OpenCL implementations of principal component analysis for large scale data analysis and benchmarking of heterogeneous clusters.

Josh Bowden, CSIRO IM&T Advanced Scientific Computing

Large imaging data sets

- Synchrotron
  - Synchrotron i.e. X-ray tomography, XRF and Maia detector.
- Medical Imaging
- Phenomics instrumentation
  - The Plant Accelerator: Crop growth studies
  - SilviScan: Wood property analysis
- Remote sensing
  - Satellite data
  - Sensor networks
  - Astronomy
SilviScan 3 Rapid wood properties analysis system

- Diffractometer provides microfibril angle, fiber angular information, crystalinity and cellulose crystal width (work in progress).

- Data requirements
  - High spatial resolution imaging at 100 μm steps (200 μm spot size)
  - ~2000-8000 images per day - up to 16 GB uncompressed
  - 4 - 40 samples per day

Analysis of data

- Statistical measures for interrogating large data sets
  - Principal component analysis PCA:
    - Exploratory data analysis
    - Useful in machine learning, outlier detection, regression and prediction procedures

\[ M_{m,n} \xrightarrow{PCA} T_{m,c} \times P_{c,n} + R_{m,n} \]

- NIPALS: An iterative method for finding principle components
  - Developed in 1960’s by H. Wold
Multivariate analysis of images

- Based on MACI\textsuperscript{2} technique using ‘vectorised’ diffraction Phi-Theta space images

\[ \text{Eigenvectors (p)} \times \text{Scores (t)} \]

- PCA (or PLS)

\[ \text{Number of PCs wanted} \]

\[ \text{‘Vectorised’ images (M)} \]


NIPALS - an iterative algorithm

- NIPALS PCA requires multiple iterations of vector-matrix operations before convergence – large memory and computational time requirements.
- Can this be addressed by use of GPUs?
- Andrecut (2008) suggests it can, at least for smaller problems that can fit in GPU RAM.
- Used cuBLAS routines and obtained a 12x speed up over single core CPU implementation.

- N.B. NIPALS can also introduce lack of orthogonality of resulting eigenvectors due to machine precision influence. Not addressed in this work – see Andrecut (2008) for details.
NIPALS – core algorithm

while (|diff| ≥ ϵ)
{| 
  \( p = t' \cdot M \)
  \( \hat{p} = p|t' \hat{p} | \)
  \( t = M \cdot \hat{p} \)
  \( \lambda_{\text{new}} = \| t \| \)
  \( \text{diff} = \lambda_{\text{orig}} - \lambda_{\text{new}} \)
  \( \lambda_{\text{orig}} = \lambda_{\text{new}} \)
|} 

NIPALS - a bandwidth limited algorithm

Each vector matrix operation results in one multiply and add per two memory reads and one memory write.

BLAS general vector - matrix routines provide efficient CPU algorithms.

Example in C:

```c
void CalculateScores (const T* M, const T* EVect, uint numRows, uint numCols, T* Scores)
{
    // for each row in M
    for (size_t x1 = 0; x1 < numRows; x1++) {
        for (size_t x2 = 0; x2 < numCols; x2++) {
            Scores[x1] += M[x2 + x1*numCols] * EVect[x2];
        }
    }
}
```
CSIRO GPU cluster

- 128 compute nodes:
  - 2 x 4 core E5462 CPU
  - 2 x Nvidia S1070 “Tesla T10” / 4 GB
  - Replaced with
    - 2 x Nvidia S2050 “Fermi” / 2.6 GB
  - 665 GB ECC VRAM
  - 4 TB DDR2 800MHz system RAM
- Tested Bandwidth:
  - GPU @ 80 GB/s
  - CPU @ 5 GB/s (non-cache resident problem)

Provides both speed via GPU’s parallel core processing and a large memory space through distributed memory and MPI.

OpenCL* - Open Computing Language

- Cross platform C API for programming of GPUs, CPUs and other accelerators.
- OpenCL library available for multiple vendors hardware – AMD and NVidia GPUs, IBM, Intel and AMD CPUs
- Has bindings for multiple languages/environments
  - C++, C#, Java, Python, Pascal, Mathematica
OpenCL parallel programming model

- Data parallel and task parallel.
- Data parallel model forces discretisation of a problem into independent blocks. Useful for mapping to multiple processing units of the GPU that do not allow efficient communication.
- Task parallel using different command queues.

OpenCL kernel functions

- Based on C99 + bunch of math.h functions as inbuilt.
- Borrows a lot from CUDA
  - get_local_id()
  - get_group_id()
  - get_global_id()
- Allow use of independent memory spaces—
  - High speed __local memory as a programmer controlled cache.
  - High speed __constant read only memory
  - High speed texture cache through 2D and 3D image objects.
- Has ‘vector’ based data types consisting of 2 to 16 elements
- Can be compiled during execution or pre-compiled.
  - JIT type compilation allows it to take advantage of heterogeneous systems – eg. MPI instances running on different accelerators within same server or across nodes.
Code optimisation in OpenCL

Array access for $p = t^\top M$:

a. $t$ \[ p, p, p, \ldots \] $M$

b. $t$ \[ p, p, p, p, p, p, p, p, \ldots \] $M$

Array column alignment:

Use of multiple GPU resources:

sum-reduce $p$ followed by broadcast of result
Results for single PCA iteration

- OpenCL (CPU and GPU), cuBLAS, MKL BLAS, ACML BLAS
- Single cluster node vs workstation
- 6000 image / $256^2$ pixel per image data set of double precision, 2.9GB total.

![Graph showing time per iteration vs number of CPU cores or GPUs]

Results on GPU cluster using MPI

![Graph showing time per iteration vs number of GPUs]
Modelling the NIPALS algorithm – per function timing

Time per iteration models:

Tesla 'time per iteration' model:

\[
t(i, p, G) = 9.499 \cdot 10^{-11} \cdot \frac{i \cdot p^2}{G} + 9.9823 \cdot 10^{-8} \cdot p^2 +
\]

\[
- 1.1784 \cdot 10^{-3} \cdot G + 0.0268 \cdot \frac{i \cdot p^2}{G} + 5.0 \cdot 10^{-5} \cdot G + 0.0013527
\]

\[
t = \text{time}
\]
\[
i = \text{number of images}
\]
\[
p = \text{number of pixels}
\]
\[
G = \text{number of GPUs}
\]
eXtreme-scale performance?

- GPU RAM is the limitation.
  - Weakest link is PCIe bus – if data too big for VRAM then procedure slows to CPU speeds and effort is wasted.
  - Possibly faster on Fermi due to two way asynchronous transfers
- Petabyte of image data:
  - 134 x10^6 images (at 1024 x 1024 pixels per image, 8 Bpp)
  - would require 260,000+ Tesla based GPUs
  - Tesla model predicts 10.3 seconds per iteration!
  - Distinct possibility that MPI timing model is not reliable at this size problem.
  - Would take SilviScan system 21 years running 24/7 to collect this much data
  - Only ~4 days of 360 fps camera

Example prediction results – model validation set

Regression coefficients
(blue +ve, green −ve coef.)

MFA:
PLS, 4 PCs
SEP = 1.4°
Prediction of species:

- Regression models can differentiate species of wood from diffraction pattern:

![Regression coefficients (25 PCs)](image)

Edge of sample effects

Contact Us
Phone: 1300 363 400 or +61 3 9545 2176
Email: enquiries@csiro.au  Web: www.csiro.au

Thank you
CPU vs GPU code for $t=Mp$

**CPU code:**

```c
void ScoresAlongRowsCPU(const T* M, const T* EVect, uint numRows, uint numCols, T* Scores)
{
    // for each row in M
    for (uint y = 0; y < numRows; y++)
    {
        T sum = 0;
        // for each column in M
        for (uint x = 0; x < numCols; x++)
        {
            sum += M[y * numCols + x] * EVect[x];
        }
        Scores[y] = sum;
    }
}
```

**GPU code:**

```c
__kernel void ckKernelMatVectAlongRows2(const __global T* M, const __global T* EVect,
                                         uint numRows, uint numCols, __global T* Scores,
                                         __local T* partialDotProduct)
{
    // Each work-group handles as many matrix rows as necessary
    for (uint y = get_group_id(0); y < numRows; y += get_num_groups(0))
    {
        const __global T* row = M + y * numCols; // Row pointer - points to the begining of the row wanted
        // Each work-item accumulates as many products as necessary
        T sum = 0;
        for (uint x = get_local_id(0); x < numCols; x += get_local_size(0))
        {
            sum += row[x] * EVect[x];
        }
        // Each partial dot product is stored in shared memory
        partialDotProduct[get_local_id(0)] = sum;
        // Perform parallel reduction to add each work-item's partial dot product together
        for (uint stride = 1; stride < get_local_size(0); stride *= 2)
        {
            barrier(CLK_LOCAL_MEM_FENCE);
            uint index = 2 * stride * get_local_id(0);
            if (index < get_local_size(0))
            {
                partialDotProduct[index] += partialDotProduct[index + stride];
            }
        }
        if (get_local_id(0) == 0)
        {
            Scores[y] = partialDotProduct[0];
        }
    }
}
```

**CPU code:**

```c
void ScoresAlongRowsCPU(const T* M, const T* EVect, uint numRows, uint numCols, T* Scores)
{
    // for each row in M
    for (uint y = 0; y < numRows; y++)
    {
        T sum = 0;
        // for each column in M
        for (uint x = 0; x < numCols; x++)
        {
            sum += M[y * numCols + x] * EVect[x];
        }
        Scores[y] = sum;
    }
}
```

**GPU code:**

```c
__kernel void ckKernelMatVectAlongRows2(const __global T* M, const __global T* EVect,
                                         uint numRows, uint numCols, __global T* Scores,
                                         __local T* partialDotProduct)
{
    // Each work-group handles as many matrix rows as necessary
    for (uint y = get_group_id(0); y < numRows; y += get_num_groups(0))
    {
        const __global T* row = M + y * numCols; // Row pointer - points to the begining of the row wanted
        // Each work-item accumulates as many products as necessary
        T sum = 0;
        for (uint x = get_local_id(0); x < numCols; x += get_local_size(0))
        {
            sum += row[x] * EVect[x];
        }
        // Each partial dot product is stored in shared memory
        partialDotProduct[get_local_id(0)] = sum;
        // Perform parallel reduction to add each work-item's partial dot product together
        for (uint stride = 1; stride < get_local_size(0); stride *= 2)
        {
            barrier(CLK_LOCAL_MEM_FENCE);
            uint index = 2 * stride * get_local_id(0);
            if (index < get_local_size(0))
            {
                partialDotProduct[index] += partialDotProduct[index + stride];
            }
        }
        if (get_local_id(0) == 0)
        {
            Scores[y] = partialDotProduct[0];
        }
    }
}
```
CPU vs GPU code for $p=t'M$

**CPU code:**

```c
void EVectsAlongRows(const T* M, const T* Scores, 
                      uint numRows, uint numCols, T* EVect)
{
    // for each row
    for (size_t x1 = 0; x1 < numRows; x1++)
    {
        for (size_t x2 = 0; x2 < numCols; x2++)
        {
            EVect[x2] += M[x1*numCols + x2] * Scores[x2];
        }
    }
}
```

**GPU code:**

```c
__kernel void ckKernelVectMatDownCols1(const __global T* M, __constant T* Scores,
                                       uint numRows, uint numCols, __global T* Evects)
{
    // column index - a thread for each dot product down a column
    uint x1 = get_global_id(0);
    if (x1 < numCols)
    {
        const __global T* col = M + x1;  // offset along first row
        T dotProduct = 0;
        for (int x2 = 0; x2 < numRows; x2++)
        {
            dotProduct += col[x2*numCols] * Scores[x2]; // compute the dot product
        }
        Evects[x1] = dotProduct; // write result to global memory
    }
}
```

Modelling done ‘by hand’ – next time get R to do it.
Multiple linear regression

\[ y = x\beta + \varepsilon \]
\[ y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \ldots + \varepsilon \]
\[ \beta = (X^T X)^{-1} X^T y \]

\[ \lambda_i \]
- images *pix*pixy/GPU
- images *(pix*pixy)^2/GPU
- pix*pixy
- 1/GPUs
- pix*pixy/GPUs

NIPALS - a bandwidth limited algorithm

\[ \text{while } |\text{diff}| \geq \varepsilon \]
\[
\begin{align*}
& p = t^TM \\
& \hat{p} = \hat{p} - \sum \lambda_j \hat{p}_j \\
& t = M - \hat{p} \\
& \lambda_{new} = |t| \\
& \text{diff} = \lambda_{new} - \lambda_{old} \\
& \lambda_{old} = \lambda_{new}
\end{align*}
\]

void ScoresAlongRows(const T* M, const T* EVect, uint numRows, uint numCols, T* Scores) {
  // for each row
  for (size_t x1 = 0; x1 < numRows; x1++) {
    for (size_t x2 = 0; x2 < numCols; x2++) {
      Scores[x1] += M[x1*numCols + x2] * EVect[x2];
    }
  }
}

void EVectsAlongRows(const T* M, const T* Scores, uint numRows, uint numCols, T* EVect) {
  // for each row
  for (size_t x1 = 0; x1 < numRows; x1++) {
    for (size_t x2 = 0; x2 < numCols; x2++) {
      EVect[x2] += M[x1*numCols + x2] * Scores[x2];
    }
  }
}

BLAS general vector - matrix routines provide efficient CPU algorithms.