Optimized fast wavelet transform utilizing a multicore-aware framework for stencil computations

M. Stürmer, H. Köstler

Para 2010: State of the Art in Scientific and Parallel Computing
Reykjavik, Iceland
6.6.2010
Contents

- Programming Paradigms for Future Architectures
- Image Denoising
- Haar Wavelet
- Numerical Results
- Future Work
Goals

- Obtain realtime performance for a medical image processing application
- Use a framework that makes it easier to write efficient code
- Show basic concepts of framework and discuss its limits
Current Trends

PROGRAMMING PARADIGMS FOR FUTURE ARCHITECTURES
Performance at all costs?

**Performance optimization**
- Optimization techniques
- Architectural factors
- Programming techniques

**Performance engineering**
- Simplicity
- Extendability
- Performance
- Effort
How do we get performance?

- Use parallel algorithms
- Use stencil computations (data locality) and structured grids (access patterns)
  - Memory bandwidth bounded!
- Use best fitting current architectures and their features
  - Try to increase cache reuse or use of local memory
  - Correct memory alignment
  - Vectorize (SIMD) code
- Goal: Provide framework for higher productivity
waLBerla: parallel structured grid framework for various applications
Performance at different scales I

waLBerla
(C++, MPI)

OpenCL framework for stencil based computations on structured grids

Low-level framework for optimized stencil computations (C++, CUDA, intrinsics, Assembler)
waLBerla

heterogeneous devices, distributed memory parallel

Low-level / OpenCL framework

Local device, shared memory parallel
waLBerla Sweep Concept
waLBerla Communication Concept

CPU Buffers → GPU Buffers → MPI Buffers

Apply BC → PCI Express Transfer → GPU - GPU Copy Operations → MPI Communication

MPI_Isend → InfiniBand Transfer
OpenCL framework

- OpenCL (Open Computing Language) is an open standard for heterogeneous parallel computing
- Managed by non-profit technology consortium Khronos group
  - www.khronos.org/opencl
- Analogous to the open industry standards OpenGL for 3D graphics
- OpenCL exploits task-based and data-based parallelism
- Similar to CUDA, OpenCL includes a language (based on C99) for writing kernels executing on devices like GPU, cell, or multi-core processors
OpenCL initialization

1. Get all available platforms (Nvidia, AMD, …)
2. Get all available devices for each platform
3. Load OpenCL source file or binary
4. Allocate data on device and copy data to device
5. Bind a functor to each kernel specifying e.g. device number or data block sizes for parallelization
6. Execute kernel on device
try {
    OpenCL ocl; ocl.init(); ocl.LoadSourceFile(„name.cl“);
    cl::Buffer Buf ( ocl.context(), CL_MEM_READ_ONLY, size*sizeof ( float ), NULL );
    ocl.queue().enqueueWriteBuffer(Buf,CL_TRUE,0, size *sizeof(float),
        begin(),NULL,&events);    ocl.queue().finish();
    cl::KernelFunctor func = ocl.initKernel ( "Kernel",cl::NDRange (ncols(),
        nrows() ));
    cl::Event event = func (Buf,1.0);
    func().wait();
    ocl.queue().enqueueReadBuffer(Buf,CL_TRUE, 0, size * sizeof(float),
        begin(),NULL, &event);  ocl.queue().finish();
} catch(…) {}
OpenCL Example

GPU:

```cpp
__kernel void Kernel(__global float *x, __private float value)
{
    size_t j = get_global_id(0);
    size_t i = get_global_id(1);

    size_t ncols = get_global_size(0);

    x[i * ncols + j] = value;
}
```
Efficient OpenCL

- Kernels adapted to specific architecture 😞
- Vectorize computations
- Define suitable workgroups (parallel running threads that share local memory)
- Reuse local memory: copy to local then perform computations, copy back to global
- Correct data alignment and coalesced memory accesses
Low-level framework architecture

Framework

- Thread management
  - creation, synchronization, affinity
- Data structures
  - alignment, padding (optional)
- Control traversal of grid
  - distribution of work, calling of library kernels

Application Code

- Data structures
  - General configuration threads and their properties, type of per-thread buffers
- Setup of shared data
- Transfer of control
  - grid size, constraints, number of sweeps
- Kernels
  - storage2buffer( tiledesc&, buf& )
  - compute( bufferstack& )
  - buffer2storage( tiledesc&, buf& )
Low-level framework vs. OpenCL

OpenCL
- General
- Supports various architectures
- Adds substantial overhead

Low-level framework
- Specific for bandwidth bounded applications
- Highly optimized
- User has more control
Image Denoising

APPLICATION
Image Denoising of 3D CT Volume

Data: Siemens AG, Healthcare Sector
Assumption:

Relation between an original, unknown image \( u : \Omega \subset R^d \mapsto R \) and an observed image \( u^0 \) can be expressed by

\[
    u^0 = u + \eta
\]

where \( \eta \) stands for the noise.
Image Denoising Models

- **Variational approach**
  - Requires solution of a nonlinear diffusion-based PDE
  - Done by a multigrid solver

- **Sparse Coding (dictionary and patch-based approach)**
  - Image is coded patch-wise by a sparse representation in an overcomplete basis (e.g. DCT basis)
  - Coefficients are computed by batch-OMP algorithm

- **Wavelet-based approach**
  - Thresholding of coefficients based on noise variance
  - Haar wavelet shows to be most efficient
Wavelet Transform

HAAR WAVELET
Haar Wavelet Transform

- Combination of high-pass and low-pass filtering with mask

\[ H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \]

- Mother Wavelet

\[ \psi(t) = \begin{cases} 
1 & \quad 0 \leq t < 1/2, \\
-1 & \quad 1/2 \leq t < 1, \\
0 & \quad \text{otherwise.}
\end{cases} \]
Wavelet Example I

Original Image

Wavelet Transform
Wavelet Example II

Original image (rescaled)
Wavelet runtime results for different architectures

RESULTS
Nvidia GeForce GTX 295

- **Costs**: 450 €
- **Interface**: PCI-E 2.0 x16
- **Shader Clock**: 1242 MHz
- **Memory Clock**: 999 MHz
- **Memory Bandwidth**: 2x112 GB/s
- **FLOPS**: 2x894 GFLOPS
- **Max Power Draw**: 289 W
- **Framebuffer**: 2x896 MB
- **Memory Bus**: 2x448 bit
- **Shader Processors**: 2x240
Memory Architecture

- Constant Memory
- Shared Memory
- Texture Memory
- Device Memory
Cell/B.E. an PowerXCell 8i
Hierarchical Cache Blocking strategy

- Image
- Active memory block in cache
- Thread 1
- Thread 2
- Transformed
- Temporary
- Char
- Float
- Float

Friedrich-Alexander-Universität Erlangen-Nürnberg

Computer Science X - System Simulation Group
Harald Köstler (harald.koestler@informatik.uni-erlangen.de)
Runtime Wavelets: Discussion

- OpenCL on GPU is currently slow
  - OpenCL exhibits up to 20% overhead
  - PCIe overhead
  - OpenCL implementation uses no texture memory
  - OpenCL implementation uses no local memory
  - Simple CUDA implementation was more than twice as fast

- Cell processor
  - Simple OpenCL implementation very slow
  - Low-level framework simplifies programming substantially

- Core i7
  - Big caches
  - Can exploit full potential through low-level framework
Comparison to Multigrid solver

<table>
<thead>
<tr>
<th></th>
<th>MUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i7</td>
<td>726</td>
</tr>
<tr>
<td>PS3</td>
<td>440</td>
</tr>
<tr>
<td>GTX 295</td>
<td>247</td>
</tr>
<tr>
<td>GTX 295 (MG)</td>
<td>348</td>
</tr>
</tbody>
</table>

Mega unknowns per second (MUPS) for Haar wavelet implementations on different architectures and a GPU multigrid solver
Discussions and Future Work

- **Performance Measurement**
  - Are GFLOPs reasonable?
  - Memory bandwidth with intensive cache reuse?
  - Performance challenges for different problems (unknowns per second?)
  - Better tools!

- **Future Work**
  - Heterogeneous CPU-GPU computing
  - OpenCL optimization