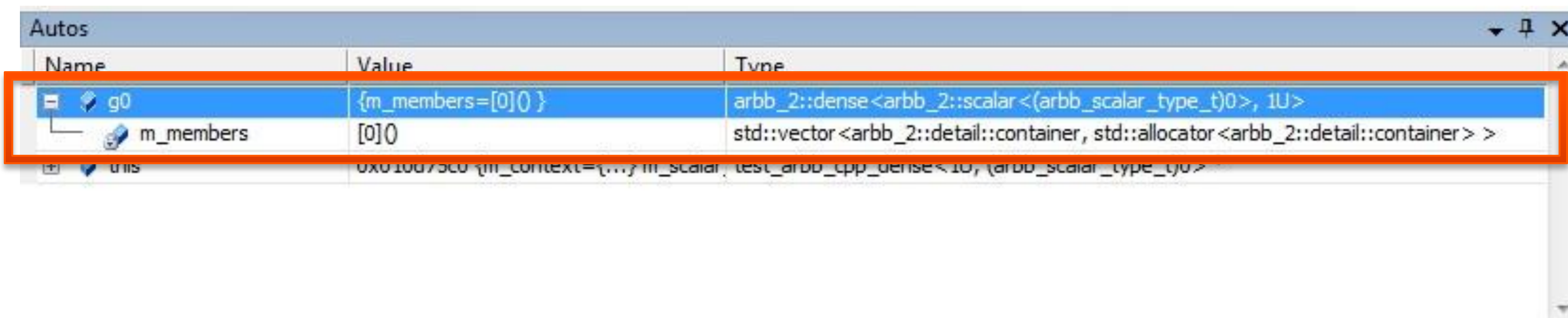
- Display of automatic variables and local variables possible
- Watch points on ArBB objects can be set as usual
- Execution can be stopped when condition becomes true



Screenshots taken from Microsoft* Visual Studio 2008*

Debugger Integration (Visual Studio*)

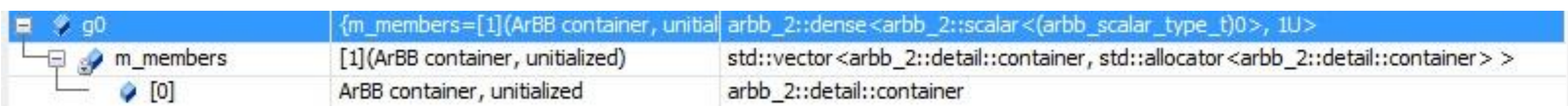
- The current state of a dense container can be monitored
  - Helps track uninitialized data
  - Introspect properties of the container
  - Retrieve current data of a container
- Uninitialized dense containers do not contain any metadata, `m_members` is empty.



Screenshots taken from Microsoft* Visual Studio 2008*

Debugger Integration (Visual Studio*)

- The current state of a dense container can be monitored
  - Helps track uninitialized data
  - Introspect properties of the container
  - Retrieve current data of a container
- Constructed (but not initialized) containers are explicitly indicated:



Screenshots taken from Microsoft* Visual Studio 2008*

Debugger Integration (Visual Studio*)

- The current state of a dense container can be monitored
  - Helps track uninitialized data
  - Introspect properties of the container
  - Retrieve current data of a container
- Metadata and payload of a container can be visualized once initialized (through copy in, binding, assignment)

g0	{m_members=[1](ArBB container [32]) }	arbb_2::dense<arbb_2::scalar<(arbb_scalar_type_t)0>, 1U>
m_members	[1](ArBB container [32])	std::vector<arbb_2::detail::container, std::allocator<arbb_2::detail::container> >
[0]	ArBB container [32]	arbb_2::detail::container
columns	32	__int64
pages	1	__int64
rows	1	__int64
[0]	0	char
[1]	0	char
[2]	0	char
[3]	0	char
[4]	0	char

Screenshots taken from Microsoft* Visual Studio 2008*

Debugger Integration (Visual Studio*)

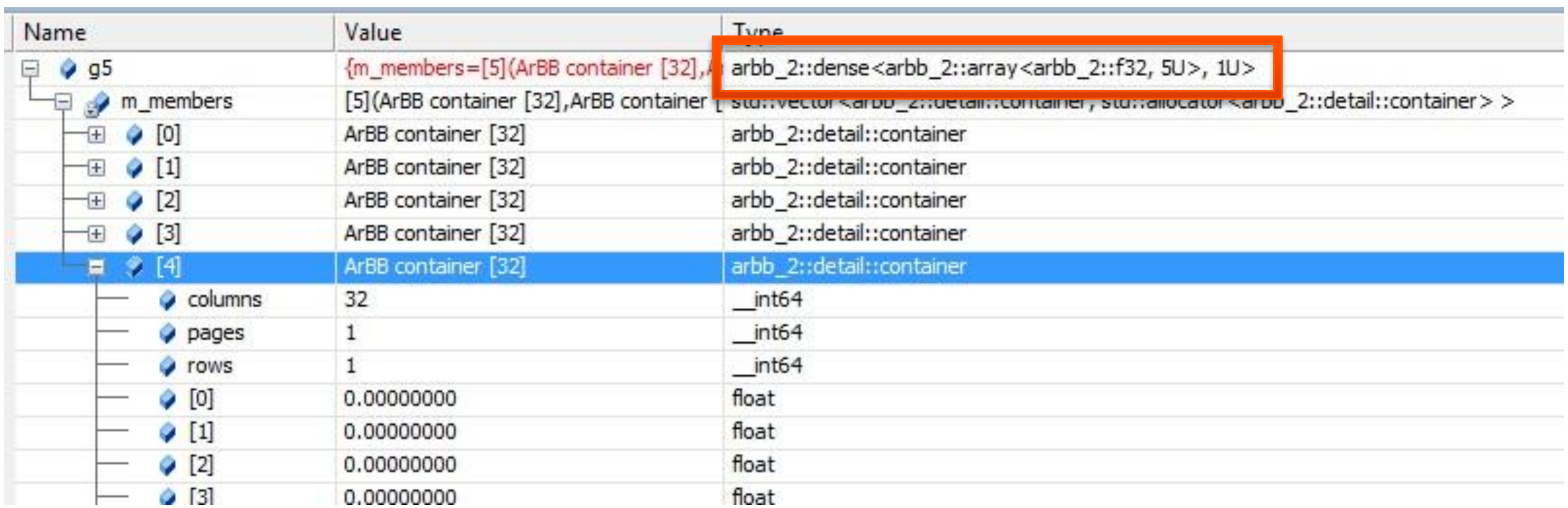
- The current state of a dense container can be monitored
  - Helps track uninitialized data
  - Introspect properties of the container
  - Retrieve current data of a container
- Metadata and payload of a container can be visualized once initialized (through copy in, binding, assignment)
  - 2D or 3D data is flattened

g2	{m_members=[1](ArBB container [8, 4])	arbb_2::dense<arbb_2::scalar<(arbb_scalar_type_t)0>, 2U>
m_members	[1](ArBB container [8, 4])	std::vector<arbb_2::detail::container, std::allocator<arbb_2::detail::container> >
[0]	ArBB container [8, 4]	arbb_2::detail::container
columns	8	__int64
pages	1	__int64
rows	4	__int64
[0]	0	char
[1]	0	char
[2]	0	char
[3]	0	char

Screenshots taken from Microsoft* Visual Studio 2008*

Debugger Integration (Visual Studio*)

- ArBB automatically performs AoS-to-SoA conversions
  - Explicitly visible in the debugging facilities
  - Components of a structured type are scattered into difference containers



Name	Value	Type
g5	{m_members=[5](ArBB container [32],A	arbb_2::dense<arbb_2::array<arbb_2::f32, 5U>, 1U>
m_members	[5](ArBB container [32],ArBB container [	std::vector<arbb_2::detail::container,std::allocator<arbb_2::detail::container> >
[0]	ArBB container [32]	arbb_2::detail::container
[1]	ArBB container [32]	arbb_2::detail::container
[2]	ArBB container [32]	arbb_2::detail::container
[3]	ArBB container [32]	arbb_2::detail::container
[4]	ArBB container [32]	arbb_2::detail::container
columns	32	__int64
pages	1	__int64
rows	1	__int64
[0]	0.00000000	float
[1]	0.00000000	float
[2]	0.00000000	float
[3]	0.00000000	float

Screenshots taken from Microsoft* Visual Studio 2008*

Debugger Integration (GNU Debugger*)

- Immediate mode triggers non-JIT execution of ArBB code
  - No IR recording and JIT compilation involved
  - ArBB execution directly happens in C++ space
- Standard debugger features work as expected (e.g. breakpoints)
- Control flow can directly be monitored through GDB debugger commands
- Note: Capturing and closure creation is not supported
- GDB extension based on Python
  - Python script to pretty-print ArBB data objects
  - Needs GDB version 7.0 or later
  - Blends well with all GDB frontends (e.g. DDD, GNU Emacs)

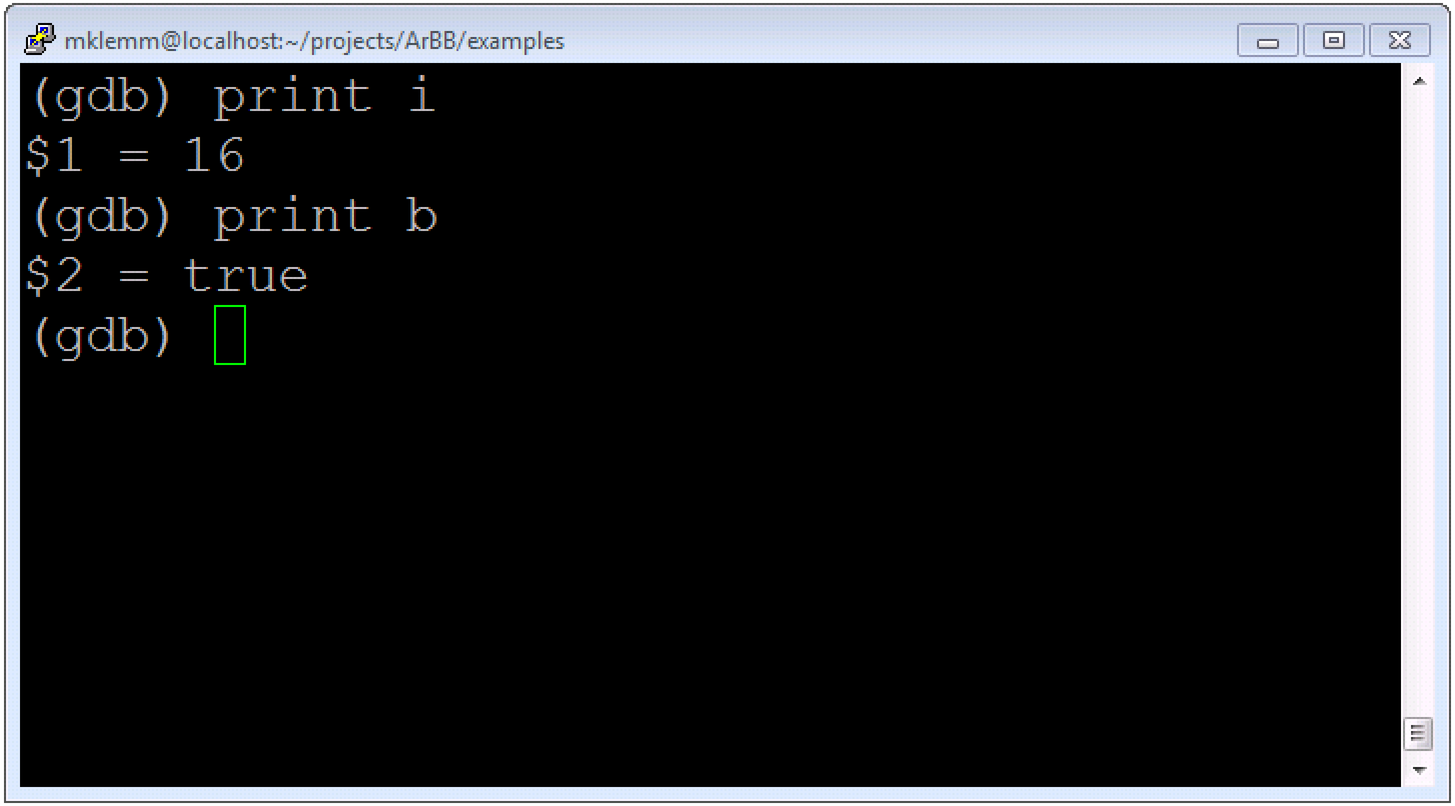
Debugger Integration (GNU Debugger*)

mklemm@localhost:~/projects/ArBB/examples

```
gdb example
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el6)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/mklemm/projects/ArBB/examples/example...done.
(gdb) break example.cpp:17
Breakpoint 1 at 0x4023f7: file example.cpp, line 17.
(gdb) run
Starting program: /home/mklemm/projects/ArBB/examples/example
[Thread debugging using libthread_db enabled]

Breakpoint 1, main (argc=1, argv=0x7fffffffd7d8) at example.cpp:17
17              return 0;
(gdb) list
12              dense<boolean> d2(8);
13              dense<i32, 2> d3(2, 4);
14              dense<i32, 3> d4(2, 2, 2);
15              dense<array<f32, 3>, 1> d5;
16
17              return 0;
18      }
(gdb)
```

Debugger Integration (GNU Debugger*)

- Use the `print` command to print values of scalars



Debugger Integration (GNU Debugger*)

- Use the `print` command to print dense containers
  - Helps track uninitialized data

mklemm@localhost:~/projects/ArBB/examples

```
(gdb) print d1
$3 = ArBB dense, uninitialized = {
  [0] = uninitialized
}
(gdb)
```

Debugger Integration (GNU Debugger*)

- Use the `print` command to print dense containers
  - Inspect properties of the container

mklemm@localhost:~/projects/ArBB/examples

```
(gdb) print d2
$4 = ArBB dense<arbb boolean, 1> = {
  [0] = container (8) = {false, false, fals
e, false, false, false, false, false}
}
(gdb) 
```

Debugger Integration (GNU Debugger*)

- **Use the `print` command to print dense containers**
  - Retrieve current data in a container

```
mklemm@localhost:~/projects/ArBB/examples
(gdb) print d2
$4 = ArBB dense<arbb_boolean, 1> = {
  [0] = container (8) = {false, false, fals
e, false, false, false, false, false}
}
(gdb)
```

Debugger Integration (GNU Debugger*)

- Use the `print` command to print dense containers
  - Retrieve current data in a container

mklemm@localhost:~/projects/ArBB/examples

```
(gdb) print d3
$5 = ArBB dense<arbb_i32, 2> = {
  [0] = container (2, 4) = {[0] = {0, 0}, [
1] = {0, 0}, [2] = {0, 0}, [3] = {0, 0}}
}
(gdb)
```

Debugger Integration (GNU Debugger*)

- Use the `print` command to print dense containers
  - Retrieve current data in a container

mklemm@localhost:~/projects/ArBB/examples

```
(gdb) print d4
$6 = ArBB dense<arbb_i32, 3> = {
  [0] = container (2, 2, 2) = {[0] = {[0] =
 {0,0}, [1] = {0, 0}}, [1] = {[0] = {0,0},
[1] = {0, 0}}}
}
(gdb)
```

Debugger Integration (GNU Debugger*)

- Modify printing behavior standard GDB commands:
  - set print array
  - set print array-indexes
  - set print elements
  - set print pretty

- Please refer to the GDB documentation for a full list

Advanced Intel® ArBB Programming

Performance Optimization

Scoped Timer Facility

- ArBB offers a platform-independent timer facility
  - Measure runtime of ArBB kernels
  - Easy means to profile ArBB code
- The `scoped_timer` class resembles the notion of RAII
  - RAII: “Resource Acquisition Is Initialization”
  - When constructed, the `scoped_timer` takes the current time
  - Upon destruction, the `scoped_timer` takes the current time again
  - It then returns the time difference

Scoped Timer Facility

```
#include <arbb.hpp>
#include <iostream>
using namespace arbb::scoped_time;
using namespace std;

void example() {
    double time;
    {
        const scoped_timer timer(time);

        // run some code in here

    }
    cout << "Time: " << time << "ms" << endl;
}
```

T_{start}

T_{end} time=T_{end} - T_{start}

Shape Expectations

- Knowing data sizes at JIT compile time helps to generate close to optimal code
  - Avoid remainder loops when slicing data into chunks for vectorization and multi-threading
  - Find cache-optimal data distribution
  - Statically pre-allocate memory at the memory manager

- The `expect_size()` call expects an integer expression

```
void fun1024(const dense<T>& a, T& result) {
  expect_size(a,1024);
  // ...
}
```

Shape Expectations

- Constraints:
  - Valid in an ArBB function only
  - The expression needs to evaluate to an integer value
  - The expression may not be an ArBB scalar value:
    - it must be a C++ expression
- The expected size can be a C++ variable
  - The variable is evaluated at IR recording
  - The value is baked into the generated code as a JIT-compile time constant

Best Practices: Memory Management

- Prefer memory managed by ArBB
  - Use "range" interface rather than "bind" when possible
  - This lets ArBB better manage data allocation, do proper alignment, avoid unnecessary copies to/from managed memory, etc.

Binding

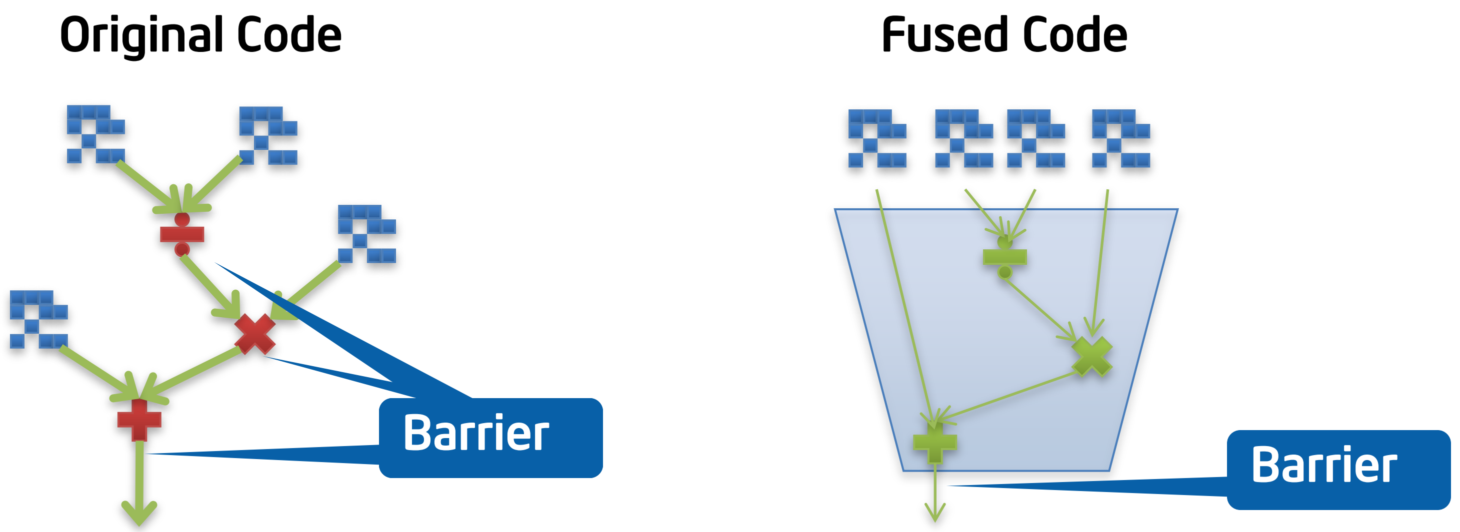
```
dense<f32> a, b;
bind(a, arr, SIZE);
bind(b, brr, SIZE);
// copy in of 'a'
call(fun1)(a, b);
// sync; copy out of 'b'
// copy in of 'a'
call(fun2)(a, b);
// sync; copy out of 'b'
```

Range interface

```
dense<f32> a(1024), b;
// initialize 'a' using write range

// copy in of 'a'
call(fun1)(a, b);
// NO copy out of 'b'
// NO copy in of 'a'
call(fun2)(a, b);
// NO copy out of 'b'
```

Best Practices: Fusion

- ArBB fuses sequences of operations into single blocks
  - Avoids barriers between operations on vectors
  - Avoids expensive temporary containers for intermediate values
  - Necessary intermediate copies can be kept in SIMD registers



At the barrier the intermediate result is ready in an intermediate container.

Best Practices: Fusion

- ArBB internally uses Intel® Threading Building Blocks to implement tasks on top of multi-threading
- Fused operations increase the portion of work per task
  - Higher computational load per task
  - Less task scheduling and threading overhead
  - Less overhead due to synchronization at barriers
- Regularity of operations/primitives matters for fusion:
  - Element-wise — very regular
  - Collective — mostly regular, but subject to barrier
  - Permute: — irregular
  - Facility: — depends

Best Practices: Global Operations

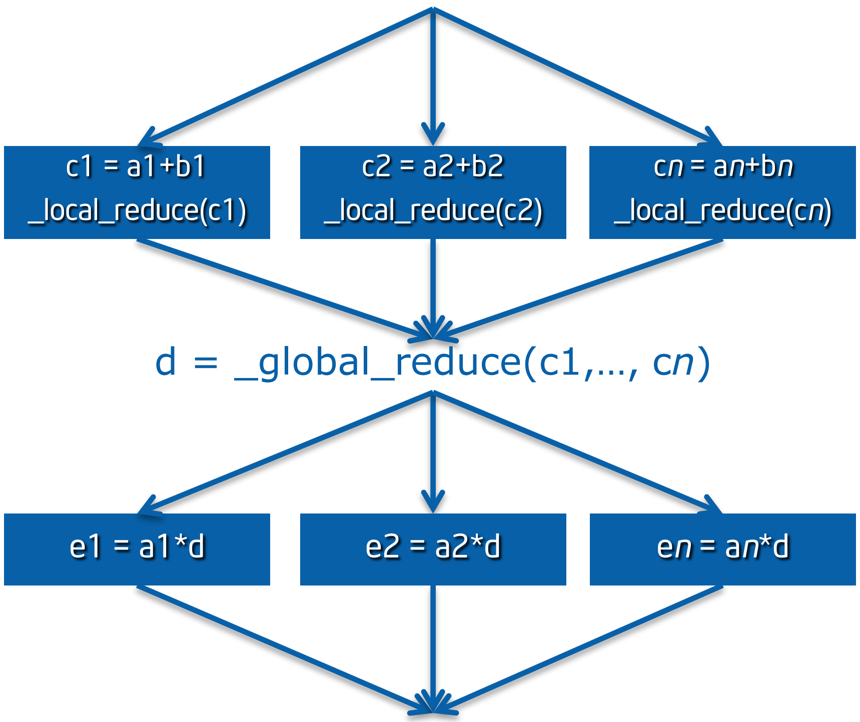
- Avoid global operations (collectives, permutes) if possible
- These operations MAY have barrier-like behavior
  - Threads compute a partial result
  - All partial results are collected into the global result
  - Threads have to wait until the global result is ready
- Remember ArBB's semantics:
  - ArBB built-in primitives execute as serial code
  - Parallelism semantically happens in the operation
- The JIT compiler tries to push collective operations to
  - the begin of the fused code sequence
  - or the end of the fused code sequence

  → avoid (frequent) intermediate barriers

Best Practices: Global Operations

```
void foo(dense<f32> a, dense<f32> b, dense<f32> c,
         dense<f32>& e) {
  c = a + b;
  f32 d = add_reduce(c);
  e = a * d;
}
```

- Parts of a global operation can run in parallel, but a barrier is involved



Best Practices: Shift

- Shifting a container is unlikely to break fusion
- Shift can efficiently mapped to vectorized code without copying the input container:
  - Small scalar loop trip count < SIMD_WIDTH
  - Large SIMD loop proportional to size of container
  - Small scalar loop trip count < SIMD_WIDTH
- Prefer shifting original containers over shifting result containers
  - Introduces a barrier before the shift operation
  - Breaks code fusion because of the barrier
  - May require an intermediate copy for the result container

Best Practices: Shift

- The following code can be fused:

```
dense<f32> in;
result1 = shift(in, i);
result2 = shift(in, j);
```

- JIT compiler does not need to create intermediate copies.
- It is sufficient to only keep the shift distance if in is not changed.

- The following code breaks fusion:

```
dense<f32> in;
result1 = shift(in, i);
result2 = shift(result1, j - i);
```

Introduces a barrier here to wait for completion of the first shift.

Best Practices: Avoid Gather / Scatter

- Avoid scatter() and gather() whenever possible
  - Especially scatter

- Scatter always breaks fusion
  - Scatter is a global and introduces a barrier synchronization
  - It cannot be turned into a gather operation in all cases

- Gather might break fusion
  - Some cases do not involve a barrier
  - In general, a barrier is needed

Best Practices: Large ArBB Functions

- Make ArBB functions as large as possible
- More opportunities for code fusion and other optimizations
  - Keep Amdahl's law in mind (keep fraction of sequential code small)
  - Fuse ArBB functions into a single function
  - Do not transition between C++ space and ArBB space frequently
- Use generative programming to create ArBB kernels
  - No reason to use the `call()` operator in ArBB code
  - Use C++ standard calls to inline function calls
  - Use C++ control flow constructs to generate ArBB kernels
  - Large kernels give more rise to code fusion

Best Practices: Large ArBB Functions

```
void sum_sq_diff(dense<f32> a, dense<f32>& b,
                 f64& result) {
    dense<f64> c = (a - b) * (a - b);
    result = add_reduce(c);
}

void compute_error(dense<f32> a, dense<f32> b,
                   f64& error) {
    f32 sq_error;
    call(sum_sq_diff)(a, b, sq_error);
    error = sqrt(sq_error);
}
```

- Leaves a call instruction in the IR and JIT code
- Less opportunities to fuse code

Recorded code

```
void compute_error(dense<f32> a, dense<f32> b,
                   f64& error) {
    f32 sq_error;
    _ir_call(sum_sq_diff)(a, b, sq_error)
    error = sqrt(sq_error);
}
```

Best Practices: Large ArBB Functions

```
void sum_sq_diff(dense<f32> a, dense<f32>& b,
                 f64& result) {
    dense<f64> c = (a - b) * (a - b);
    result = add_reduce(c);
}

void compute_error(dense<f32> a, dense<f32> b,
                   f64& error) {
    f32 sq_error;
    sum_sq_diff(a, b, sq_error);
    error = sqrt(sq_error);
}
```

- Always inlines sum_qs_diff
- More opportunities to fuse code

Recorded code

```
void compute_error(dense<f32> a, dense<f32> b,
                   f64& error) {
    f32 sq_error;
    dense<f64> c = (a - b) * (a - b);
    sq_error = add_reduce(c);
    error = sqrt(sq_error);
}
```

Best Practice: Use Most Specific Function

- Use the most specific function possible to solve a problem

- Give rise to the JIT compiler and the runtime system for better optimization
  - More generic functions are more difficult to implement internally
  - Very specific functions contain the most context knowledge possible

- Use less operations to express the algorithm
  - Higher computational load per operator application
  - Better chance for code fusion

Best Practice: Use Most Specific Function

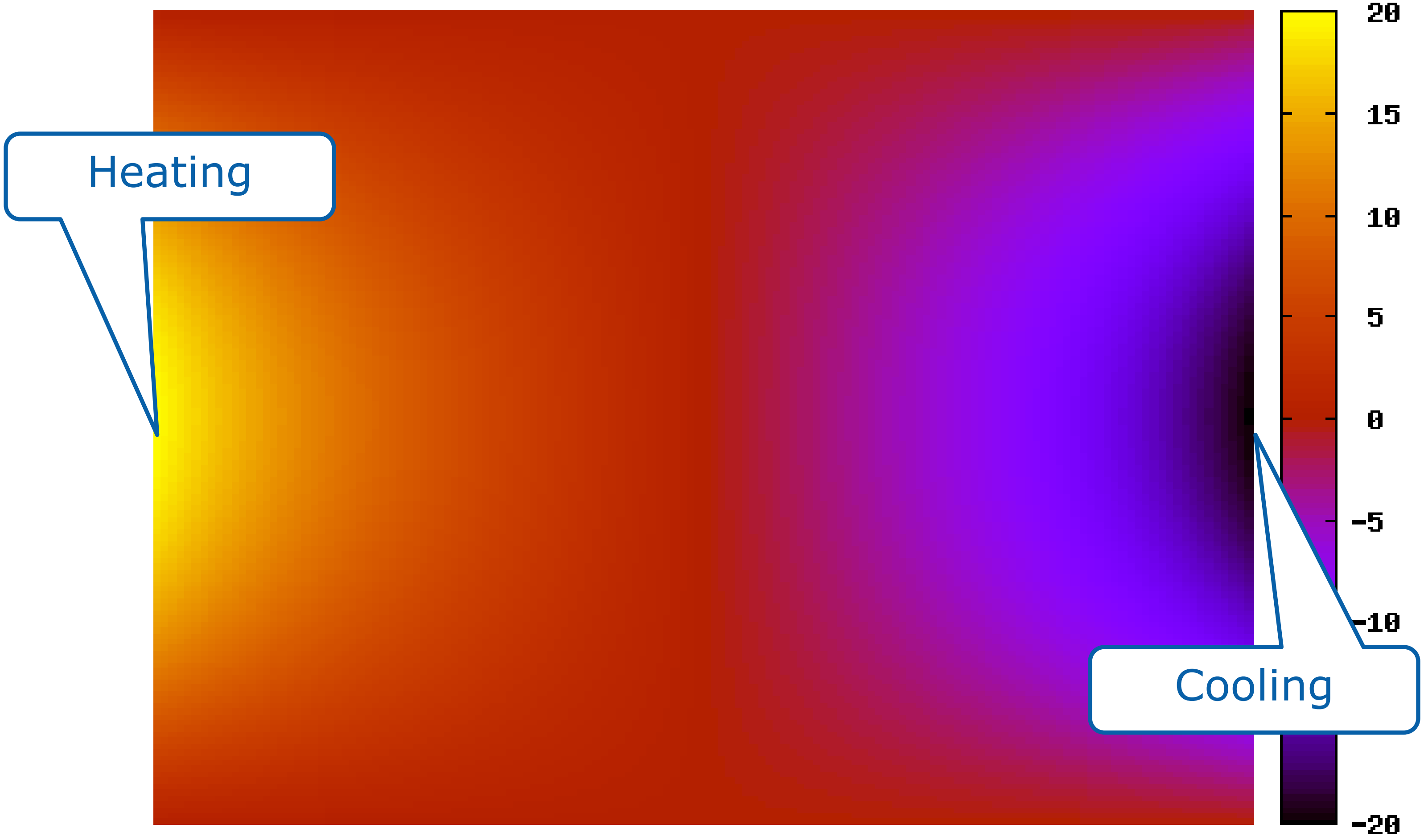
Example: Limit values of a container to a given range

```
dense<f32> x = …;

x = select(x < 0.0f, 0.0f, x);          // Bad
x = select(x > 255.f, 255.f, x);


x = max(0.0f, x)                        // Better
x = min(255.f, x);


x = clamp(x, 0.0f, 255.f);              // Optimal solution
```

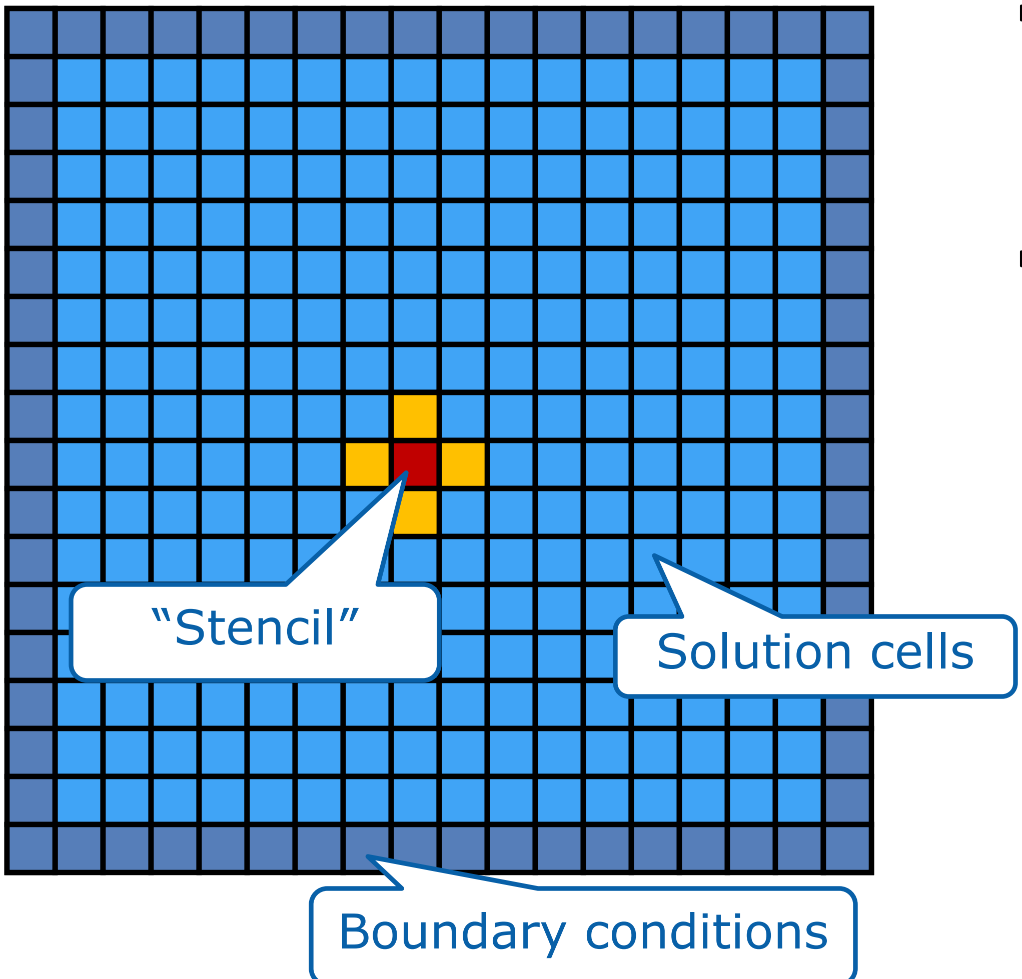
Advanced Intel® ArBB Programming

Final Example: A Stencil Code

Heat Dissipation Example



Heat Dissipation Example (Algorithm)



- **Data structure:**
  - 2D grid (N x M cells)
  - Boundary cells
- **Algorithm:**
  - Sweep over the grid
  - Update non-boundary cells
    - Read cells N, S, E, and W of the current cell
    - Take the average of the value

Heat Dissipation Example (C/C++)

```
void run(double** grid1, double** grid2) {
    for (int iter = 0; iter < ITERATIONS; iter++) {
        step(grid1, grid2);
        tmp   = grid1;
        grid1 = grid2;
        grid2 = tmp;
}   }
```

Run ITERATIONS sweeps over the 2D grid.

After each sweep, swap source and destination grid.

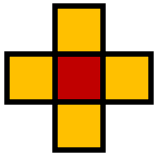
```
void step(double** src, double** dst) {
    for (int i = 1; i < SIZE-1; i++) {
        for (int j = 1; j < SIZE-1; j++) {
            dst[i][j] = 0.25*(src[i+1][j] + src[i-1][j]+
                              src[i][j+1] + src[i][j-1]);
}   }   }
```

For each grid cell...

... apply stencil.

Heat Dissipation Example (ArBB)

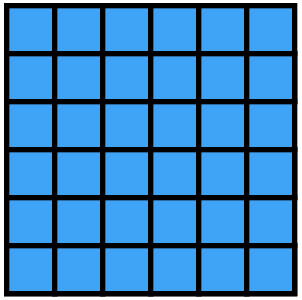
```
void stencil(f32& value) {
    f32 north = neighbor(value,  0, -1);
    f32 south = neighbor(value,  0, +1);
    f32 west  = neighbor(value, -1,  0);
    f32 east  = neighbor(value, +1,  0);
    value = 0.25f * (north + south + west + east);
}
```



- The stencil averages neighbors in north, south, east, and west.
- Note:
  - The stencil uses a single parameter for both input and output
  - The ArBB runtime and memory manager take care of the shadow copy

Heat Dissipation Example (ArBB)

```
void apply_stencil(usize niterations,
                   dense<f32, 2>& grid) {
  _for (usize i = 0, i != niterations, ++i) {
      map(stencil)(grid);
  } _end_for;
}
```



- An ArBB `_for` implements the iterative application of the sweeps on the grid
- The `map()` operator applies the stencil for each solution cell
- Worth to repeat:
  - The stencil uses a single parameter for both input and output
  - The ArBB runtime and memory manager take care of the shadow copy

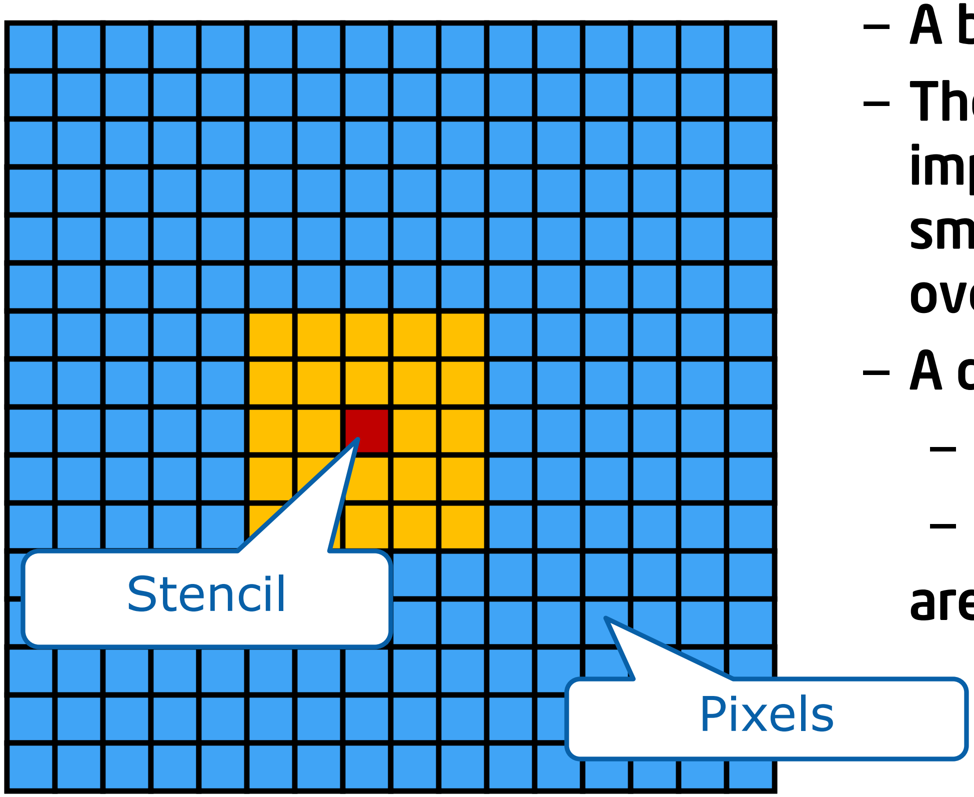
Blur Filter Example

Original photo:

Blurred photo:

Blur Filter Example

- Blur filters usually are stencil computations



- A bitmap is a “2D grid of pixels”
- The heat dissipation example also implemented a “blur” effect by smoothing the heat distribution over the solution grid
- A close look reveals that
  - grid traversal
  - stencil application

  are orthogonal.

- What about code re-usage?

Blur Filter Example (Pseudo C/C++)

```
void run(rbga** grid1, rbga** grid2, funcptr stencil) {
    for (int iter = 0; iter < ITERATIONS; iter++) {
        step(grid1, grid2, stencil);
        tmp = grid1; grid1 = grid2; grid2 = tmp;
}   }

void step(rbga** src, rbga** dst, funcptr stencil) {
    for (int i = offset; i < SIZE-offset; i++) {
        for (int j = offset; j < SIZE-offset; j++) {
            stencil(i, j, src, dst);
}   }   }

void stencilA(int x, int y, rbga** src, rbga** dst) {
    dst[i][j] = ...; // imagine a complicated stencil formula here
}
```

Usage: `run(input, output, stencilA)`

Blur Filter Example (Pseudo C/C++)

```
void run(rbga** grid1, rbga** grid2, funcptr stencil) {
    for (int iter = 0; iter < ITERATIONS; iter++) {
        step(grid1, grid2, stencil);
        tmp = grid1; grid1 = grid2; grid2 = tmp;
}   }

void step(rbga** src, rbga** dst, funcptr stencil) {
    for (int i = offset; i < SIZE-offset; i++) {
        for (int j = offset; j < SIZE-offset; j++) {
            stencil(i, j, src, dst);
}   }   }

void stencilA(int x, int y, rbga** src, rbga
    dst[i][j] = ...; // imagine a complicated stencil form
}
```

Usage: `run(input, output, stencilA)`

What about...
1D, 2D, 3D...
CYMK...

- Generally a bad idea...
- Compilers might not inline the stencil function
- $O(n^2)$ function calls $\rightarrow$ overhead

Generic Stencil Framework (ArBB)

Data Abstraction through Template Type Arguments

- We can get rid of the explicit data type of the heat dissipation example by using a template type argument:

```
template<typename T>
void stencil(T& value) {
    const T north = neighbor(value,  0, -1);
    const T south = neighbor(value,  0, +1);
    const T west  = neighbor(value, -1,  0);
    const T east  = neighbor(value, +1,  0);
    // TODO: implicit type conversions and overflows
    value = (north + south + west + east) / 4;
}
```

Generic Stencil Framework (ArBB)

Data Abstraction through Template Type Arguments

```
template<typename T>
void generic_stencil(usize niterations,
                     dense<T, 2>& grid) {
    _for (usize i = 0, i != niterations, ++i) {
        map(stencil<T, 2>)(grid, /* additional arguments */);
    } _end_for;
}
```

Will be become important later.

Possible instantiations of the stencil code:

- Heat dissipation solver, T = f64:

```
dense<f64, 2> grid;
generic_stencil(niter, grid);
```

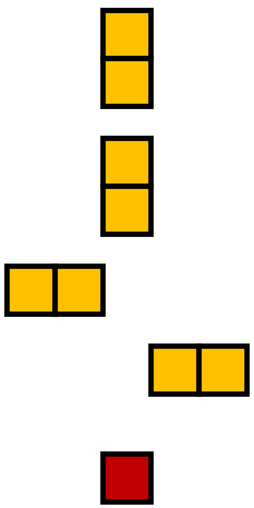
- Blur filter, RGBA bitmap, T = rgba:

```
typedef array<u8, 4> rgba;
dense<rgba, 2> grid;
generic_stencil(niter, grid);
```

Generic Stencil Framework (ArBB)

Abstraction from Stencil Implementations

```
size_t radius;
template<typename T>
void stencil(T& value, usize height, usize width) {
    array<usize, 2> p; position(p);
    array<usize, 2> s; s[0] = width; s[1] = height;
    _if (all(radius <= p && p < s - radius))  {
        value -= value;
        for (int w = 1; w <= radius; ++w) {
            value += neighbor(value,  0, -w);
            value += neighbor(value,  0,  w);
        }
        for (int h = 1; h <= radius; ++h) {
            value += neighbor(value, -h,  0);
            value += neighbor(value,  h,  0);
        }
        value /= (4 * radius);
    } _end_if;
}
```



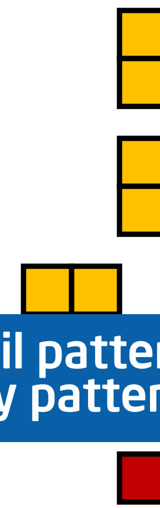
Generic Stencil Framework (ArBB)

Abstraction from Stencil Implementations

```
size_t radius;
template<typename T>
void stencil(T& value, usize height, usize width) {
    array<usize, 2> p; position(p);
    array<usize, 2> s; s[0] = width; s[1] = height
    _if (all(radius <= p && p < s - radius))  {
        value -= value;
        for (int w = 1; w <= radius; ++w) {
            value += neighbor(value,  0, -w);
            value += neighbor(value,  0,  w);
         }
         for (int h = 1; h <= radius; ++h) {
            value += neighbor(value, -h,  0);
            value += neighbor(value,  h,  0);
         }
        value /= (4 * radius);
     } _end_if;
}
```

scalar or an array of scalars.

Without the center's value.



Create stencil pattern along x and y pattern.

Compute avg. on T type.

Generic Stencil Framework (ArBB)

Abstraction from Stencil Implementations

```
size_t radius;
Template<typename T, size_t D>
void stencil(T& value, array<usize, D> size) {
    array<usize, D> p; position(p);
    _if (all(radius <= p && p < size - radius))  {
        value -= value;
        array<isize, D> offset;
        for (size_t d = 0; d != D; ++d) {
            offset.fill(0);
            for (int r = 1; r <= radius; ++r) {
                offset[d] = r;
                value += neighbor(value,  offset);
                value += neighbor(value, -offset);
        }   }
        value /= (2 * D * radius);
    } _end_if;
}
```

Generic Stencil Framework (ArBB)

Abstraction from Stencil Implementations

```
size_t radius;
Template<typename T, size_t D>
void stencil(T& value, array<usize, D> size) {
    array<usize, D> p; position(p);
    _if (all(radius <= p && p < size - radius))  {
        value -= value;
        array<isize, D> offset;
        for (size_t d = 0; d != D; ++d) {
            offset.fill(0);
            for (int r = 1; r <= radius; ++r) {
                offset[d] = r;
                value += neighbor(value,  offset);
                value += neighbor(value, -offset);
        }   }
        value /= (2 * D * radius);
    } _end_if;
}
```

Dimensionality D of the grid.

The array contains the stencil offset along all dimensions from the stencil's center.

Set all offsets to zero.

Assign offset along x, y (2D), and z (3D).

Generic Stencil Framework (ArBB)

Abstraction from Stencil Implementations

- Note:
  - The given stencil does not compute a Gaussian blur on a bitmap
  - For the sake of presentation, only a cross-shaped stencil was implemented
  - Approach can be extended to arbitrary (weighted) stencils
  - Can even use C++ control flow to handle special cases (zero, one, symmetry) for weights efficiently

- The global variable `radius` can be removed:
  - Involves some more C++ magic for the current version of ArBB
  - Future version of ArBB might support non-static class members

Questions?

Best Practice: Convert Branches to Masks

- Replace sequential control flow by mask
  - Use select when possible instead of `_if`
  - Avoid expensive global operations in `_if` statements with small branch bodies

- The _if statement introduces control flow
  - Branches can be expensive in tight loops
  - Branches can be difficult to vectorize / parallelize

- Also try to use `if` instead of `_if`
  - Avoid control at runtime
  - Baked in control flow branches at JIT compilation time

Best Practice: Convert Branches to Masks

Sequential code:

```
for (int i = 0; i < SIZE; i++ ) {
    if (src[i] < SOME_VALUE)
        dst[i] = src[i] * 2;
    else
        dst[i] = src[i] / 2;
}
```

ArBB code:

```
dense<T> src = …;
dense<boolean> mask =  src < SOME_VALUE;
dst = select(mask, src * 2, src / 2);
```

- Select evaluates both branches
- "Wasted" computation proportional to container size