Introduction to GPU VSIPL

Dan Campbell[*], Andrew Kerr[+]

[*]Georgia Tech Research Institute

[+]School of Electrical and Computer Engineering

dan.campbell@gtri.gatech.edu

23 September 2010

Outline

- VSIPL Background
- Using VSIPL
- GPU VSIPL
- Conclusion

VSIPL - Vector Signal Image Processing Library

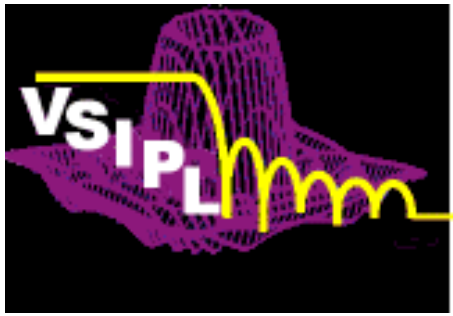
- **Portable API for linear algebra, image and signal processing**
  - **Aimed at Embedded / Desktop / Cluster environments**
- **Goal: Improve productivity, maintain performance**
  - **High level math kernels → fewer lines of code**
  - **No rewrite required to move platforms; rewrite to tune may be needed**
  - **Scalable parallelism under the hood (map file -> automap)**
- **Fairly mature technology:**
  - **Original API Specification approved April 2000, continued growth since**
  - **Supported by DARPA in mid ’90s, Navy, AFRL, HPCMO, ODUSD (S&T) transition support since; consortium effort**
- **Lots of demos, in fairly wide use:**
  - **Aegis, JSF, others**
  - **Historically 90%+ speed of hand-optimized**

VSIPL – Key Features

- **Integrated memory model; direct, first-class coherence controls**
- **Direct support for mathematical objects**
  - **Scalars, vectors, matrices, tensors: first-class, lightweight objects attached to heavyweight memory model**
  - **Heavyweight state objects: e.g. FFT plan, QRD**
- **Flexible precision and type support: includes floating point, fixed point, integer; complex, boolean, index**
- **Functional coverage driven by SP application needs & architecture opportunities**
  - **Basic math: operators, trig, clamps, exponents, max, *etc***
  - **Linear algebra: operators, scatter/gather, system solvers (LLS, QRD, Covariance, Toeplitz, LUD, Cholesky, general, SVD)**
  - **Signal Processing: FFT, Convolution/Correlation, Windowing, FIR/IIR Filters, histograms, RNG**
  - **Defined subsets: Core & Core Lite**

VSIPL – Further Information

VSIPL Website: http://www.vsipl.org



Vector Signal Image Processing Library

The Open Industry Standard For Signal Processing

- **Full API Specification Documents**
- **Reference Implementations**
- **VSIPL Implementation Validation Test Suite**
- **Profile Definitions**
- **Links to implementations**

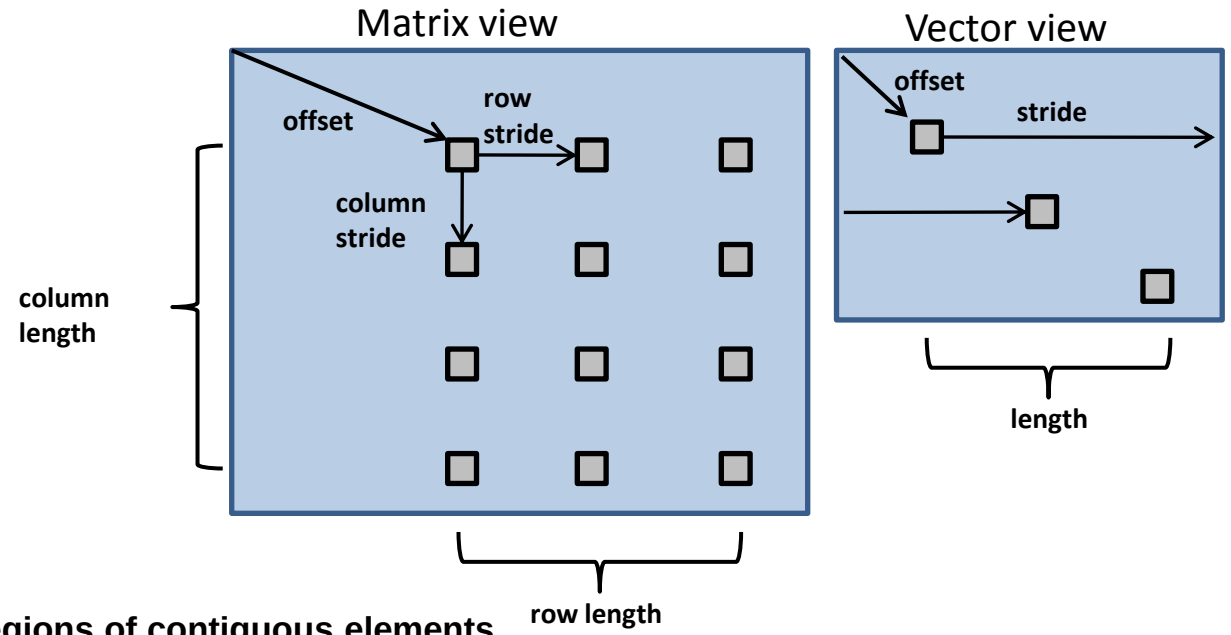
Outline

- VSIPL Background
- Using VSIPL
- GPU VSIPL
- Conclusion

VSIPL Memory Model

- **Blocks are the primary memory abstraction**
  - Opaque representation of a dense array
  - Portable, complete encapsulation of memory management
  - All blocks have a type: int, float, boolean, *etc*; real or complex
  - Optionally associated with app-supplied pointer
  - Ownership state set via admit() & release(); controls coherence
- **Views are the primary mathematical abstraction**
  - Opaque representation of math objects such as vectors, matrices
  - All views have an underlying block (and thus fixed type)
  - Mapped to block via offset, per-dimension stride and length
    - May be remapped easily; mapping is lightweight
  - All math functions operate on views; corresponding block must be in admitted state

VSIPL Blocks and Views



- **Blocks**
  - Linear regions of contiguous elements
    - int, float, complex – translated to interleaved format on block admit
  - Owned by VSIPL – opaque to application
  - Explicit admit and release operations implement VSIPL consistency model
- **Views**
  - fundamental mathematical objects
  - map elements of block into object
  - offset, row stride, column stride, row length, column length
  - subviews take slices of existing views
  - enable flexible data layout

VSIPL Memory Model – Simple Example

```
vsip_scalar_f *d4 = malloc(sizeof(vsip_scalar_f)*LENGTH);
vsip_block_f *b4 = vsip_blockbind_f (d4, LENGTH, VSIP_MEM_NONE);
vsip_vview_f *v4 = vsip_vbind_f (b4, 0, 1, LENGTH);

vsip_vfill_f (1.0f, v4);                              /* This is an error! */

vsip_blockadmit_f (b4, VSIP_FALSE);                     /* No copy here */
vsip_vfill_f (1.0f, v4);
vsip_blockrelease_f (b4, VSIP_TRUE);     /* Copies data to malloc'd array */

for (i=0; i<LENGTH; i++) printf ("%f ", d4[i]);

for (i=0; i<LENGTH; i++) d4[i] = 0.4f;
vsip_blockadmit_f (b4, VSIP_TRUE);                /* Copy from user array */
for (i=0; i<LENGTH; i++) printf ("%f ", vsip_vget_f(v4, i));
```

VSIPL Memory Model – Additional Elements

- **Some operations leverage heavyweight state objects**
  - **FFT**
  - **Random Number Generator**
  - **FIR/IIR**
  - **Convolution/Correlation**
  - **LU/QR/Cholesky/SV Decompositions**
- **Import / Export functionality**
  - **Heavyweight state objects can be exported to memory for retention**
  - **Exported objects not portable**

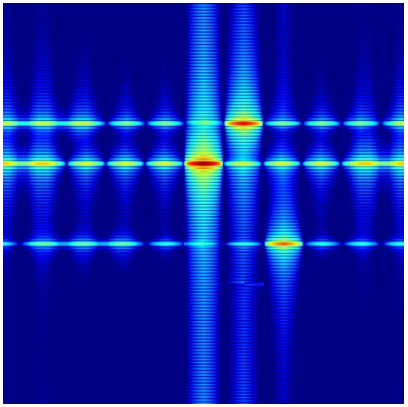
VSIPL Toy Example

```
#define LENGTH 10
vsip_vview_f *v1 = vsip_vcreate_f (LENGTH, VSIP_MEM_NONE);

vsip_block_f *b2 = vsip_blockcreate_f (LENGTH, VSIP_MEM_NONE);
vsip_vview_f *v2 = vsip_vbind_f (b2, 0, 1, LENGTH);

vsip_block_f *b3 = vsip_blockcreate_f (LENGTH*2+3, VSIP_MEM_NONE);
vsip_vview_f *v3 = vsip_vbind_f (b3, 3, 2, LENGTH);

vsip_vramp_f (0,  0.2f, v1);
vsip_vramp_f (1, -0.1f, v2);
vsip_vfill_f (-44.0f,   v3);
vsip_vadd_f (v1, v2, v3);

for (i=0; i<LENGTH; i++) printf ("%f ", vsip_vget_f(v3, i));
```

Output: 1.0 1.1 1.2 1.3 1.4 1.5……

Application Example: Range Doppler Map

- Simple Range/Doppler data visualization demo
- Intro app for new VSIPL programmer
- Speedup TASP ➔ GPU-VSIPL
- No changes to source code



Section	GTX480 Time (ms)	X5650 (1 core) Time (ms)	Speedup
Admit			
Baseband			
Zeropad			
Fast time FFT			
Multiply			
Fast Time FFT^{-1}			
Slow time FFT, 2x CT			
log10 $\|.\|^2$			
Release			
Total:			

VSIPL Example – RD Map 1

```
/* Admit input blocks for VSIPL processing */
vsip_blockadmit_f(d->data_if_block,VSIP_TRUE);
vsip_cblockadmit_f(d->filter_block,VSIP_TRUE);

/* Admit output & scratchpad blocks for VSIPL processing */
vsip_blockadmit_f(d->rd_map_mag_block,VSIP_FALSE);
vsip_cblockadmit_f(d->data_bb_block,VSIP_FALSE);
vsip_cblockadmit_f(d->data_padded_block,VSIP_FALSE);

/* Multiply IF signals by synthesized carriers */
vsip_rcvmul_f(d->data_if_vview,d->carrier_cvview,d->data_bb0_cvview);

/* Apply low-pass filters */
vsip_cfirflt_f(d->data_lpf_fir,d->data_bb0_cvview,d->data_bb_cvview);

/* Initialize padded data to zeros */
vsip_vfill_f((float)0,d->z_re_vview);
vsip_vfill_f((float)0,d->z_im_vview);
```

VSIPL Example – RD Map 2

```
/* Copy data row-by-row from real input to complex workspace */
for(i = 0; i < p->num_pulses; i++) {
  vsip_cvputoffset_f(d->data_bb_row_cvview,i*p->samples_per_pulse);
  vsip_cvputoffset_f(d->z_short_row_cvview,i*p->num_range_bins);
  vsip_cvcopy_f_f(d->data_bb_row_cvview,d->z_short_row_cvview);
}

/* Compute FFT of each pulse of the data */
vsip_ccfftmip_f(d->fft_plan_fast,d->z_cmview);

/* Multiply rows element-wise */
for(i = 0; i < p->num_doppler_bins; i++) {
  vsip_cvputoffset_f(d->z_long_row_cvview,i*p->num_range_bins);
  vsip_cvjmul_f(d->z_long_row_cvview,d->s_cvview,d->z_long_row_cvview);
}

/* Map Back to Range Domain (in fast-time dimension) */
vsip_ccfftmip_f(d->inv_fft_plan_fast,d->z_cmview);
```

VSIPL Example – RD Map 3

```
/* Compute Slow Time FFT of Matched Filtered Data */

/* Transpose */
vsip_cmtrans_f(d->rd_map_cmview,d->rd_map_trans_cmview);

/* Compute FFT across pulses */
vsip_ccfftmip_f(d->fft_plan_slow,d->rd_map_trans_cmview);

/* Transpose */
vsip_cmtrans_f(d->rd_map_trans_cmview,d->rd_map_cmview);

/* Compute Magnitude of range-Doppler Map */
vsip_vcmagsq_f(d->rd_map_cview,d->rd_map_mag_vview);
vsip_vlog10_f(d->rd_map_mag_vview,d->rd_map_mag_vview);

/* Copy results to host */
vsip_blockrelease_f (d->rd_map_mag_block, VSIP_TRUE);
```

VSIPL Platform Issues

- **VSIPL API compliant software is portable, however…**
- **Accelerator-based VSIPL implementations have different performance considerations than pure CPU**
  - Small operations more costly
  - Release/admit, get/put much more costly
  - Higher dimensionality is more efficient
- **CPU implementations more tolerant of application memory errors**

Outline

- VSIPL Background
- Using VSIPL
- GPU VSIPL
- Conclusion

VSIPL & GPU: Well Matched

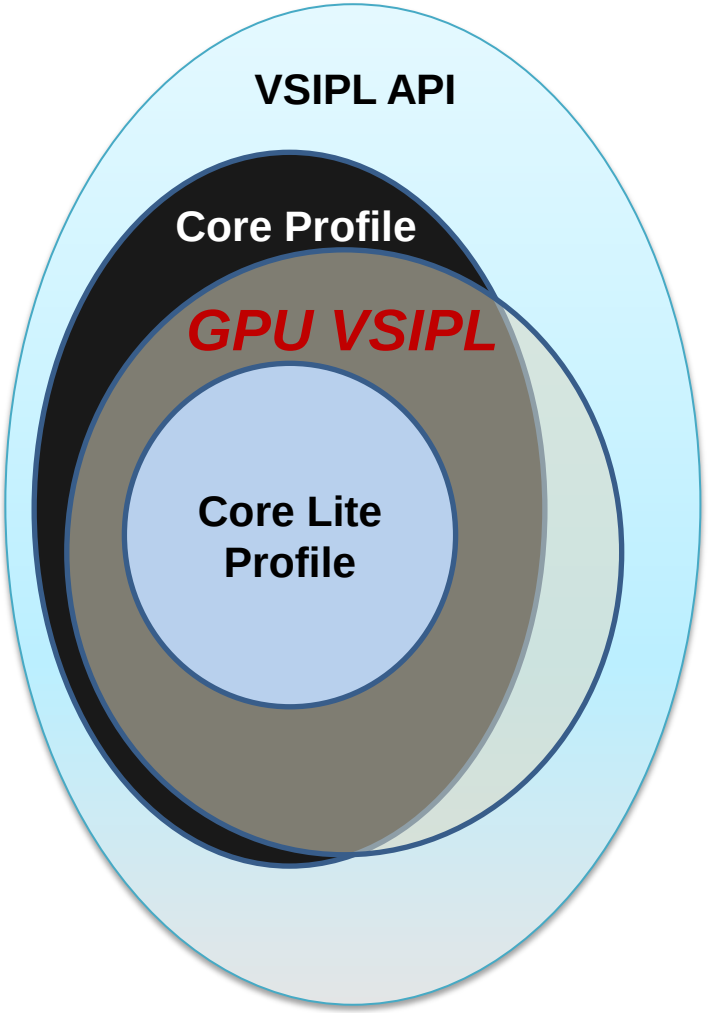
- **VSIPL is great for exploiting GPUs**
  - High level API with good coverage for dense linear algebra
  - Allows non experts to benefit from hero programmers
  - Explicit but abstracted memory access controls
  - API precision flexibility

- **GPUs are great for VSIPL users**
  - Improves prototyping by speeding algorithm testing
  - Cheap addition allows more engineers access to HPC
  - Large speedups without needing explicit parallelism at application level

GPU-VSIPL Implementation

- **GPU VSIPL: Implementation of VSIPL for CUDA GPUs**
- **Fully encapsulated CUDA backend**
  - Leverages CUFFT library
  - All VSIPL functions accelerated
  - No CUDA-specific memory management required
- **Functional Coverage:**
  - Single precision floating point, some basic integer
  - Matrix, Vector & Scalar, complex & real support
  - Elementwise, FFT, FIR, histogram, RNG, support
  - Several linear system solvers
  - All of Core Lite Profile, most of Core Profile, some additional

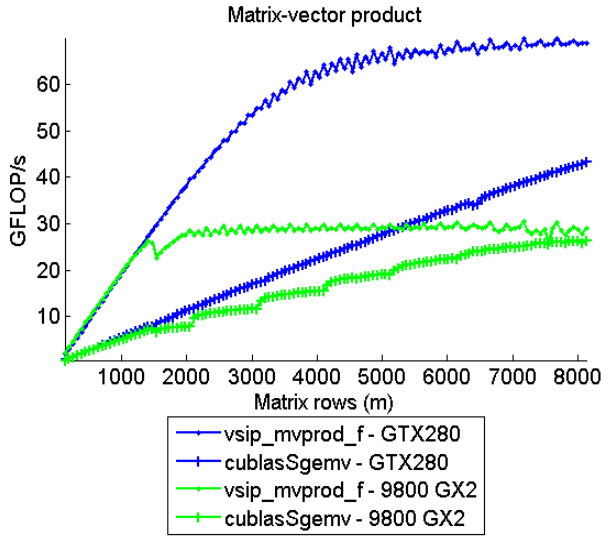
GPU-VSIPL Functional Coverage



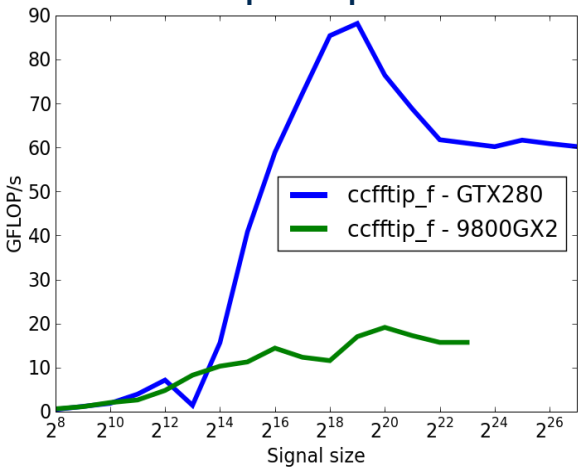
- What's covered from VSIPL Core
  - Data Types
    - real, complex, integer, boolean, index
  - View Types
    - Matrix, vector
  - Element-wise Operators
    - arithmetic, trigonometric, transcendental, scatter/gather, logical, and comparison
  - Signal Processing
    - FFT (in-place, out-of-place, batched)
    - Fast FIR filter, window creation, 1D correlation
    - Random number generation, histogram
  - Linear Algebra
    - generalized matrix product
    - QR decomposition, least-squares solver
- What's Not (yet)
  - Linear Algebra
    - Covariance, Linear Least Squares Solver
- What's Added Beyond VSIPL Core
  - Scalar and matrix versions of element-wise vector operators
  - Matrix utility functions

Some GPU VSIPL Performance

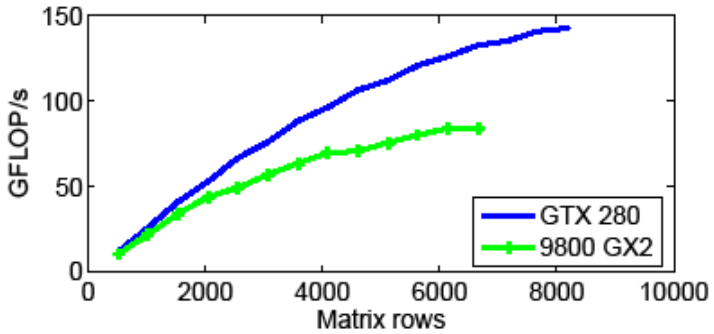
Matrix-vector product



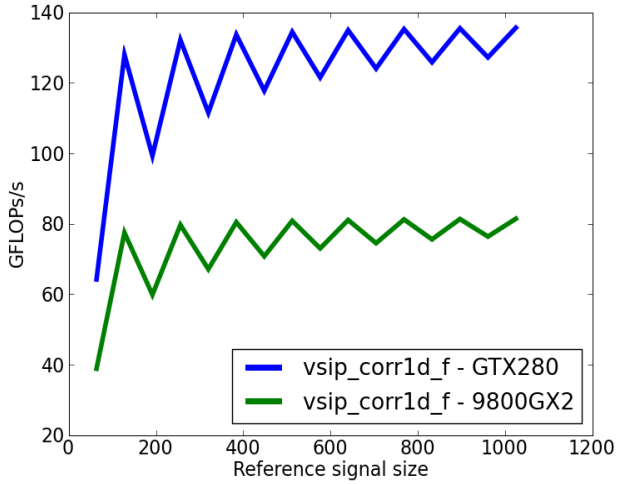
Complex In-place FFT



QR Decomposition

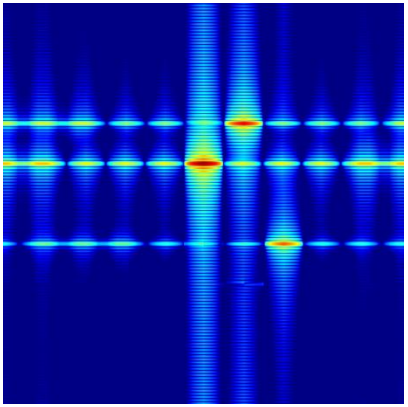


1D Correlation



Application Performance: RD Map

- Simple Range/Doppler data visualization demo
- Intro app for new VSIPL programmer
- 53x Speedup TASP ➔ GPU-VSIPL
- No changes to source code



Section	GTX480 Time (ms)	X5650 (1 core) Time (ms)	Speedup
Admit	0.81	0	0
Baseband	0.75	127.09	169
Zeropad	0.83	10.73	12.9
Fast time FFT	.59	513.89	871
Multiply	14.01	13.13	0.93
Fast Time FFT^{-1}	0.61	513.70	842
Slow time FFT, 2x CT	5.68	503.42	88.6
log10 $\|.\|^2$	0.90	124.12	137.9
Release	9.36	0	0
Total:	33.82	1806.10	53.4

Optimization Techniques: Template Kernels

- Template kernels for specialization
- Example: guard conditionals in inner loops
  - Boolean template parameter may short-circuit conditional expression
  - Optimizer removes control flow entirely for `notConditional=true` specialization

```
template <bool notConditional> __global__ void kernel(int *A, int N) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x;

    if (notConditional || tid < N) {     // conditional expression that may be
      A[tid] = f(A[tid]);                //   short-circuited at compile time
    }
}
…

bool notCconditional = !(N % blockSize.x);   // true if N is multiple of blockSize

kernel< notConditional ><<< gridSize, blockSize >>>(A, N);
```

- `notConditional = true`: no control flow, 13 instructions, 17 registers
- `notConditional = false`: control flow, 16 instructions, 23 registers

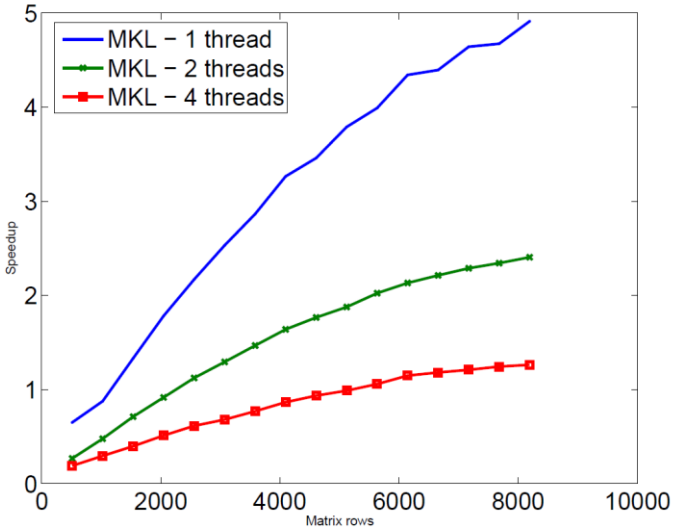
Optimization Techniques: Template Kernels

- **Template kernels and functors for real and complex-valued operators**

```
struct F_mul_f {
    __device__ float operator()(float a, float b) { return a*b; }
};

struct F_cmul_f {
    __device__ float2 operator()(float2 a, float2 b) {
        return make_float2(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x);
    }
};

template<bool notConditional, typename binary_operator_T, typename view_T, typename T>
__global__ void kernel_vsip_vbinary_f(view_T A, const T *a, view_T B, const T *b, view_T R, T *r) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    if (notConditional || tid < R.length) {
        binary_operator_T f;
        r[tid] = f(a[tid], b[tid]);
    }
}

...

// vsip_vmul_f
kernel_vsip_vbinary_f< notConditional, F_mul_f, vsip_vview_f, float ><<< grid, block >>>(
            *A, A->block->device, *B, B->block->device, *R, R->block->device);

// vsip_cvmul_f
kernel_vsip_vbinary_f< notConditional, F_cmul_f, vsip_cvview_f, float2><<< grid, block >>>(
            *A, A->block->device, *B, B->block->device, *R, R->block->device);
```

Optimization Techniques: QR Decomposition

- Matrix decompositions are common operations in signal processing applications
- Standard algorithms:
  - have computationally intensive serialized procedures
  - require fine-grain synchronization and communication across threads
  - are composed of memory-bound primitive operations

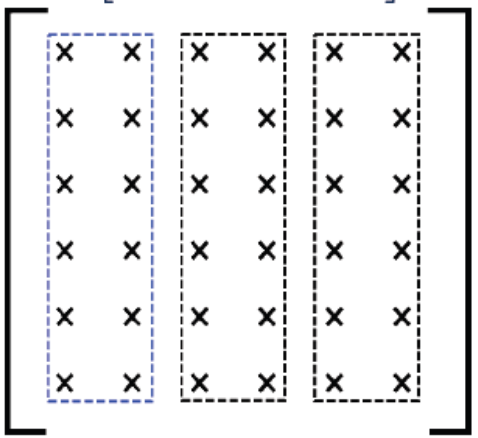


- GPU VSIPL implements Blocked Householder QR decomposition
  - applies *r* reflections in one matrix-matrix product
  - dominant computation is compute-bound
  - coarse-grain serialization of kernels
  - kernel-level parallelism offers performance opportunity on Fermi-class GPUs

Speedup on GTX280 compared to Intel MKL on Intel Xeon at 2.83 GHz with 6MB L2

Optimization Techniques: QR Decomposition

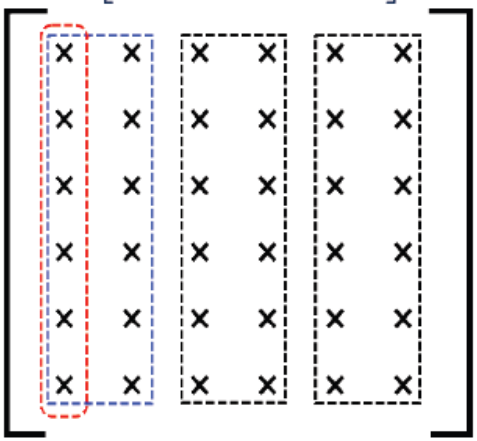
$$A = \begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix}$$



1.) Input matrix is partitioned into blocks $A_1, A_2, \ldots A_p$, each with r columns.

Optimization Techniques: QR Decomposition

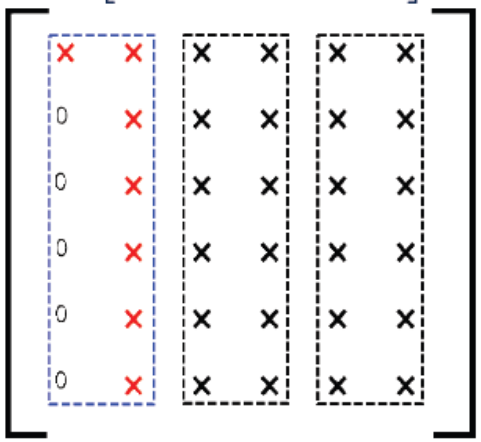
$$A = [A_1 \quad A_2 \quad A_3]$$



2.) A Householder reflection is computed from the first column

Optimization Techniques: QR Decomposition

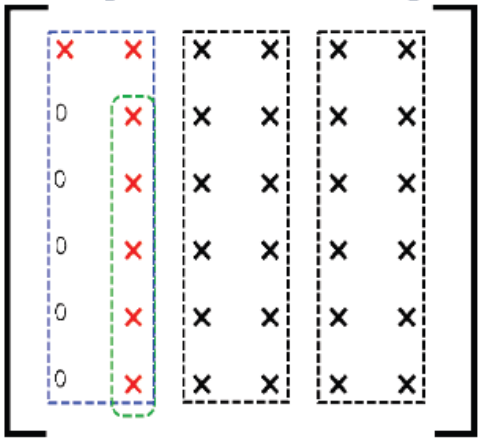
$$A = \begin{bmatrix} P_1 A_1 & A_2 & A_3 \end{bmatrix}$$



3.) and applied to the remaining columns in A_1.

Optimization Techniques: QR Decomposition

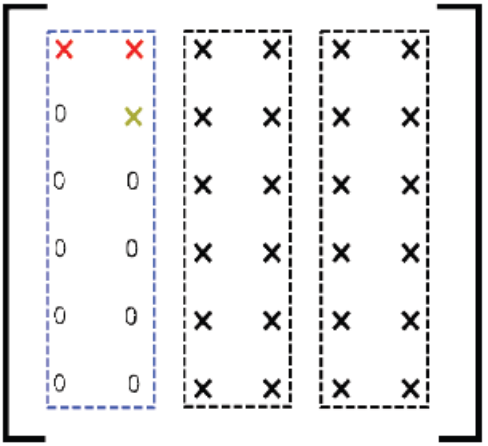
$$A = \begin{bmatrix} P_1 A_1 & A_2 & A_3 \end{bmatrix}$$



4.) A Householder reflection is computed from the second column

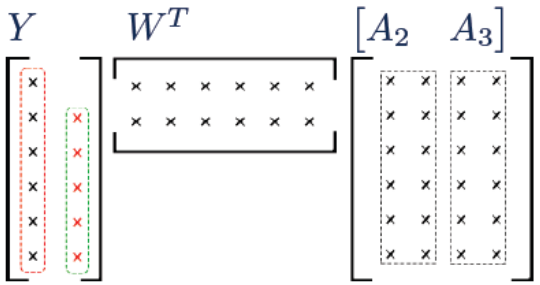
Optimization Techniques: QR Decomposition

$$A = \begin{bmatrix} P_2 P_1 A_1 & A_2 & A_3 \end{bmatrix}$$



5.) and applied to the remaining columns in A_1.

Optimization Techniques: QR Decomposition



6.) After r reflections are applied to block A_1, W is computed from Y.

Then, matrix $[A_2\ A_3\ \cdots A_p]$ and Q are updated according to

$$A_{[2\cdots p]} \leftarrow A_{[2\cdots p]} + YW^T A_{[2\cdots p]}$$

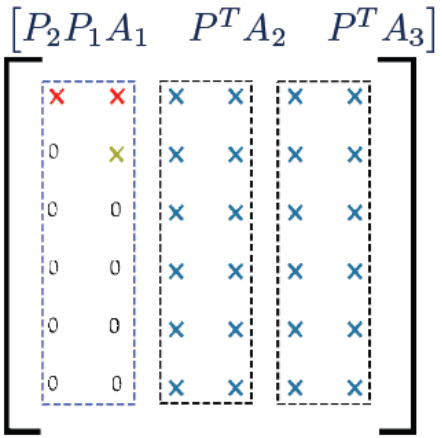
$$Q \leftarrow Q + QWY^T$$

Algorithm 2 Computation of W and Y from V and B [1]

1: $Y = V(1 : end, 1)$
2: $W = -B(1) \cdot V(1 : end, 1)$
3: **for** $j = 2$ to r **do**
4: $\quad v = V(:, j)$
5: $\quad z = -B(j) \cdot v - B(j) \cdot WY^H\ v$
6: $\quad W = [W\ z]$
7: $\quad Y = [Y\ v]$
8: **end for**

[1] Kerr, Campbell, Richards. "QR Decomposition on GPUs." GPGPU '09.

Optimization Techniques: QR Decomposition



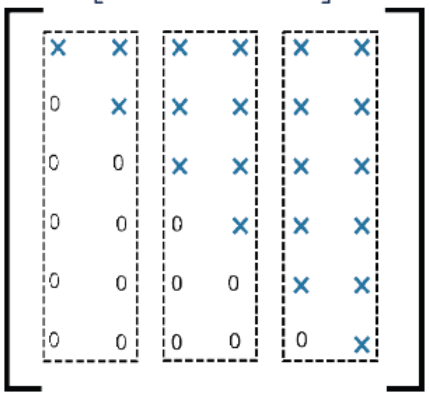
7.) Applying the block Householder update $I + YW^T$ to A is equivalent to performing the first r Householder reflections according to the original algorithm.

Problem sizes for matrix-vector product are much smaller.

Q is updated strictly with matrix-matrix products.

Optimization Techniques: QR Decomposition

$$R = \begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix}$$



8.) Repeat with the next block until all of A is triangularized.

Optimization Techniques: QR Decomposition

Average performance of QP and $P^T A$

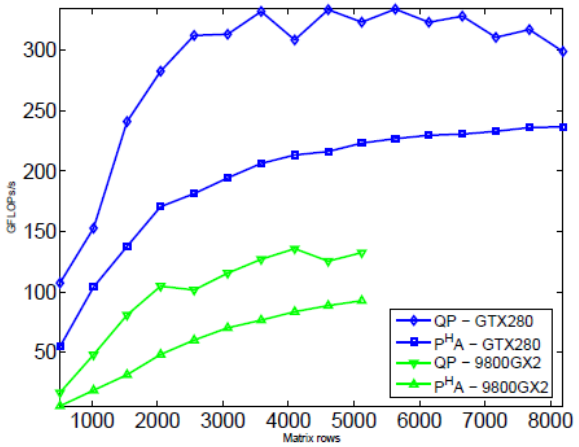


Table: Runtime in seconds for phases of blocked Householder QR on GPUs

Operation	GeForce GTX 280	GeForce GTX 280
Problem size	6656×3328	8192×4096
Householder	0.326	0.565
$A = P \cdot A$	0.952	1.45
WY Computation	1.25	1.86
$A \leftarrow (I + WY^T)^T A$	0.534	0.971
$Q \leftarrow Q(I + WY^T)$	1.36	2.79
Total (seconds)	4.43	7.629
GFLOP/s	129 GFLOP/s	143 GFLOP/s

- **Dominant computations in Blocked Householder Reflections are compute-bound**
  - **WY computation consists of matrix-matrix products with up to 32 columns**
  - **A and Q updates are matrix products with dimensions that are multiples of 32 with no fringes**
  - **A and Q updates may be performed in parallel with memory-bound operations**

Outline

- VSIPL Background
- Using VSIPL
- GPU VSIPL
- Conclusion

GPU VSIPL Summary

- **GPU VSIPL: CUDA performance without CUDA optimization**
- **Information on GPU VSIPL website: http://gpu-vsipl.gtri.gatech.edu**
- **Available for free download**
  - Windows 32/64; Linux 32/64; OSX
  - Unsupported, no redistribution permitted
  - Other licenses available
- **Commercially supported version available from:**
  Runtime Computing Solutions
  http://www.runtimecomputing.com/
  GE Intelligent Platforms http://www.ge-ip.com/

gpu-vsipl.gtri.gatech.edu