More with Less: GPUs, Compressed Sensing, and Sensors

WESLEY FALER
SOFTWARE DEVELOPMENT TEAM LEAD
PART-TIME SCIENTISTS

WF@PTSCIENTISTS.COM
Motivation
Motivation
Motivation
Motivation
Outline

• The data collection pipeline and bottlenecks
• Sampling and reconstruction
• Sparseness
• Compressed Sensing
• Matching Pursuit and AMP algorithms with CUDA
• Expanding data
• Limits and planning
• Conclusions and Q&A
Data collection pipeline

World → Sensor → Filter → uProc

RAM

Comm

Store

Power

Typical: 2k, 32K
Data collection pipeline

World → Sensor → Filter → uProc → Store

RAM

Comm

Power

SRAM: 10 ns
Flash: 10-1,000 us

Flash: 10-1,000 us
Data collection pipeline

- World
- Sensor
- Filter
- uProc
- RAM
- Comm
- Store

- 1200 baud commands
- HD video

Power
Wish list

• Better interpolation
  • Temperatures
  • Solar cell voltages
• Increase frequency response
  • Plasma sensors
  • Motion sensors
• Image reconstruction
A sample sampling problem

- Temperatures, 1/second, 60 total readings
A sample sampling problem

- Linear fit, MSE = 0.059
60:6 Compression

- Polynomial fit, MSE = 0.194
- (Linear fit, MSE = 0.059)
60:20 Compression

- Polynomial fit, MSE = 0.056
- (Linear fit, MSE = 0.059)
Sparseness

- Zero
- Zero
- Zero
- Zero
- Zero
- Not Zero
- Zero
Sparseness with DCT matrix

$$\sqrt{\frac{2}{T}} \cos\left[\frac{\pi}{T} (m - 0.5)(n - 1)\right]$$
Linear Mixing

$$Ax = b$$

\[
\begin{pmatrix}
    a & a & a & a & a & a \\
    a & a & a & a & a & a \\
    a & a & a & a & a & a \\
    a & a & a & a & a & a \\
\end{pmatrix}
\begin{pmatrix}
    x \\
    x \\
    x \\
\end{pmatrix}
= 
\begin{pmatrix}
    b \\
    b \\
    b \\
\end{pmatrix}
\]
Compression with Linear Mixing

\[ Ax = b \]

\[
\begin{pmatrix}
    a & a & a & a & a & a & a \\
    a & a & a & a & a & a & a \\
    a & a & a & a & a & a & a
\end{pmatrix}
\begin{pmatrix}
    x \\
    x \\
    x \\
\end{pmatrix}
= 
\begin{pmatrix}
    b \\
    b \\
\end{pmatrix}
Least Squares Sparse Recovery

Minimize \( \text{Norm}[\text{dct. } x, 2] \)

\[ s.t. \ A. x = b \]
Least Squares Sparse Recovery

Minimize $\text{Norm}[\text{dct. } x, 2]

s.t. A. x == b

<table>
<thead>
<tr>
<th>MSE</th>
<th>60:20</th>
<th>60:6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>0.059</td>
<td>0.059</td>
</tr>
<tr>
<td>Polynomial</td>
<td>0.056</td>
<td>0.194</td>
</tr>
<tr>
<td>L2/DCT</td>
<td>0.043</td>
<td>0.053</td>
</tr>
</tbody>
</table>
**L2 vs. L1 vs. L0**

- **L2**: Sqrt of sum of squares
- **L1**: Sum of absolute values
- **L0**: Number of non-zero items

<table>
<thead>
<tr>
<th>Examples</th>
<th>L2</th>
<th>L1</th>
<th>L0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 3, 5, 7</td>
<td>9.3</td>
<td>17</td>
<td>4</td>
</tr>
<tr>
<td>0, 0, 5, 7</td>
<td>8.6</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>0, 0, 0, 9</td>
<td>9</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
True Sparse Recovery with $\text{Lo}$

Minimize $\text{Norm}[\text{dct. } x, 0]$

$s.t. \ A. x \ =\ = \ b$
Likely True Sparse Recovery with $L_1$

Minimize $\|\text{dct. } x, 1\|$

$s.t. \ A \cdot x == b$

- “A” must be specially formed
- Level of sparseness is critical
- Better at high dimensions
- Special solvers needed – actively researched
Likely True Sparse Recovery with L₁

Minimize \( \text{Norm}[\text{dct. } x, 1] \)

\[ s.t. \ A. x = b \]

<table>
<thead>
<tr>
<th>MSE</th>
<th>60:20</th>
<th>60:6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>0.059</td>
<td>0.059</td>
</tr>
<tr>
<td>Polynomial</td>
<td>0.056</td>
<td>0.194</td>
</tr>
<tr>
<td>L₂/DCT</td>
<td>0.043</td>
<td>0.053</td>
</tr>
<tr>
<td>L₁/DCT</td>
<td>0.030</td>
<td>0.048</td>
</tr>
</tbody>
</table>
Compressed Sensing

- Recovery of non-sparse data from a compressed, sparse, intermediate, data representation.

- Cycle:
  - Create sparse data.
  - Compress sparse data.
  - Recover sparse data.
  - Convert sparse data to full data.
An aside on Wavelets

• Wavelets make sparse data.
• Sparse data can be compressed.
• Wavelets alone are not compression.
• Compressed Sensing can use wavelets, DCT, FFT, and a million other sparsifiers.
• while(true)
  • Wavelets alone are not compression.
Better Interpolation summary

Minimize $\text{Norm}[\text{dct. } x, 1]$

$s.t. \ A \cdot x = b$

- Remote system
  - Applies mixing matrix
    - Bit, Int, Float
  - Comm packet unchanged

- Ground system
  - Applies sparsifier – floating point
Combining Steps

\[ A \cdot x = b \]
Combining Steps

\[ A.x = b \]

\[ x = D.x_s \]

\[ A.(D.x_s) = b \]

\[ (A.D).x_s = b \]
Combining Steps

\[ A.x = b \]

\[ x = D.x_s \]

\[ A.(D.x_s) = b \]

\[ (A.D).x_s = b \]

Minimize \( \text{Norm}[x_s, 1] \) s.t. \( (A.D).x_s = b \)

\[ x = D.x_s \]
Matching Pursuit algorithm
**Algorithm** Matching Pursuit

Input: Signal: $f(t)$, dictionary $D$.

Output: List of coefficients: $(a_n, g_{\gamma_n})$.

Initialization:

$$R_1 \leftarrow f(t);$$
$$n \leftarrow 1;$$

Repeat:

- find $g_{\gamma_n} \in D$ with maximum inner product $|\langle R_n, g_{\gamma_n} \rangle|$;
- $a_n \leftarrow \langle R_n, g_{\gamma_n} \rangle$;
- $R_{n+1} \leftarrow R_n - a_n g_{\gamma_n}$;
- $n \leftarrow n + 1$;

Until stop condition (for example: $\|R_n\| < \text{threshold}$)

http://en.wikipedia.org/wiki/Matching_pursuit
/**
 * Find sparsest sparseData given:
 * dictionary * sparseData = compressedSignal.
 * dictionary is dictionaryColumns columns x dictionaryRows rows
 * (M x N in other docs)
 * sparseData is 1 columns x dictionaryColumns rows.
 * This is the result.
 * compressedSignal is 1 columns x dictionaryRows rows.
 * NOTE: Overwritten during calculations.
 *
 * Based upon code from:
 * "Fast GPU Implementation of Sparse Signal Recovery
 * from Random Projections" by M. Andrecut
 */

bool matchingPursuit(int dictionaryColumns, int dictionaryRows,
 const device_vector<double> &dictionary,
 device_vector<double> &compressedSignal,
 host_vector<double> &sparseData);
device_vector<double> residual; // Needs to be dictionaryColumns long,  
    // and seeded with the dictionaryRows  
    // of data from compressedSignal
residual.assign(dictionaryColumns, 0.0);
residual = compressedSignal;
// Maximum number of iterations
const int T = 2 * dictionaryColumns;
// Residual error
double epsilon = 1.0e-7;
int t, q;
double normi, normf, a;
// Decoding the signal on device
host_vector<double> c;
c.assign(1, 0.0);
normi = cublasDnrm2(dictionaryRows,
    raw_pointer_cast(&compressedSignal[0]), 1);
epsilon = sqrt(epsilon * normi);
normf = normi;
t = 0;
while(normf > epsilon && t < T) {
    cublasDgemv('t', dictionaryColumns, dictionaryRows, 1.0,
    raw_pointer_cast(&dictionary[0]), dictionaryColumns,
    raw_pointer_cast(&compressedSignal[0]), 1, 0.0,
    raw_pointer_cast(&residual[0]), 1);

    q = cublasIdamax(dictionaryColumns,
    raw_pointer_cast(&residual[0]), 1) - 1;
    cublasGetVector(1, sizeof(double *),
    raw_pointer_cast(&residual[q]),
    1, raw_pointer_cast(&c[0]), 1);
    sparseData[q] = c[0] + sparseData[q];
    cublasDaxpy(dictionaryRows, -c[0],
    raw_pointer_cast(&dictionary[q*dictionaryRows]),
    1, raw_pointer_cast(&compressedSignal[0]), 1);

    normf = cublasDnrm2(dictionaryRows,
    raw_pointer_cast(&compressedSignal[0]), 1);
    t++;
}
// Index into a CUBLAS array
#define id(row,column,rows) (((column)*(rows)+(row)))

// +1/-1 sign of a number
double sign(double x) { return(x>=0)-(x<0); }

// Obtain an entry from a type 2 DCT matrix.
__host__ __device__
double dctCell(int size, int row, int column) {
    return sqrt(2.0/(double)size) * cos((M_PI/(double)size)*(column+1-0.5)*(row+1-1));
}
Testing your algorithms

- Create a `random` sparse signal ("$X_s$") of -1, 0, +1
- Create a `random` dictionary (i.e. "A.D")
  
  ```
  dictionary[i] = sign(2.0*rand()/(double)RAND_MAX - 1.0) /
  sqrt(compressedSize);
  ```
- Create sample ("b")
- Call the algorithm
- **Check** the results
Matching Pursuit results

- Iterations increase quickly with variable count
- Good results with small problem sizes
- Good results with imperfect sparseness
Lo with Thrust

```cpp
struct nonZeroCount_functor {
    __host__ __device__
    double operator()(const double &x) const {
        return (x != 0? 1 : 0);
    }
};

double l0Norm(const device_vector<double> &x) {
    return thrust::transform_reduce(
        x.begin(), x.end(),
        nonZeroCount_functor(),
        0.0, thrust::plus<double>()) ;
}
```
AMP algorithm
Approximate Message Passing (AMP) algorithm

\[ r^t = y - Ax^t + b_t r^{t-1} \]

\[ x^{t+1} = \eta(x^t + A^T r^t; \theta_t) \]

\[ b_t = \frac{1}{m} \mid x^t \mid_0 \]

\[ \theta_t = \alpha \sqrt{\frac{1}{m} \mid r^t \mid^2} \]

\[ \eta(c; \theta) = \begin{cases} 
  c - \theta & \text{if } c > \theta \\
  0 & \text{if } -\theta \leq c \leq \theta \\
  c + \theta & \text{if } c < -\theta 
\end{cases} \]

\[
\begin{align*}
  t &\geq 0 \\
  r^{-1} &= 0 \\
  x^0 &= 0 \\
  \alpha &= 1.40814
\end{align*}
\]
/**
 * Find sparsest sparseData given
 * dictionary * sparseData = compressedSignal.
 * dictionary is dictionaryColumns columns x dictionaryRows rows
 * sparseData is 1 columns x dictionaryColumns rows.
 * This is the result.
 * compressedSignal is 1 columns x dictionaryRows rows.
 * Overwritten during calculations.
 ***/
bool AMP(int dictionaryColumns, int dictionaryRows,
         const device_vector<double> &dictionary,
         device_vector<double> &compressedSignal,
         host_vector<double> &sparseData);
// Maximum number of iterations
const int T = 2*dictionaryColumns;
// Acceptable error = epsilon * l2 norm of residual
double epsilon = 1.0e-8; // 1.0e-7;
// Empirically 1e-8 takes a few extra iterations vs. 1e-7
// and cuts absolute error by about 3x

device_vector<double> residual; // vector, dictionaryRows long
residual.assign(dictionaryRows,0.0);

double b, theta;
const double alpha = 1.40814; // empirically determined in literature

device_vector<double> x; // vector, dictionaryColumns long
x.assign(dictionaryColumns,0.0);

device_vector<double> c; // vector, dictionaryColumns long
c.assign(dictionaryColumns,0.0);

// Decoding the signal on device
while(normf > epsilon && t < T) {
    // Update residual
    if(0 == t) { // The first residual is a special case
        // residual = -Dictionary * x + compressedSignal
        // Since x == 0 in the first iteration, residual = compressedSignal
        residual = compressedSignal;
    } else {
        // b = (1/rows) * l0 norm of X
        b = l0Norm(x) / (double)dictionaryRows;
        // residual = b*residual + compressedSignal
        thrust::transform(residual.begin(), residual.end(),
                          compressedSignal.begin(), residual.begin(), saxpy_functor(b));
        // residual = -Dictionary * x + residual
        cublasDgemv('n', dictionaryRows, dictionaryColumns, -1.0,
                    raw_pointer_cast(&dictionary[0]), dictionaryRows,
                    raw_pointer_cast(&x[0]), 1, 1.0, raw_pointer_cast(&residual[0]), 1);
    }
    // Update X
    ...
}
// Update X
  c = x;
// c = (A^T) * residual + c
cublasDgemv('t', dictionaryRows, dictionaryColumns, 1.0,
            raw_pointer_cast(&dictionary[0]), dictionaryRows,
            raw_pointer_cast(&residual[0]), 1, 1.0, raw_pointer_cast(&c[0]), 1);
// normf = l2 norm of residual
normf = cublasDnrm2(dictionaryRows, raw_pointer_cast(&residual[0]), 1);
// theta
theta = alpha * sqrt(normf*normf/dictionaryRows);
// x = clipped(c,theta)
thrust::transform(c.begin(),c.end(),x.begin(),amp_clipping_functor(theta));
t++;

// Move the result back
sparseData = x;
struct amp_clipping_functor {
    double theta;
    double negativeTheta;

    amp_clipping_functor(double thetaToUse)
        : theta(thetaToUse), negativeTheta(-thetaToUse) { }

    __host__ __device__ double operator()(const double &x) const {
        if(x > theta) {
            return x - theta;
        } else if(x < negativeTheta) {
            return x + theta;
        } else {
            return 0;
        }
    }
};
AMP results

- Iterations increase slowly with variable count
- Poor results with small problem sizes
- Poor results with imperfect sparseness
- Better fit than Matching Pursuit
Expanding data

• Given uncompressed, uniformly sampled data
• Create additional data points between samples
  • Training algorithms
  • Derivative estimates
Expanding data

\[(A \cdot D_E) \cdot E_s = b\]
\[E = D_E \cdot E_s\]

- “b” holds the collected readings
- “Es” holds the sparse expanded readings
- “E” holds the expanded readings
- “DE” is a DCT matrix sized for the expanded readings
- “A” compresses the expanded readings
A: model sensor reading overlap

\[
\begin{pmatrix}
0.675 & 0.25 & 0.075 & 0 & 0 & 0 \\
0 & 0 & 0.675 & 0.25 & 0.075 & 0 \\
0 & 0 & 0 & 0 & 0.75 & 0.25 \\
\end{pmatrix}
D_E
E_s = \begin{pmatrix}
b \\ b \\ b \\
\end{pmatrix}
\]

- 2x expansion
- 3-point mixing model (empirical)
Expansion summary

\((A \cdot D_E).E_S = b\)

\[ E = D_E \cdot E_S \]

- Remote system
  - No changes to collection
  - No mixing or sparsification
  - Comm packet unchanged

- Ground system
  - Applies estimate of sample interdependence
Limit: Dictionary randomness

- Restricted Isometry Condition
  - Guassian $\pm \frac{1}{\sqrt{N}}$
  - N random rows from Fourier matrix
- No zero columns
- Overlaps
Limit: Recovery phase transition

![Graph showing the transition between safe and unsafe regions based on sparseness/transmit ratio and transmit/read ratio. The graph includes two lines: one for Pos & Neg and another for Positive only. The graph delineates the boundary between the safe and unsafe regions.]
Limit: Recovery phase transition

- 1000 readings
- 250 transmitted
- 125 non-zeros
Limit: Recovery phase transition

- 1000 readings
- 250 transmitted
- 25 non-zeros
Limit: Recovery phase transition

- 60 readings
- 20 transmitted
- 3 non-zeros
Preconditioning drives sparsity

Raw data

DCT spectrum
Preconditioning drives sparsity

60:20 Reconstruction, MSE=0.041

60:6 Reconstruction, MSE=7.507
Preconditioning drives sparsity

Data – Linear Fit

DCT spectrum

\[ y[i] = \text{data}[i] - (49.992 + 0.081x) \]
Preconditioning drives sparsity

60:20 Reconstruction, MSE=0.030

60:6 Reconstruction, MSE=0.048
Solver tips

• Isolate algorithms
• Minimize clutter - see the algorithm
• Avoid generic N, M, K variable names
• Comments with function summary - not obvious from CUBLAS names
• Unit tests - both fixed and random
• Integrate visualization
• Feed the sparsifier
Future: Image reconstruction

- Restrictions with small microprocessors
- Using a pixel stream
- Using dictionary generators
  - Large memory requirements otherwise
- Including scene templates
  - Fewer variables vs. Increased sparsity
Conclusions

• Thrust and CUBLAS used
• Sparse solvers need only sparse amount of code
• No custom kernels needed
• Code and performance are not barriers
• Code to be published open-source
Thank you!

Q&A?

WESLEY FALER
SOFTWARE DEVELOPMENT TEAM LEAD
PART-TIME SCIENTISTS

WF@PTSCIENTISTS.COM