VSIPL++: A High-Level Programming Model for Productivity and Performance

Dr. Brooks Moses

2012-05-15

Outline

- Performance and Portability Matter
- Introduction to Sourcery VSIPL++
- Portable, High-Performance Programming
  - Building a Wall
  - Examples in VSIPL++ Applications
  - Examples in the Sourcery VSIPL++ Library
- Portability in Practice: A Radar Benchmark
- Summary and Conclusions

Performance Matters

Systems limited by size, weight, power, and cost.

More computation means you can do more:

- Better images with the same radar antenna
- More accurate target recognition
- Faster turnaround on medical data
- More realistic and detailed virtual worlds

Performance is efficiency: How much computation can you do the given hardware?

Portability Matters

Software products last for decades…

- Investment is too high to throw away
- Longevity is a competitive advantage

…But hardware generations only last a few years.

- New architectures at the technology leading edge

  (How many of you are writing Cell/B.E. code?)
- Performance tuning changes with existing architectures

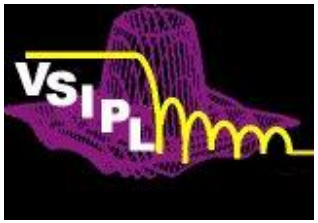
Portability is productivity: If it's portable, you don't need to think about hardware details when writing it.

Introduction to Sourcery VSIPL++

The Sourcery VSIPL++ Library

VSIPL++: Open standard API for a "**V**ector, **S**ignal, and **I**mage **P**rocessing **L**ibrary in C**++**".

- High-level library for many kinds of embedded high-performance computing
- Designed for portability across platforms



Standardization History (2001-present)

- Developed by HPEC Software Initiative
- Builds on earlier C-based VSIPL standard
- Submitted to Object Management Group

Sourcery VSIPL++ is a high-performance implementation from Mentor Graphics

CodeSourcery

mentor embedded

Sourcery VSIPL++ Platforms

Supported Platforms

- x86
  - SIMD support: SSE3, AVX
  - Libraries: IPP/MKL, ATLAS, FFTW
- Power Architecture
  - SIMD support: AltiVec
  - Libraries: ATLAS, FFTW, Mercury SAL
- Cell Broadband Engine
  - Processing Elements: PPE and SPE
  - Libraries: CodeSourcery Cell Math Library
- **NVIDIA GP-GPU + x86 CPU**
  - Libraries: CuBLAS, CuFFT, CULAtools
- Other
  - Custom development services available to meet your high performance computing needs

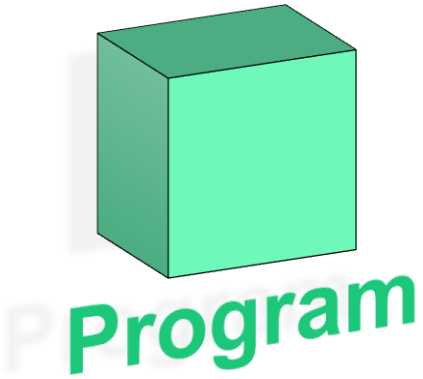
How to Write Portable, High-Performance Software

Portable High-Performance Programming

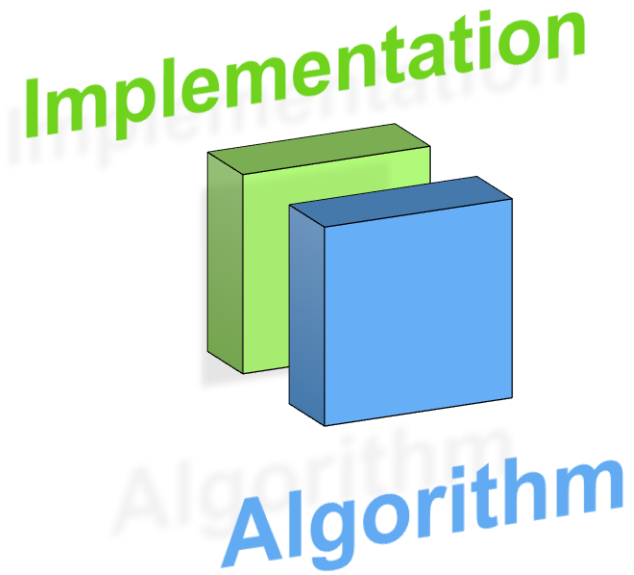
It’s all about building the right wall.

Berkel Gate, Zutphen, NL (Wikimedia Commons)

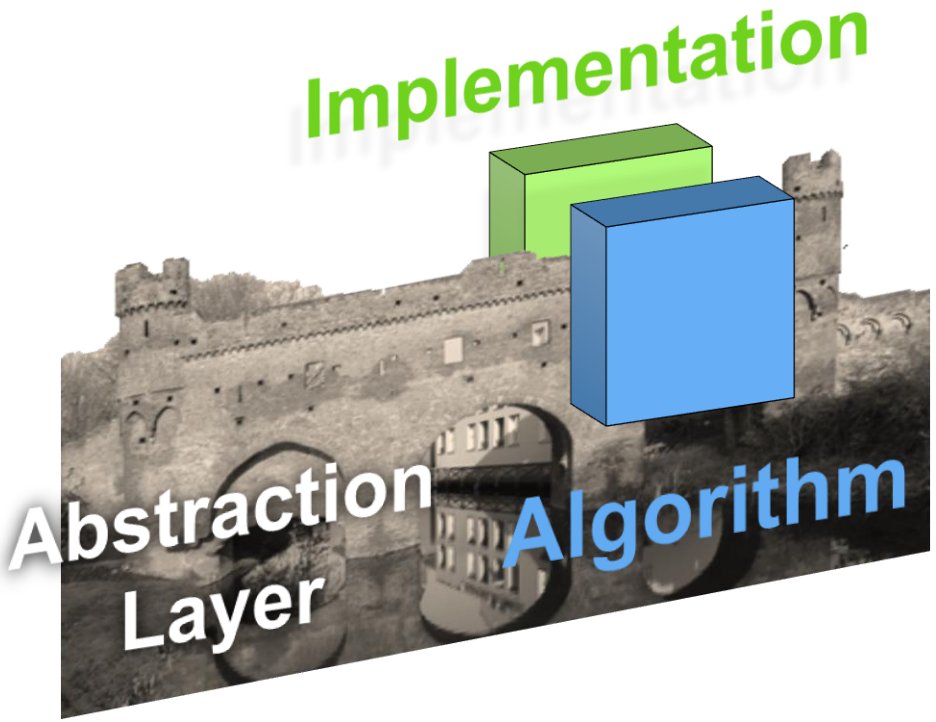
Portable High-Performance Programming



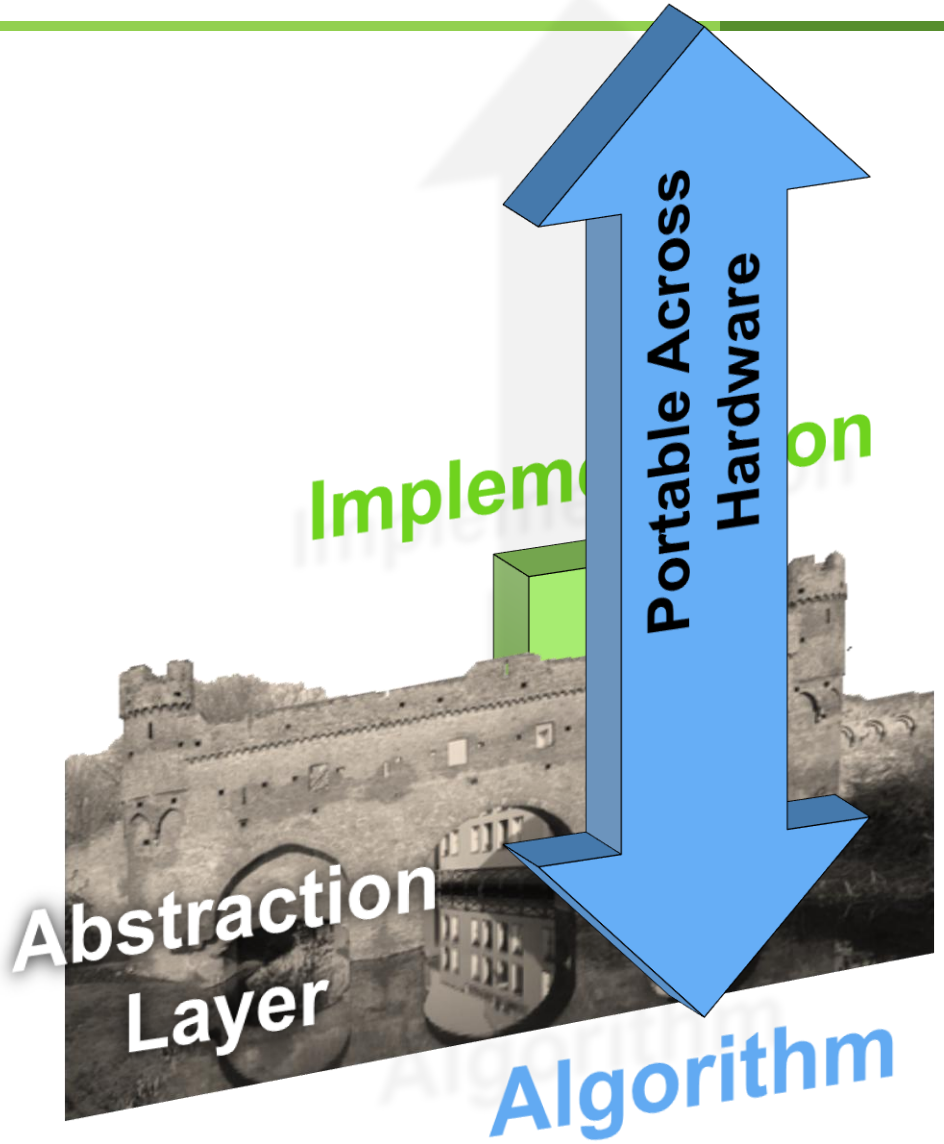
Portable High-Performance Programming



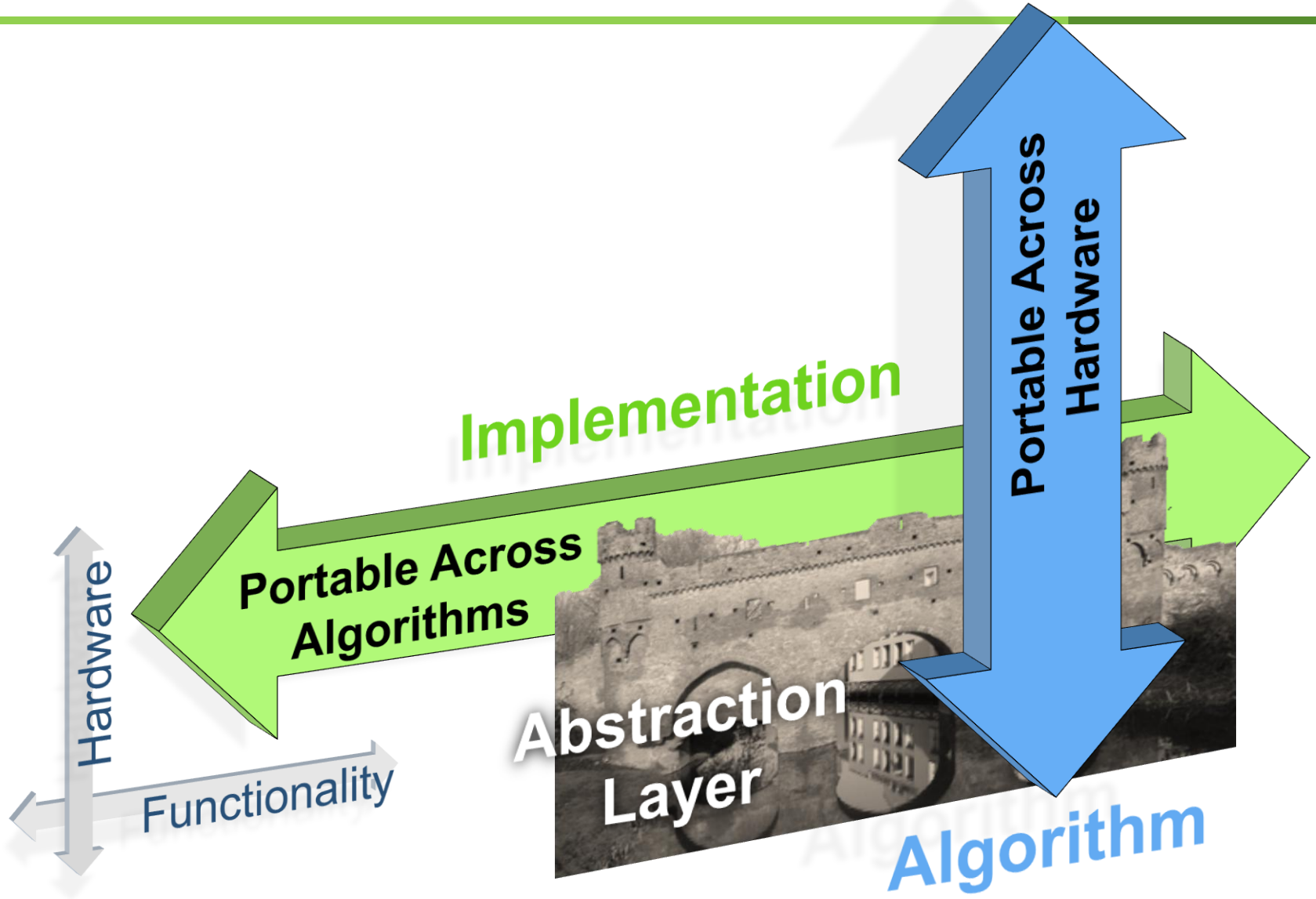
Portable High-Performance Programming



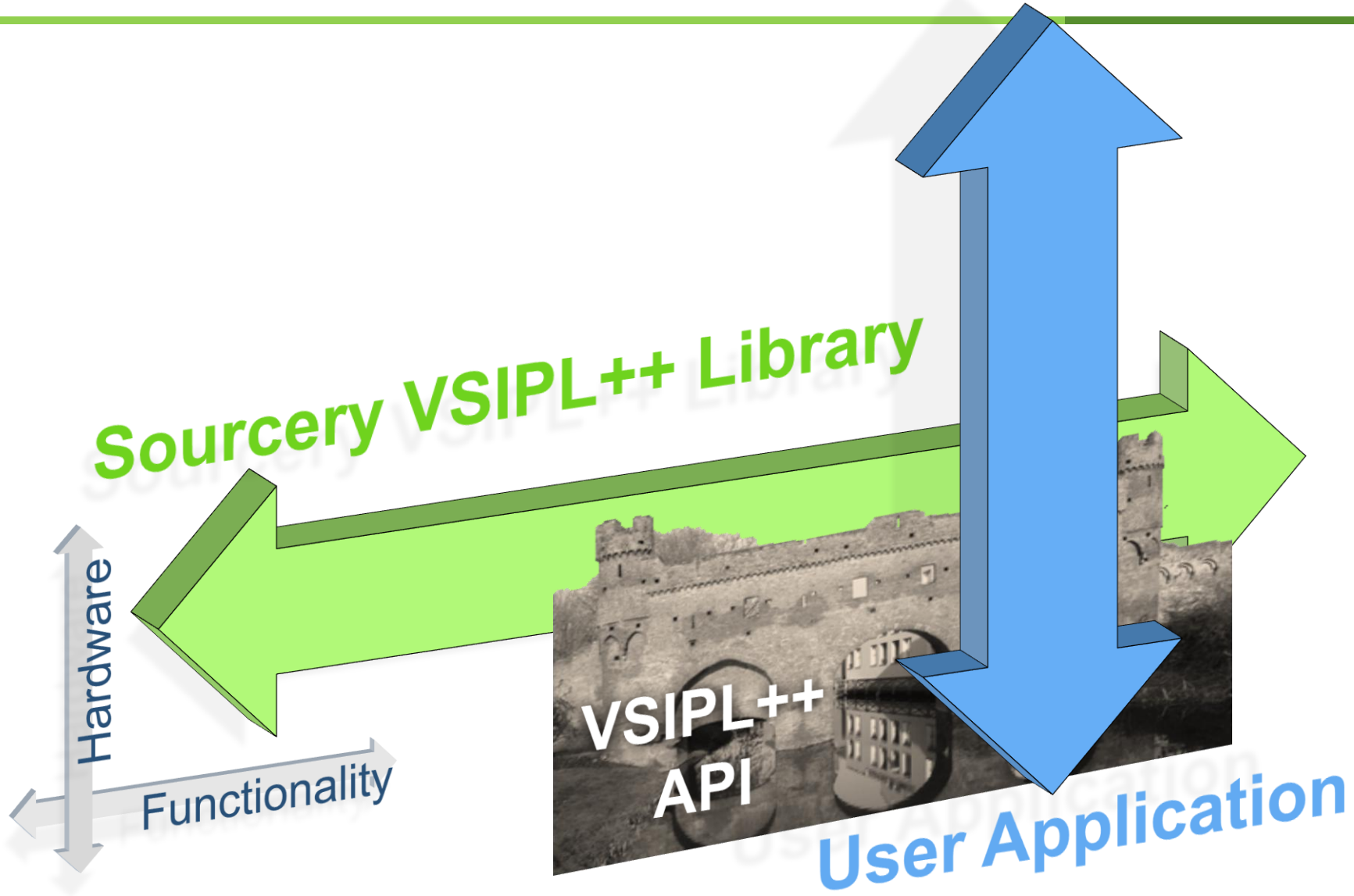
Portable High-Performance Programming



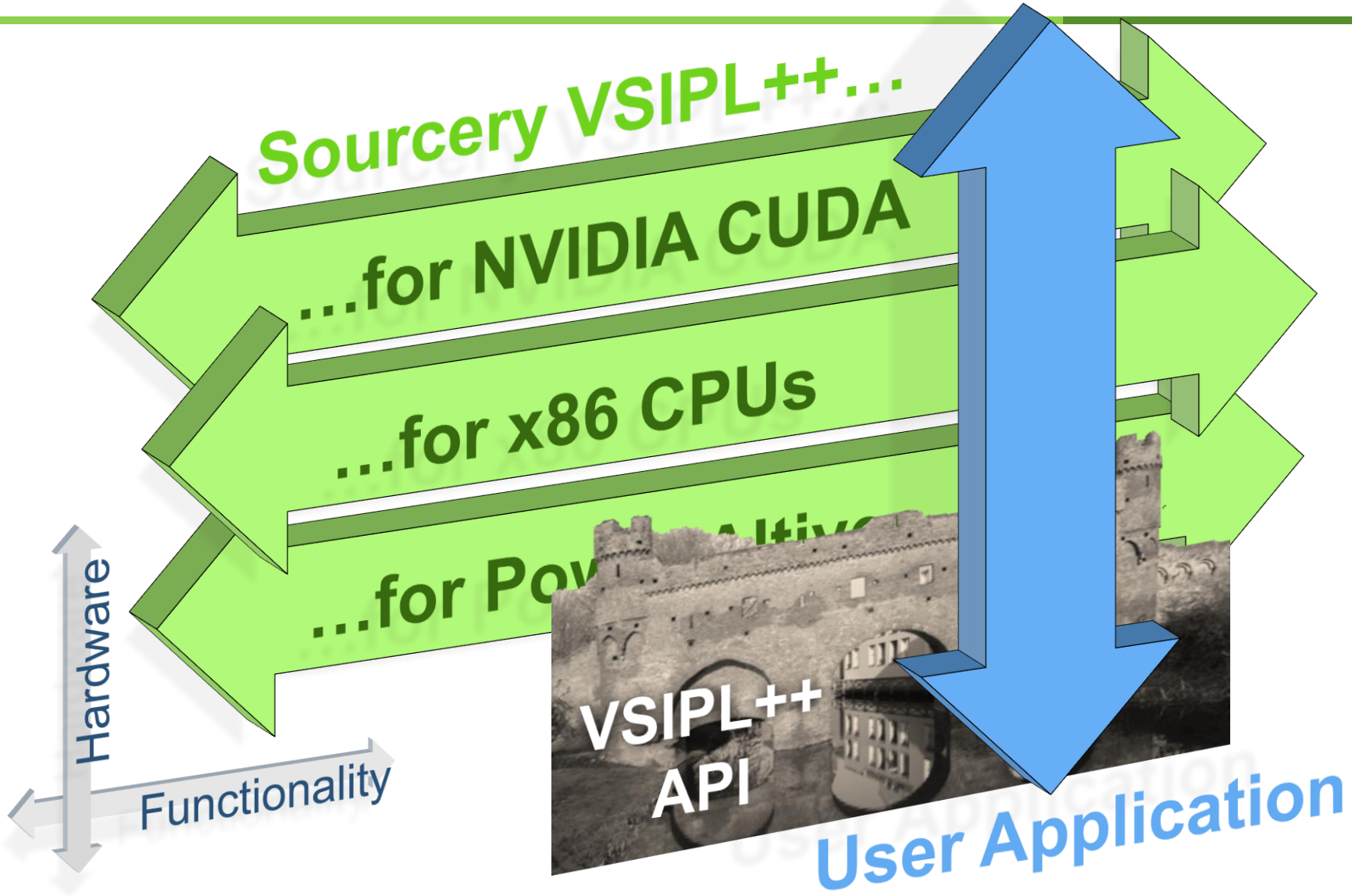
Portable High-Performance Programming



Portable High-Performance Programming



Portable High-Performance Programming



Portable, High-Performance Programming

High-Level Algorithm

- Hardware-independent
- Written in expressive, high-level language
- Can be reused with many different implementations

Abstraction Layer

Low-Level Implementation

- Hardware-specific
- Written in detail-oriented specific language
- Can be reused with many different algorithms

Abstraction Layer Requirements

What makes a good wall?

- Expresses the right ideas for the domain
  - Contains all the building blocks that algorithms will need.
  - Algorithms can be built from small number of blocks.
- Defines operations in ways that can be implemented effectively on the hardware.
  - Each building block is a large chunk of work.
- Low latency crossing the wall
  - Doesn't add performance cost.

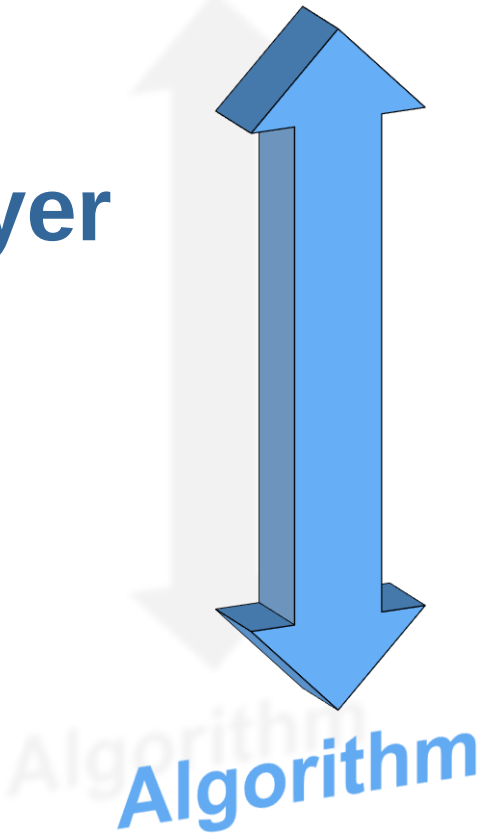
Abstraction Layer Requirements

What makes a good wall?

- Lets the programmer get through to the hardware when needed
  - All abstraction layers leak
  - Plan for it, rather than avoiding it
- Usability
  - This is a whole different talk!

The VSIPL++ Abstraction Layer

(The “application” side of the wall)



Programming Language

Why does the VSIPL++ API use C++?

It's not too exotic to be usable in real applications:

- Plays well with existing C code
- Not dependent on a specific compiler or toolchain
- Portable to nearly all hardware platforms

But we do need some things C doesn't provide:

- Encapsulates abstractions as objects
- Programmable intelligence at compile time

C++ is the only language that combines all of these features.

Data Encapsulation

The data model is of critical importance.

Data representation is a hardware-level detail

- Location in memory space (system, device, pinned, etc.)
- Layout (complex split/interleaved, etc.)
- Some data may not be stored at all, but computed on demand or awaiting asynchronous computation.

Algorithmic code should just see “data”.

- Code should not need to change because data is moved

(Sometimes the algorithm does need to give hints)

Data Encapsulation

VSIPL++ uses a two-level model:

- Algorithm sees data as “Views”: generic Vector, Matrix, or Tensor objects.
- Implementation sees data as “Blocks”: specific data types that describe the representation.

Views are smart pointers to blocks.

- Allocate and de-allocate by reference counting.
- Use C++ template parameters to indicate their block type and allow efficient (non-generic) code generation.

Function Set

Functions in the API need to be large building blocks, and expressive enough to cover algorithms.

For much of the VSIPL++ domain, there are commonly-accepted sets of functions:

- Elementwise operations on arrays
- Linear algebra (BLAS, LAPACK)
- Fast Fourier Transforms
- and some signal-processing-specific functions: convolutions, correlations, FIR/IIR filters, etc.

We can also assemble chunks from multiple functions (see later!).

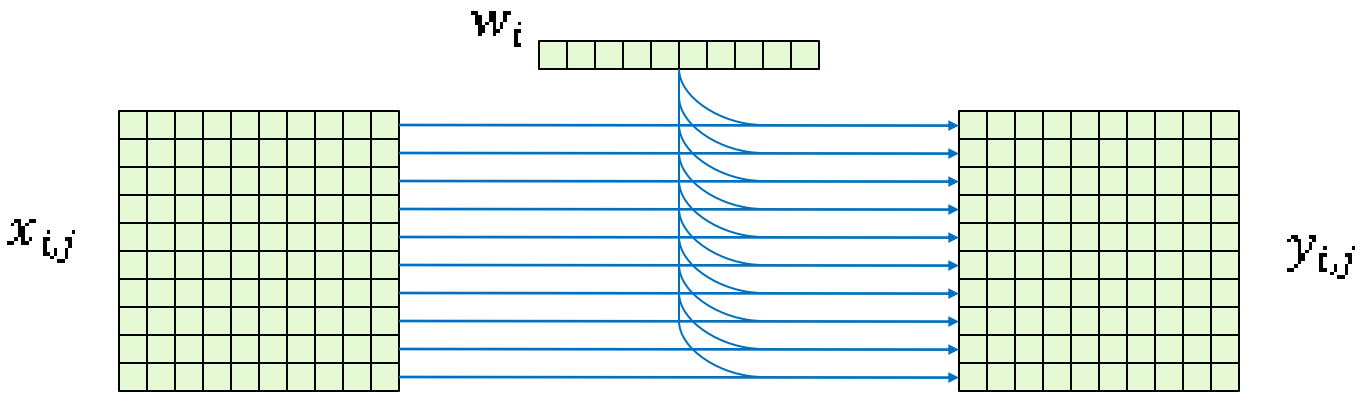
Explicit Parallelism

We can do a lot with data-parallelism within function calls, but sometimes that's not enough. Users can also specify parallelism explicitly:

- Cross-process data parallelism
  - Data is distributed across multiple processes (which may be on different machines, or using different GPUs)
  - Users explicitly describe data locality.
- Task parallelism
  - Users define asynchronous tasks which can be pipelined and executed concurrently.

An Example: Fast Convolution Algorithm

- Radar data in Matrix rows
- Convolve each row with a constant weight vector



- Algorithm: “Fast” (frequency-domain) Convolution

$$X_i = FFT(x_i)$$
$$W = FFT(w)$$

Fourier Transform

$$Y_{i,j} = W_i X_{i,j}$$

Elementwise Multiply

$$y_i = FFT^{-1}(Y_i)$$

Inverse Fourier Transform

An Example: Fast Convolution Code

The algorithm $conv(x_i, w) = FFT^{-1}\left(FFT(w)FFT(x_i)\right)$

Conventional C code implementation

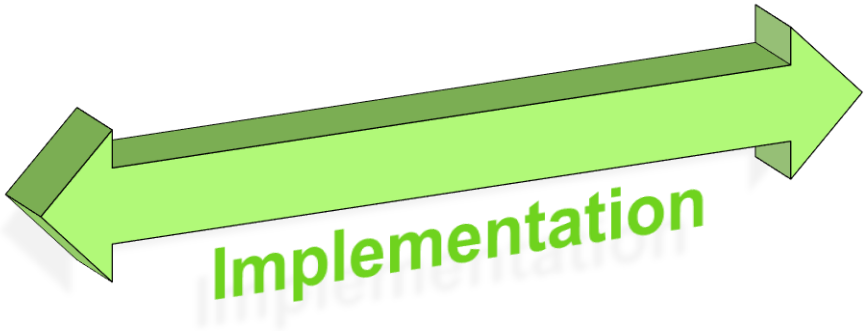
```
void convolveTest3x3( float *imageIn, float *imageOut,  int nc, int nr, float *k ) {
    int ncm1 = nc - 1, nrm1 = nr - 1;
    float s = k[ 0 ] + k[ 1 ] + k[ 2 ] + k[ 3 ] + k[ 4 ] + k[ 5 ] + k[ 6 ] + k[ 7 ] + k [ 8 ]; // Sum kernel elements so can gen
normalized variables
    float k00 = k[ 0 ]/s, k01 = k[ 1 ]/s, k02 = k[ 2 ]/s, k10 = k[ 3 ]/s, k11 = k[ 4 ]/s, k12 = k[ 5 ]/s, k20 = k[ 6 ]/s, k21 = k[ 7 ]/s,
k22 = k[ 8 ]/s;
    for ( int i = 1; i < nrm1; i++ ) {
        float *r00 = imageIn + ( i - 1 ) * nc;
        float *r01 = r00 + 1;
        float *r02 = r01 + 1;
        float *r10 = r00 + nc;
        float *r11 = r10 + 1;
        float *r12 = r11 + 1;
```

VSIPL++ Implementation

```
typedef complex<float> T
Matrix<T> input(M, N), output(M, N);
Vector<T> weights(N);
/* ... assign values to input, output, weights ... */

Fftm<T, T, row, fft_fwd> fwd(Domain<2>(M, N), 1.);
Fftm<T, T, row, fft_inv> inv(Domain<2>(M, N),  1./N);
Fft<T, T, fft_fwd> weight_fft(Domain<1>(N), 1.);

weight_fft(weights);  /* Do in-place FFT */

output = inv(vmmul<row>(weights, fwd(input)));  /* convolve */
```

Sourcery VSIPL++ Optimizations

(The “implementation” side of the wall)



Data Movement and Caching

Critical GPU optimization: Avoid extra data movement!

With encapsulated data, we don't have to make unnecessary promises about where the data is.

- User doesn't have a host/device pointer to the data.

Sourcery VSIPL++ uses caching:

- Data is moved to GPU when needed there, and cached in case it is needed again.
- Data computed on GPU stays on GPU, until it is needed back on the CPU.

Data is only moved as many times as necessary.

Leveraging Existing Libraries

There are many good hardware-specific libraries that would be difficult to replicate:

- On the x86: Intel's IPP and MKL
- On the GPU: CUBLAS, CUFFT, CULA, …

Sourcery VSIPL++ uses these, abstracted behind the common hardware-independent API.

- User just passes encapsulated data to VSIPL++ function
- We determine from data size, locality, etc., which implementation is best to use.
- Then we move the data if needed, and call specific library.

Expression Fusion

Consider a nested array expression:

```
C = sqrt(square(A) + square(B));
```

In Sourcery VSIPL++, we use expression fusion:

- Uses C++ template metaprogramming for code generation.
- The functions do no computation.
- Their return types encode the expression tree.
- The assignment operator unwinds the expression tree and performs the computation.

For this example, we end up with one single loop.

More info: talk by Jonathan Cohen, GTC 2010

Profiling

Magic is all well and good, but sometimes the user needs to know what's really going on.

- Improve code so we can do something better with it.
- Add "hints" if we are making bad choices.
- Understand and trust results.

Sourcery VSIPL++ includes built-in code profiling:

- Data transfers
- Function choices
- Performance timing

Allows a feedback loop of testing and improving code.

Direct access to data and hardware

(When you need to get through the wall)

Directly accessing data and hardware

Sometimes the user does need a pointer to raw memory:

- Get data from sensors
- Pass data to a network layer
- Call hand-optimized CUDA or CPU code

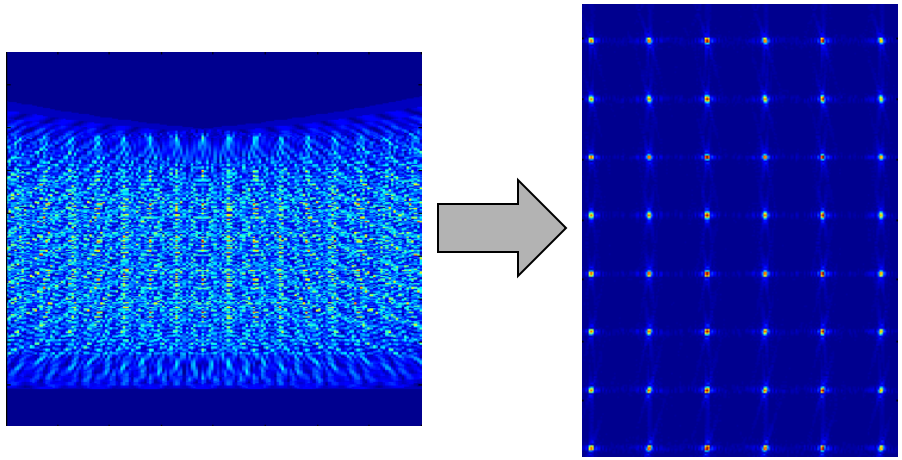
VSIPL++ provides “Direct Data Access”:

- User asks for pointer with specified characteristics
- Gets “zero-copy” pointer to internal block data if possible
- Otherwise, we make a copy for them.

Efficient in most cases, and works for *any* block type.

Portability in Practice:

Synthetic Aperture Radar Benchmark



Synthetic Aperture Radar Benchmark

Standard benchmark from MIT Lincoln Labs

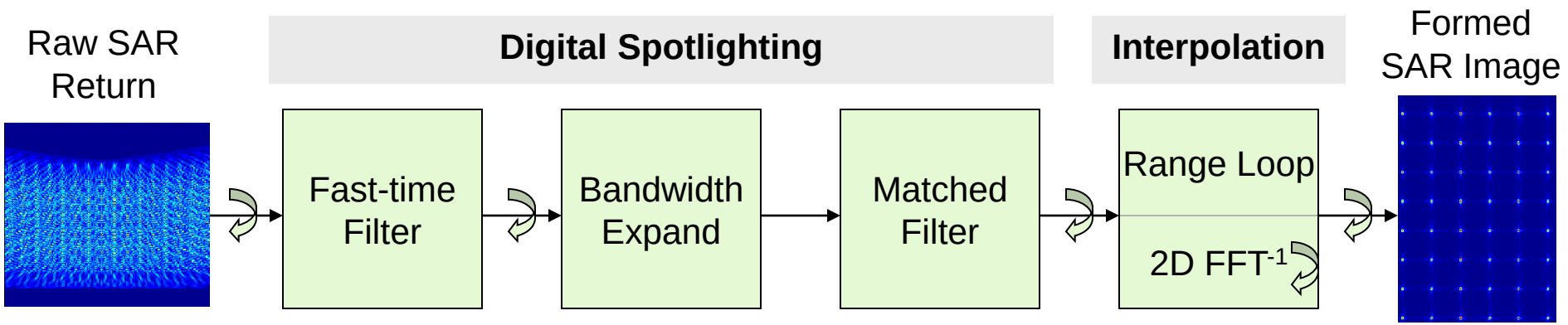
- Part of “HPEC Challenge”, www.ll.mit.edu/HPECchallenge

Simplified but realistic model of real-world code.

Also a good example of porting legacy code!

- Initial VSIPL++ version written in 2006.
- Updated for use in Cell/B.E. presentation in 2008
- Ported to Tesla-architecture GPUs in 2009
- Now we are running it on a Kepler architecture GPU.

Synthetic Aperture Radar Benchmark



Major Computations:	FFT mmul	mmul FFT pad FFT^{-1}	FFT mmul	interpolate 2D FFT^{-1} magnitude

Features:

- Realistic algorithms
- Scalable to arbitrarily large data sizes
  - (we used 2286*1492)
- Matlab & C reference implementations

Challenges:

- 5 corner-turns (transposes)
- Non-power of two data sizes with large prime factors
- Polar to Rectangular interpolation

Implementation Characteristics

Most portions use standard VSIPL++ functions

- Fast Fourier transform (FFT)
- Vector-matrix multiplication (vmmul)

Range-loop interpolation implemented in user code

- Simple C++ by-element implementation (portable)
- Cell/B.E. coprocessor implementation (hardware-specific)

Result is a concise, high-level program

- 200 lines of code in portable VSIPL++
- +200 additional lines for Cell/B.E. optimizations.

Conclusions from 2008 Presentation

Productivity:

- Optimized VSIPL++ is easier to write than even unoptimized C.
- Portable version runs well on x86 and Cell/B.E.
- Hardware-specific interpolation code greatly improves Cell/B.E. performance with small additional effort.

Performance:

- Orders of magnitude faster than reference C code
- Cell/B.E. was 5.7x faster than circa-2008 Xeon x86

Portability: GPU versus Cell/B.E.

Cell/B.E.:

- 8 single-threaded coprocessor cores
- Cores are completely independent
- 256kb local storage
- Fast transfers from RAM to local storage
- Program in C, C++

Kepler GTX-680 GPU:

- 1536 multithreaded coprocessor cores
- Cores execute in (partial) lock-step
- 2 GB device memory
- Slow device-to-RAM data transfers
- Program in CUDA, OpenCL

Very different concepts; low-level code is not portable

Initial CUDA Results

Initial CUDA results			Baseline x86	
Function	**Time**	**Performance**	**Time**	**Speedup**
Digital Spotlight				
Fast-time filter	2.4 ms	48 GF/s	37 ms	15.4
BW expansion	9.9 ms	42 GF/s	128 ms	12.9
Matched filter	7.3 ms	39 GF/s	109 ms	14.9
Interpolation				
Range loop	165 ms	2 GF/s	165 ms	-
2D IFFT	15 ms	24 GF/s	53 ms	3.5
Data Movement	13 ms		91 ms	7.0
Overall	**212 ms**		**583 ms**	**2.8**

A 2.8x speedup – but we can do better!

Interpolation Improvements

Range Loop takes most of the computation time

- Does not reduce to high-level VSIPL++ calls
- Thus, moving to CUDA provides no improvement.

As we did on Cell/B.E., we write a custom low-level implementation using the coprocessor.

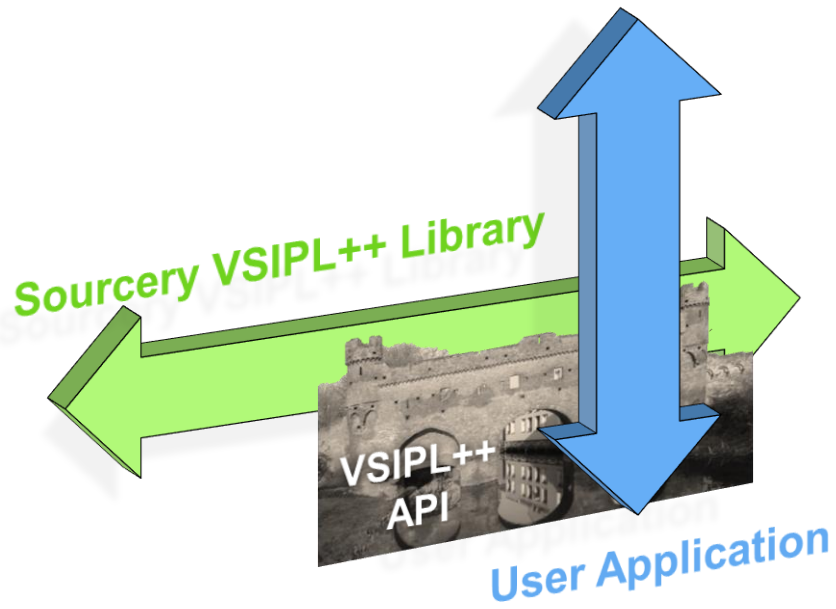
- Sourcery VSIPL++ handles data movement, and provides access to data in GPU device memory.
  - Much simpler than using CUDA directly
- We only need to supply computation code
  - 150 source lines

Improved CUDA Results

Improved CUDA results			Baseline x86	
Function	**Time**	**Performance**	**Time**	**Speedup**
Digital Spotlight				
Fast-time filter	2.4 ms	48 GF/s	37 ms	15.4
BW expansion	9.9 ms	42 GF/s	128 ms	12.9
Matched filter	7.3 ms	39 GF/s	109 ms	14.9
Interpolation				
Range loop	102 ms	3 GF/s	165 ms	1.6
2D IFFT	10 ms	24 GF/s	53 ms	5.3
Data Movement	13 ms		91 ms	7.0
Overall	**144 ms**		**583 ms**	**4.0**

Result with everything on the GPU: a 4x speedup.

Summary and Conclusions



Summary and Conclusions

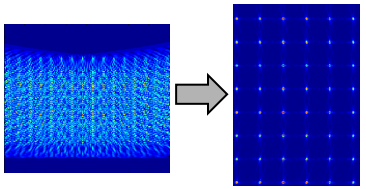
To write portable high-performance code:

- Build a wall between algorithm and implementation!
- Algorithms are portable across different hardware.
- Implementations portable across different algorithms.

VSIPL++ standard API is an effective abstraction layer.

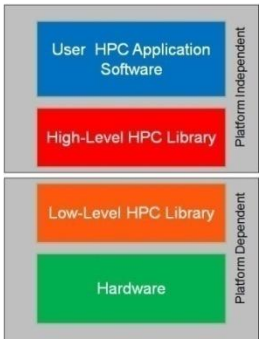
Sourcery VSIPL++ leverages that API to provide both portability and performance.

- Demonstrated results with real programs.
- SAR benchmark showed easy porting effort from Cell/B.E. to Kepler GPU.

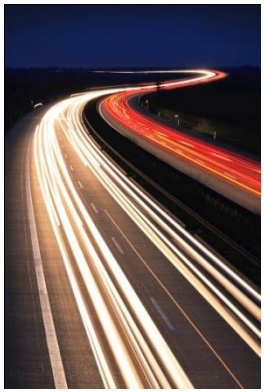


Mentor Embedded HPC Solutions

Thank You!



- We can help with your HPC challenges
  - Sourcery VSIPL++
    - Open Standard API
    - Performance, Portabilty, Productivity
  - Custom HPC Libraries
    - Custom development services
  - Toolchain Support and Services



- http://go.mentor.com/vsiplxx

CodeSourcery

mentor embedded