Automatically converting C/C++ to OpenCL/CUDA

Introduction by David Williams
This presentation provides an introduction to autoparallelisation, focusing on our GPSME toolkit.

We will cover:

- What autoparallelisation is and why we want it.
- How the autoparallelisation process is performed.
- An introduction to using our toolkit.
- Benchmarking the toolkit and performance considerations.
- A demonstration of using the toolkit and frontend.

Toolkit is available.
Who are we?

- The GPSME project is a collaboration between industry and academia.
  - Multiple partners across Europe.
  - All with different problems to solve.
- Our research project aims to make GPU computing more accessible.
  - Reduce need for expert knowledge.
  - Eliminate need for specialised languages.
  - Avoid rewriting existing code.
Using the GPU

- Flexibility / Performance
- Ease of use / Speed of development
- Assembly
- OpenCL/CUDA
- Libraries
- Autoparallelisation
- Drag-and-drop
Why autoparallelisation?

- Automatically converting C/C++ to OpenCL/CUDA has a number of advantages:
  - Single codebase – Simplifies the process of targeting machines both with and without GPUs.
  - Reuse existing code.
  - Target a wide range of hardware.
  - Achieve independence from specific backend technologies.
  - Avoid lengthy boilerplate code.
How autoparallelisation works

- At its heart, the GPSME toolkit converts C/C++ code into OpenCL/CUDA by following compiler #pragmas.
  - Transfer required data to the GPU
  - Copy the body of a loop into an OpenCL/CUDA program.
  - Execute the program on each core simultaneously.

- This is built on a framework called ROSE, by extending a tool called Mint.
  - See [www.rosecompiler.org](http://www.rosecompiler.org) for more information.
How autoparallelisation works
A simple example

- Keep in mind that the GPU has two key architectural differences compared to the CPU:
  - Multiple cores operating in parallel.
  - Separate memory space.
A simple example

The code below performs a simple low-pass filter (blur) from a source to a destination.

```c
for (y = 1; y < imageHeight-1; y++)
{
    for (x = 1; x < imageWidth-1; x++)
    {
        float sum = 0.0f;
        for(offsetY = -1; offsetY <= 1; offsetY++)
        {
            for(offsetX = -1; offsetX <= 1; offsetX++)
            {
                int finalX = x + offsetX;
                int finalY = y + offsetY;
                sum += srcImage[finalY * imageWidth + finalX];
            }
        }
        dstImage[y * imageWidth + x] = sum / 9.0f;
    }
}
```

The code below performs a simple low-pass filter (blur) from a source to a destination.
A simple example

We can augment this with GPSME directives:

```c
#pragma GPSME copy( srcImage, toDevice, imageWidth, imageHeight)
#pragma GPSME copy( dstImage, toDevice, imageWidth, imageHeight)
#pragma GPSME parallel
{
    #pragma GPSME for nest(2) tile ( 16, 16 )
    for (y = 1; y < imageHeight-1; y++)
    {
        for (x = 1; x < imageWidth-1; x++)
        {
            float sum = 0.0f;
            for(offsetY = -1; offsetY <= 1; offsetY++)
            {
                for(offsetX = -1; offsetX <= 1; offsetX++)
                {
                    //Removed code for brevity
                }
            }
            dstImage[y * imageWidth + x] = sum / 9.0f;
        }
    }
}
#pragma GPSME copy( srcImage, fromDevice, imageWidth, imageHeight)
#pragma GPSME copy( dstImage, fromDevice, imageWidth, imageHeight)
```
A simple example

- The translator is a command line tool which runs under Linux:
  
  `gpsme inputFile.cpp [options]

- Generates output C++ and CUDA in a single file.
- Additional command line options can be provided
  - --shared
  - --register

- For people who don’t run a Linux system the translator can be run via a web interface.
A simple example

The resulting code can be quite large but here are some core snippets:

cudaMemcpy3DParms param_1_dev_1_srcImage = {0};
param_1_dev_1_srcImage.srcPtr = make_cudaPitchedPtr((void*)
    srcImage),(imageWidth) * sizeof(float),(imageWidth),(imageHeight));
param_1_dev_1_srcImage.dstPtr = dev_1_srcImage;
param_1_dev_1_srcImage.extent = ext_dev_1_srcImage;
param_1_dev_1_srcImage.kind = cudaMemcpyHostToDevice;
stat_dev_1_srcImage = cudaMemcpy3D(&param_1_dev_1_srcImage);

if (_gidy >= 0 && _gidy <= imageHeight - 1) {{
    if (_gidx >= 0 && _gidx <= imageWidth - 1) {{
        if ((((_gidx > 0) && (_gidx < (imageWidth - 1))) &&
            (_gidy > 0)) && (_gidy < (imageHeight - 1))) {
            float sum = 0.0f;
            for (_p_offsetY = -1; _p_offsetY <= 1; _p_offsetY++) {
                _index1D = _gidx;
                for (_p_offsetX = -1; _p_offsetX <= 1; _p_offsetX++) {
                    int finalX = (_gidx + _p_offsetX);
                    int finalY = (_gidy + _p_offsetY);
                    sum += srcImage[(finalY * imageWidth) + finalX];
                }
            }
        }
    }
}}
Within your project you can now replace the original C/C++ file with the generated one.

Also set up your project for OpenCL/CUDA
- Install software development kit
- Set up include/linker paths in your project
- Install runtime/drivers
  - This must also be done on target machines.

Watch out for naming conflicts if you keep the old code as well.
A simple example

Several of the GPSME directives are available:

- #pragma GPSME parallel
  - Marks the region to be parallelised.
- #pragma GPSME for
  - A ‘for’ loop to be transferred to the GPU. Options are available to control the way this is split across threads.
- #pragma GPSME barrier
  - Inserts a synchronisation point.
- #pragma GPSME single
  - Marks a region to be executed serially.
- #pragma GPSME copy
  - Performs a memory transfer.
A real world example

```c
int iter = 0;
int iX, iY, iZ;
CPU_FLOAT_TYPE* pTemp;

#pragma GPSME copy(pInputData, toDevice, width, height, depth)
#pragma GPSME copy(pOutputData, toDevice, width, height, depth)
#pragma GPSME copy(pFullMaskData, toDevice, width, height, depth)

#pragma GPSME parallel
{
    for(iter = 0; iter < 50; iter++)
    {
        #pragma GPSME for nest(all) tile(8,8,8)
        for(iZ = 0; iZ < depth; iZ++)
        {
            E = 1.0f + first[0] * first[0] / (first[2] * first[2]);
            L = (2.0f*first[0]*first[2]*second[0 * 3 + 2] - first[0]...
            M = (first[0]*first[2]*second[1 * 3 + 2] +first[1]*first[2]...
            N = (2.0f*first[1]*first[2]*second[1 * 3 + 2] - first[1]...
            ...
        }
    }
}

#pragma GPSME copy(pInputData, fromDevice, width, height, depth)
#pragma GPSME copy(pOutputData, fromDevice, width, height, depth)
#pragma GPSME copy(pFullMaskData, fromDevice, width, height, depth)
```
Practical concerns

- The GPSME toolkit can create huge speedups
  - Depends on underlying code structure.

- The code should:
  - Include (nested) for loops which can be moved to the GPU.
  - Avoid interloop dependencies.
  - Avoid function calls and recursion.
  - Avoid conditional logic.
  - Avoid system operations (allocations, disk access, etc)
  - Avoid dependencies on external libraries.

- The performance increase from parallelism must outweigh the cost of start up and memory transfers.
What if we want to apply multiple passes of our previous filter?

```cpp
for (count = 0; count < 1000; count++)
{
    for (y = 1; y < imageHeight-1; y++)
    {
        for (x = 1; x < imageWidth-1; x++)
        {
            float sum = 0.0f;
            for(offsetY = -1; offsetY <= 1; offsetY++)
            {
                for(offsetX = -1; offsetX <= 1; offsetX++)
                {
                    int finalX = x + offsetX;
                    int finalY = y + offsetY;
                    sum += srcImage[finalY * imageWidth + finalX];
                }
            }
            dstImage[y * imageWidth + x] = sum / 9.0f;
        }
    }
    swap(srcImage, dstImage);
}
```
Interloop dependencies

- In general such interloop dependencies are problematic for all GPUification approaches as they break parallelism.
  - Techniques exist to reduce them but they are limited.
- You should consider whether you can revise your code to remove the dependencies.
- In some cases it would help to add synchronisation primitives to the toolkit. We’re investigating this.
Function calls

- Proper function calls are not supported on all GPU hardware.
  - Functions are usually inlined in the compiled code.
  - GPSME toolkit only supports functions which can be inlined.
  - Recursion is not possible

- Possible workarounds:
  - Make sure the function can be inlined and contains code appropriate for the GPU.
  - Bring the function call outside the loop if it doesn’t really need to be executed every iteration.
  - Split the loop in to two loops – one following the other. Only parallelise one of them.
Conditional logic

- GPUs have a *Single Instruction Multiple Data* (SIMD) architecture.
- All threads follow the same execution path.
  - Relevant when testing boundary conditions (e.g. at edge of image)
- Conditional logic is possible but might not deliver the expected benefits.
  - This was relevant for the MedicSight code.
Conditional logic

```c
#pragma GPSME for nest(2) tile(16,16)
for(int x = 0; x < 128; x++)
{
    for(int y = 0; y < 128; y++)
    {
        float val = someArray[x][y];
        if(val < 0.001f)
        {
            continue; // Optimisation
        }
        else
        {
            // Some expensive code here
        }
    }
}
```
Memory transfers

- GPUs typically have memory which is physically separate from the main system memory.
  - The #pragma GPSME copy directive performs transfers.
- Transfers must be performed immediately before execution of the parallel region.
  - The GPSME toolkit will enforce this.
You should consider:

- **Bandwidth**: There is a limit to the rate at which data can be transferred to the GPU. This rate varies between cards (typically 10-200 Gb/sec).
- **Latency**: There is a small delay between requesting a memory transfer and it actually happening. Therefore one large transfer is faster than several small one.
- **Memory Size**: GPUs typically have between 128Mb to 2Gb of memory, and some is reserved for rendering processes.
Use of External Libraries

- It is common (and generally good practice) to build applications on third-party libraries.
- Unfortunately this causes some problems for parallelisation toolkits.
  - Must be able to see source code to the libraries being used.
  - Libraries must be available on Linux.
  - Libraries cannot be used within parallel regions.
  - Webserver add some extra complications.

How can we work around these issues?
Use of External Libraries

- This is a problem case:

```c
#include <windows.h>
.
.
.
someWindowsFunction();
.
.
#pragma GPSME for nest(2) tile(16,16)
for(int x = 0; x < 128; x++)
{
    for(int y = 0; y < 128; y++)
    {
        //Some code here
    }
}
```
Use of External Libraries

- Solve it by splitting the file in two:

```cpp
// In ‘parallelisable.cpp’ (for example)
#pragma GPSME for nest(2) tile(16,16)
for(int x = 0; x < 128; x++)
{
    for(int y = 0; y < 128; y++)
    {
        //some code here
    }
}

// In main.cpp
#include <windows.h>
#include "parallelisable.h"
  
  someWindowsFunction();
  
  // Now call parallelised function in parallelisable.cpp
```
Use of External Libraries

A more difficult scenario:

```c
#pragma GPSME for nest(2) tile(16,16)
for(int x = 0; x < 128; x++)
{
    for(int y = 0; y < 128; y++)
    {
        .
        .
        // External function call
        cvSomeFunction();
        .
        .
    }
}
```
Use of External Libraries

- When working through the webserver:
  - Make sure the required dependencies are installed.
  - Upload all project-specific headers which are needed.

```cpp
#include "OpenCV.h"
#include "VTK.h"

#include "MyHeader1.h" // Upload this one
#include "MyHeader2.h" // Upload this one

int main(int argc, char** argv)
{
    //Some code here
}
```
Now let’s see how this works on some harder problems…
Polybench benchmark suite

- Collection of micro-benchmarks
- Originally developed for the CPU
- CUDA/OpenCL versions were developed recently

- Implemented OpenMP, OpenACC and GPSME version
- Recently submitted a paper that presents the results
Convolution:
- 2DCONV - 2D convolutional filter
- 3DCONV - 3D convolutional filter

Linear Algebra:
- 2MM - 2 Matrix Multiplications (D=A*B; E=C*D)
- 3MM - 3 Matrix Multiplications (E=A*B; F=C*D; G=E*F)
- ATAX - Matrix Transpose and Vector Multiplication
- BICG - BiCG Sub Kernel of BiCGStab Linear Solver
- GEMM - Matrix-multiply C=alpha.A.B+beta.C
- GESUMMV - Scalar, Vector and Matrix Multiplication
- GRAMSCHMIDT-Gram-Schmidt decomposition
- MVT - Matrix Vector Product and Transpose
- SYR2K - Symmetric rank-2k operations
- SYRK - Symmetric rank-k operations

Datamining:
- CORRELATION - Correlation Computation
- COVARIANCE - Covariance Computation

Stencils:
- FDTD-2D - 2-D Finite Difference Time Domain Kernel
Open standards

- **OpenMP**
  - Open standard for directive-based multi-core programming
  - Most compilers support it by now
  - Easy to harness shared memory multi-core parallelism

- **OpenACC**
  - Open standard for directive-based GPU computing
  - Announced at SC11 [November 2011]
  - Caps, Cray, and PGI are currently providing OpenACC compilers
  - Version 2.0 is to be released soon…
Polybench initial results

- Most tests benefit from speed-ups compared to the OpenMP version.
#pragma acc data copyin(A[NI*NJ],B[NI*NJ]) copyout(C[NI*NJ]){
  #pragma acc kernels loop independent vector(32)
  for (i = 0; i < NI; i++) {
    #pragma acc loop independent vector(32)
    for (j = 0; j < NJ; j++) {
      C[i*NJ + j] = 0.0;
      for (k = 0; k < NK; ++k) {
        C[i*NJ + j] += A[i*NK + k] * B[k*NJ + j];
      }
    }
  }
}

Example – GEMM OpenACC
Example – GEMM GPSME

```c
#pragma GPSME copy(A, toDevice, NI, NJ)
#pragma GPSME copy(B, toDevice, NI, NJ)
#pragma GPSME parallel {
#pragma GPSME for nest(2) tile(32,32)
for (i = 0; i < NI; i++) {
    for (j = 0; j < NJ; j++) {
        C[i*NJ + j] = 0.0;
        for (k = 0; k < NK; ++k) {
            C[i*NJ + j] += A[i*NK + k] * B[k*NJ + j];
        }
    }
}
#pragma GPSME copy(C, fromDevice, NI,NJ)
```
Example – GRAMSchmidt

```c
#pragma GPSME copy(A, toDevice, N, M)
#pragma GPSME copy(R, toDevice, N, M)
#pragma GPSME copy(Q, toDevice, N, M)
#pragma GPSME parallel{
#pragma GPSME for nest(1) tile(128)
  for (k = 0; k < N; k++) {
    nrm = 0;
    for (i = 0; i < M; i++) {
      nrm += A[i*N + k] * A[i*N + k];
    }
    R[k*N + k] = sqrt(nrm);
  }
  for (i = 0; i < M; i++) {
    Q[i*N + k] = A[i*N + k] / R[k*N + k];
  }
  for (j = k + 1; j < N; j++) {
    R[k*N + j] = 0;
    for (i = 0; i < M; i++) {
      R[k*N + j] += Q[i*N + k] * A[i*N + j];
    }
    for (i = 0; i < M; i++) {
    }
  }
}
#pragma GPSME copy(A, fromDevice, N, M)
```

Reduction limits 2\textsuperscript{nd} level parallelization
for (k = 0; k < N; k++) {
    nrm = 0;
    for (i = 0; i < M; i++) {
        nrm += A[i*N + k] * A[i*N + k];
    }
    R[k*N + k] = sqrt(nrm);
    for (i = 0; i < M; i++) {
        Q[i*N + k] = A[i*N + k] / R[k*N + k];
    }
}

#pragma GPSME copy(A,ToDevice, N, M)
#pragma GPSME copy(R,ToDevice, N, M)
#pragma GPSME copy(Q,ToDevice, N, M)
#pragma GPSME parallel{
#pragma GPSME for nest(2) tile(16,16)
for (k = 0; k < N; k++) {
    for (j = k + 1; j < N; j++) {
        R[k*N + j] = 0;
        for (i = 0; i < M; i++) {
            R[k*N + j] += Q[i*N + k] * A[i*N + j];
        }
        for (i = 0; i < M; i++) {
        }
    }
}
#pragma GPSME copy(A,fromDevice, N, M)
Thread blocks can be:
- Full: All threads are part of the iteration space. Resources are not wasted.
- Empty: No thread is part of the iteration space. Resources are not wasted.
- Half-full: This creates divergent branch behavior. Some threads are to be executed, and some are not.
Triangular support increases performance by more than 30 times
Outperforms OpenACC by a good margin on these tests
Tests have been modified to access memory in a 2D manner \(a[i][j]\), as opposed to \(a[i*M+j]\)

GPSME finds extra optimization opportunities by exploiting the 2D access pattern

25% performance increase when using explicit 2D arrays

<table>
<thead>
<tr>
<th></th>
<th>2MM-1D [s]</th>
<th>2MM-2D [s]</th>
<th>SYR2K-1D [s]</th>
<th>SYR2K-2D [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenACC</td>
<td>3.921</td>
<td>8.927</td>
<td>16.671</td>
<td>32.272</td>
</tr>
<tr>
<td>GPSME</td>
<td>3.814</td>
<td>2.812</td>
<td>17.01</td>
<td>12.08</td>
</tr>
</tbody>
</table>
Arithmetic intensity is defined as the ratio between computation and memory load/store.
GPSME is equal or better than OpenACC in all cases
Conclusions on Polybench

- GPSME outperforms OpenACC on the majority of cases:
  - Better register usage
  - Cleaner output code

<table>
<thead>
<tr>
<th>Memory Space</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register memory</td>
<td>≈ 8,000 GB/s</td>
</tr>
<tr>
<td>Shared memory</td>
<td>≈ 1,600 GB/s</td>
</tr>
<tr>
<td>Global memory</td>
<td>≈ 177 GB/s</td>
</tr>
<tr>
<td>Mapped memory</td>
<td>≈ 8 GB/s</td>
</tr>
</tbody>
</table>

Source: Rob Farber
“CUDA Application Design and Development”
Rotasoft Evaluation

- The ASIFT algorithm for feature extraction
  - Keypoint matching
- Rotasoft have successfully evaluated the ASIFT implementations
  - On their own dataset
  - On a dataset provided by the RTD performers
- Matching accuracy is almost the same as with the CPU version
  - Highly invariant to camera viewpoint change
- Main modification: Replaced Array of Structures with Structure of Arrays
Array of Structures vs Structures of Arrays

- GPU global memory is accessed in chunks and aligned.

```c
struct key_aos
{
    int angle;
    int scale;
    int descriptor[128];
};

key_aos *d_keys;
cudaMalloc((void**) &d_keys, ...);

struct key_soa
{
    int * angle;
    int * scale;
    int * descriptor[128];
};

key_soa d_keys;
cudaMalloc((void**) &d_keys.angle, ...);
cudaMalloc((void**) &d_keys.scale, ...);
cudaMalloc((void**) &d_keys.descriptor, ...);
```
Rotasoft Evaluation – Keypoint matching

- Tested on 800x600 image:
  - Computes matches between two sets of around 11,000 keypoints

<table>
<thead>
<tr>
<th>Workstation Type</th>
<th>Rotasoft: Core <a href="mailto:i3@2.1GHz">i3@2.1GHz</a>+GT520M (time in seconds)</th>
<th>Groningen: Core <a href="mailto:i7@3.4GHz">i7@3.4GHz</a>+GTX680 (time in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>69.5</td>
<td>25.9</td>
</tr>
<tr>
<td>OpenMP</td>
<td>25.7</td>
<td>6.7</td>
</tr>
<tr>
<td>Manual GPU</td>
<td>12.5</td>
<td>1.9</td>
</tr>
<tr>
<td>Auto GPU</td>
<td>14.6</td>
<td>3.2</td>
</tr>
</tbody>
</table>

- Speed-up of 6x for a lower grade system
- Speed-up of up to 13.6x for a high-performance system
Rotasoft Evaluation – Keypoint matching
Rotasoft Evaluation

- We continue with evaluating parts of ASIFT keypoint detection, starting with convolution
  - Convolution is about 45-50% of the detection stage
Convolution - GPSME

```c
#pragma GPSME copy (A, toDevice,N,M)
#pragma GPSME copy (B, toDevice,N,M)
#pragma GPSME copy (c, toDevice,3,3)
#pragma GPSME parallel {
#pragma GPSME for nest(2) tile(32,16)
  for (int i = 1; i < M - 1; ++i) {
    for (int j = 1; j < N - 1; ++j) {
      B[i][j] = c[0][0] * A[i - 1][j - 1] + c[0][1] * A[i + 0][j - 1] +
                c[0][2] * A[i + 1][j - 1] + c[1][0] * A[i - 1][j + 0] +
                c[1][1] * A[i + 0][j + 0] + c[1][2] * A[i + 1][j + 0] +
                c[2][0] * A[i - 1][j + 1] + c[2][1] * A[i + 0][j + 1] +
                c[2][2] * A[i + 1][j + 1];
    }
  }
#}  
#pragma GPSME copy (B, fromDevice,N,M)
```
## Convolution performance

- Intel i7@3.4GHz; NVidia GTX680

<table>
<thead>
<tr>
<th></th>
<th>Small data model*</th>
<th>Small data model*</th>
<th>Big data model**</th>
<th>Big data model**</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3x3 kernel [Hz]</td>
<td>5x5 kernel [Hz]</td>
<td>3x3 kernel [Hz]</td>
<td>5x5 kernel [Hz]</td>
</tr>
<tr>
<td>CPU – GCC</td>
<td>486</td>
<td>64.5</td>
<td>2.94</td>
<td>0.44</td>
</tr>
<tr>
<td>PGI OpenACC</td>
<td>4629</td>
<td>2127</td>
<td>26.17</td>
<td>12.33</td>
</tr>
<tr>
<td>GPSME</td>
<td>4901</td>
<td>2785</td>
<td>34.6</td>
<td>16.28</td>
</tr>
</tbody>
</table>

- Speed-up between 10x and 43x vs. CPU code
- Between 5%-30% faster than PGI’s OpenACC

* 1024x1024 image
**12288X12288 image
OpenACC vs. GPSME

- OpenACC advantages:
  - It’s an open standard implemented by compiler vendors.
  - Flexibility
    - Synchronisation, memory and device management, caching.
  - Ease of use (integrated into Visual Studio)

- GPSME advantages:
  - Simplicity
  - Generates cleaner output code
    - CUDA, as well as OpenCL code
  - Doesn’t incur performance penalties for the above advantages
  - Full access to source code makes it easily extendable
GPSME toolkit can deliver large performance gains for some classes of problems.

Better or equal than PGI OpenACC compiler on Polybench

For real-world code, usually some revising of is needed:
- Isolate code you wish to parallelise
- Try to eliminate library and loop dependencies.
- Consider memory transfers, especially inside loops
- Use SoA instead of AoS