Spiral
Towards a Universal Program Generator

Yevgen Voronenko
Carnegie Mellon University
SpiralGen Inc.
Traditional Library Development

High performance library optimized for given platform

Spiral: Complete Automation

Spiral

High performance library optimized for given platform

Comparable performance
Scientific Computing

- Unlimited need for performance
- Large set of applications, but ...

- Relatively small set of kernels
  - Matrix multiplication
  - Discrete Fourier transform
  - ...

- Can be provided by *performance libraries*

Consumer Computing

- Can be provided by *performance libraries*

Embedded Computing

- But also unlimited need for *non-functional customization*
Concept

Goal: Replace hand-written function by an auto-generated

Caveats:
- Must be a fully compatible replacement
- Support code will be needed (e.g., verifier)
- Scaling up to 1000s of functions is hard

Let’s look at the issues that must be addressed to achieve this
Industry-Grade Proof of Concept: Intel IPP 6.0

1700+ functions (ippg*)
684,000 lines of code
Written by a computer

AVX, Larrabee code already generated for next IPP release
Issues to Consider

Non-functional Requirements
- Interface
- Code size
- One-/two-call invocation
- Memory allocation scheme
- Unaligned vector support
- Verification
- Naming
- # of entry points (in threaded code)

Functional Requirements:
- Input and output data types
- Internal precision and data type
Interface

Complex FFT

IPPAPI(IppStatus, ippgDFTFwd_CToC_32fc, size, scaling)
(const Ipp32fc *pSrc, Ipp32fc *pDst, int length, int flag)

Walsh-Hadamard Transform

IPPAPI(IppStatus, ippgWHT_32f, log(size), scaling, memory)
(const Ipp32f *pSrc, Ipp32f *pDst, int order, int flag, Ipp8u *pBuf)

IPPAPI(IppStatus, ippgWHTGetSize_32f, memory)
(int order, Ipp32u *pBufferSize)

Interface = key to scaling up
Syntactic + semantic elements
Code Size Fundamental Tradeoff I: Performance vs. Code Size

Discrete Fourier transform performance [Gflop/s]

Transform size

KLOC = kilo lines of code

Good choice for auto-tuning
Code Size Fundamental Tradeoff I: Universality vs. Performance

The available libraries are not always a good match to requirements.
One- / two-call invocation

fft_init(&desc, 1024);
fft_compute(X, Y, desc) vs. fft_compute(X, Y, 1024)

<table>
<thead>
<tr>
<th></th>
<th>Two-call</th>
<th>One-call</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard for</strong></td>
<td>FFTs</td>
<td>BLAS</td>
</tr>
<tr>
<td><strong>Autotuning</strong></td>
<td>init() or offline</td>
<td>offline</td>
</tr>
<tr>
<td><strong>Init() cost</strong></td>
<td>amortized</td>
<td>upfront</td>
</tr>
<tr>
<td><strong>Total cost</strong></td>
<td>init + n*compute</td>
<td>n*compute</td>
</tr>
<tr>
<td><strong>Ease of use</strong></td>
<td>difficult</td>
<td>easy</td>
</tr>
</tbody>
</table>

What if you make a wrong decision?
Two-call vs. one-call invocation

**init-time / compute-time ratio**

- **FFTW 3.2 pregenerated wisdom**
  - 25x overhead

- **4x**

**Performance [GFlop/s]**

- **FFTW 3.2 pregenerated wisdom**
  - 25x

- **4x**

---

Autogenerated one-call library for small 2-power sizes:

http://spiral.ece.cmu.edu/fft_sse/
Memory Allocation Schemes

Possible choices \((N \text{ bytes})\)
- Use stack
- `malloc()` at initialization time
- `free()` at destruction time
- `malloc()`/`free()` at compute time
- User-provided buffer

Threading \((P \text{ threads})\)
- Threads has private stacks
- \(P*N\) bytes must be allocated
- Need thread-safe `malloc()`
- \(P\) buffers or \(P*N\) large buffer

<table>
<thead>
<tr>
<th></th>
<th>Two-call</th>
<th>One-call</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td></td>
<td>X</td>
<td>No</td>
</tr>
<tr>
<td><code>malloc()</code> at init</td>
<td>X</td>
<td></td>
<td>Space</td>
</tr>
<tr>
<td><code>malloc()</code> at compute</td>
<td>X</td>
<td>X</td>
<td>Time</td>
</tr>
<tr>
<td>User-provided buffer</td>
<td>X</td>
<td>X</td>
<td>Hard to use</td>
</tr>
</tbody>
</table>

Used in Intel IPP

Different scheme can be used for read-only memory (eg. twiddles)
Unaligned Memory Support

- SIMD instructions require aligned data

- Cannot always assume aligned
  - Alignment is not always enforceable in the user app
  - No portable alignment pragmas in Fortran
  - Odd-sized FFT will require unaligned loads

- Interface decision: alignment is known at
  - compute() time (Intel IPP)
  - init() time (FFTW)
Verification

- Custom interface requires a **custom verifier**
  - Must be autogenerated
  - Can generated verifier for each **auxiliary function**

Per-function verifier allows to verify all possible paths in an autotuned library
Data Types

- For scalar code: a minor issue
- For vectorized SIMD code: data type = separate ISA
- Uniform data type assumption is false

Example from IPP:

- Integer samples:
  - `ippsFIR_16s_Sfs`
  - `ippsFIR_32s_Sfs`
  - `ippsFIR_32f_16s_Sfs`
  - `ippsFIR64f_16s_Sfs`
  - `ippsFIR64f_32s_Sfs`
  - `ippsFIR32s_16s_Sfs`
  - `ippsFIR32f_16s_Sfs`
  - `ippsFIR64f_16s_Sfs`
  - `ippsFIR64f_32s_Sfs`
  - `ippsFIR64fc_16sc_Sfs`
  - `ippsFIR64fc_16sc_Sfs`
  - `ippsFIR64fc_32sc_Sfs`
  - `ippsFIR32sc_16sc_Sfs`
  - `ippsFIR32fc_16sc_Sfs`
  - `ippsFIR64fc_16sc_Sfs`
  - `ippsFIR64fc_32sc_Sfs`

- Floating point samples:
  - `ippsFIR_32f`
  - `ippsFIR_64f`
  - `ippsFIR_32fc`
  - `ippsFIR_64fc`
  - `ippsFIR64f_32f`
  - `ippsFIR64fc_32fc`
Scaling Up

- Key issues for 1000+ functions:
  - Getting the interface right
  - Proper naming
  - No further human intervention
  - Autogeneration of all supplemental files
  - Vanilla “template”-based approach does not work
What’s next

- Spiral team recently founded a startup

- Looking for interested partners / customers

- Research on new directions and domains continues at CMU