Introduction to OpenACC and OpenACC – First Experiences with Real-World Applications

Authors: Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey

Magnus Mikalsen

TDT24 – Parallel Environments and Numerical Methods
November 1, 2012
Overview

• Introduction to OpenACC
  – Background and goal
  – Execution Model
  – OpenACC Constructs (directives)
  – Simple example

• Paper
  – Introduction
  – Test application description
  – Performance evaluation and results
  – Conclusion
What is OpenACC?

- An open standard developed by Nvidia, The Portland Group, CAPS, and Cray.
- The standard describes a API for GPU programming.
  - Compiler directives, library routines, and environment variables.
- Uses compiler directives to indicate what part of the code that should run on the GPU in parallel.
- Goal is to make it easier for non-CUDA/OpenCL experts and increase productivity.
- Portable across OS's, host CPU's, and accelerators.
What is OpenACC?

• Version 1.0 of OpenACC API released November, 2011.
• To be integrated into OpenMP standard.
• Commercial OACC compilers available from PGI, Cray, and CAPS.
• Open source compiler developed at the University of Laguna in Spain called accULL (MIT license).
  – Source to source compiler framework using Yacf, and runtime library called Frangollo.
• Other high-level GPGPU programming models exists:
  – PGI Accelerator Model, CAPS HMPP, hiCUDA, OpenMPC, and StarSs Programming Model.
Some CUDA

• **Execution Model**
  – Threads are grouped into blocks, and blocks grouped into grid.
  – Kernel is executed as a grid of blocks of threads.
  – Adjacent threads execute in scheduling groupings called warps.
  – OCL: Thread == work-item, block == work-group, grid == NDRange

• **Memory Model**
  – Host and Device memory
  – Registers – R/W per-thread
  – Local memory – R/W per-thread
  – Shared memory – R/W per-block
  – Global memory – R/W per-grid
  – Constant memory – R per-grid

• Constrained by the bandwidth between host and device.
Some CUDA
OpenACC Execution Model

- Host-directed execution model. Compute intensive regions are offloaded to accelerator.
- Three levels: **gang**, **worker**, and **vector**.
- Implementation-dependent mapping. Example: Gang == block, worker == warp, and vector == threads.
- Optional to explicitly specify that a loop should map to gangs, workers, and/or vectors.
- Also optional to specify number of gangs/workers/vectors.
OpenACC Constructs

• \#pragma acc directive [clause ...]
  structured block

• Constructs
  – Parallel construct
  – Kernel construct
  – Data construct
  – Loop construct

• General clauses
  – if (condition)
  – async (expression)
Data Construct

• `#pragma acc data [clause ...]`
  Structured block

• Data clauses
  – `copy(list)` – Allocate mem on GPU and copies data from host to GPU when entering region, and copies data back when exiting region.
  – `copyin(list)` – Allocates mem on GPU and copies data from host to GPU when entering region.
  – `copyout(list)` – Allocates mem on GPU and copies data to the host when exiting region.
  – `create(list)` – Allocates mem on GPU but does not copy.
  – `present(list)` – Data is already present on GPU from another containing data region.
  – Also `present_or_copy[in|out]`, `present_or_create`, `deviceptr`. 
Kernels Construct

• `#pragma acc kernels [clause ...]` structured block
• Loop nests in a kernels construct are converted into parallel kernels that run on the GPU.
• Identifies loops that can be executed in parallel and maps abstract loop parallelism onto hardware parallelism (grid and thread for CUDA).
• Clauses
  – General and any data clauses.
Parallel Construct

- `#pragma acc parallel [clause ...]` structured block
- Creates a number of parallel gangs that immediately begin executing the body of the construct.
- Gangs execute code until they reach a work-sharing construct, then split the work of that construct across the threads or gangs.
- Number of gangs/workers remains constant in parallel region.
Loop Construct

- `#pragma acc loop [clause ...]`
  
  loop

- Instructs the worksharing of a loop among the accelerators workers.

- Can be combined with parallel and kernels directives.

- Gives detailed control of the parallel execution of the following loop.
Loop Construct

• Clauses
  – `collaps(n)` – applies directive to the following n nested loops.
  – `seq` – executes the loop sequentially on the GPU.
  – `private(list)` – a copy of each variable in list is created for each iteration of the loop.
  – `reduction(operator:list)` – private variables are combined across iterations.
  – Can also use parallel or kernels clauses if combined with these directives.
Other Directives

- **cache construct** – caches data in software managed data cache (CUDA shared memory).
- **host_data construct** – makes the address of device data available on the host.
- **wait directive** – wait for asynchronous GPU activity to complete.
- **declare directive** – specify that data is to be allocated in device mem of the duration of data region created during the execution of a subprogram.
Example: Jacobi Iteration

- Iteratively converges a correct value (e.g. Temperature) by computing new values at each point from the average of neighbouring points.
- Example of use is to solve Laplace equation in 2D.

\[ A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4} \]
Example: Jacobi Iteration

```c
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
    err = 0.0;

#pragma acc kernels loop reduction(max:err)
    for ( int j = 1; j < n-1; j++ ) {
        #pragma acc loop gang(16) vector(32)
        for ( int i = 1; i < m-1; i++ ) {
            err = max(err, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels loop
    for ( int j = 1; j < n-1; j++ ) {
        #pragma acc loop gang(16) vector(32)
        for ( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
```
Paper

- OpenACC – First Experiences with Real-World Applications
- Authors: Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey
- RWTH Aachen University, Center for Computing and Communication, Germany
Introduction

- HPC architecture trending towards heterogeneous computer systems (perf per watt + price).
- Programming accelerators with low-level API is difficult, may complicate software design and tie code to device or vendor.
- Not ideal for large development projects with long projected code lifetime.
- Solution: make the compiler responsible for the low-level programming tasks by using directive-based high-level API (OpenACC).
Introduction, cont.

• Does not simplify programming accelerators in general, but improve development productivity and simplify code maintenance.

• Paper presents the authors first experience with OpenACC applied to two real-world simulation codes from fields of engineering and medicine.

• This includes performance analysis and evaluation of programmability and productivity.
Test Application 1 – Simulation of Bevel Gear Cutting

- 3D simulation bevel gear cutting process.
- Used in automotive industry.
- Minimize number of tool changes in the production process of bevel gears.
- Contribute to a cost-efficient manufacturing by enabling detailed tool load and wear analysis.
- Compute the intersection of tool and gear using loop nest.
- Loops typically iterates tens of thousands of times.
Test Application 1 – Simulation of Bevel Gear Cutting, cont.

- **basic**, **restruct**, and **locMem** implementations
  - **basic**: divide outer loop amongst all work-items and execute inner loop serially using parallel and loop directives.
  - **restruct**: uses a restructured data format for optimized, coalesced data access (data clauses).
  - **locMem**: store intermediate and input data needed multiple times in the fast software cache of the GPU. Only possible with OpenCL.
Test Application 2 – Neuromagnetic Inverse Problem

- Magnetoencephalography (MEG).
- Magnetic field induced by the current density inside the human brain is measured outside the head.
- Uses minimum p-norm solution to solve the neuromagnetic inverse problem to reconstruct the focal activity in the brain.
- Consists of an objective function and first and second order derivatives computed by automatic differentiation.
- 3 Kernels called about a thousand times each.
Test Application 2 – Neuromagnetic Inverse Problem, cont.

• This includes computing matrix-vector products using matrix dim of 128x512000.

• *basic, blocked and l2par* implementation
  – *basic*: outer loops in parallel and inner loops serial
  – *l2par*: algoritmic simpler version with outer loops distributed to gangs and inner loops in vector mode (only OpenACC version).
  – *blocked*: leverage GPU cache by blocking the matrix-vector multiplication. Also uses loop unrolling (manuel with PGIAcc).

• First and second order derivatives require resolving race conditions by storing intermediate values with globally synchronized access.
Performance Evaluation

• Runtime comparison and analysis using Nvidia Visual Profiler.
• Runtimes include accelerator setup time, data transfer, and kernel execution time.
• Accelerator used: Nvidia Tesla C2050 with ECC enabled and CUDA toolkit 4.0.
• OpenACC code compiled on AMD Magny-Cours 12-core running SUSE ES 11 and Cray 8.1.0 compiler.
• OpenCL and PGIACC code compiled on Intel Westmere 4-cores running Scientific Linux 6.1 and Intel 12.1.2 and PGI 12.3 compilers.
Bevel Gear Cutting Results

**Fig. 1.** Runtimes of OpenCL, PGI Accelerator and OpenACC (engineering application)

**Fig. 2.** Percentages of OpenCL performance in double precision
Bevel Gear Cutting Results, cont.

- Same ratio between basic and restruct version with OpenACC and PGIAcc in both single and double precision.
- Due to L2 read misses in OpenACC program which must be resolved by long-latency global memory access.
- Assume that constant cache is ignored in OpenACC compiler leading to higher L2 read misses.
- On restruct version PGIAcc outperforms OpenCL, and OpenACC gets 80% perf.
- Use of local memory beneficial but the cache directive was not implemented in OpenACC version.
Neuromagnetic Inverse Problem Results

(a) Objective function  (b) 1st-order derivative  (c) 2nd-order derivative

(d) Overall runtimes  (e) Percentages of OpenCL performance
Neuromagnetic IR Results, cont.

- Only Performed in double precision due to high impact of numerical inaccuracy.
- Basic version outperformed by blocked version.
- Blocked version better with OpenCL because of fewer read requests to L2 cache and device global memory (due to lack of caching clause).
- OpenACC performs better using l2par version, almost matching OpenCL.
- Best-effort PGIACC and OpenACC versions achieve about 20% and 40% of OpenCL version.
- OpenACC version is faster than PGIACC version in most cases.
Programmability and Productivity

- OpenACC capable on explicitly managing warps.
- OpenACC restricted to single gang or vector dimension.
- Non-inlined function calls in accelerated code not supported in used OpenACC (and PGIAcc) version.
- No atomic and critical region functionality.
- No asynchronous data transfer.
- OpenACC does not handle multiple GPU's automatically.
Conclusions

• OpenACC gives comparatively good performance on moderately complex kernel of bevel gear cutting sim (80% best-effort).
• This result matches authors expectations of a directive-based programming model, even when using a incomplete implementation.
• Not that good performance on the more complex medical program (40% best-effort).
• Assume mainly due to lack of ability to leverage local memory on the GPU, and authors global synchronization to prevent race conditions.
Conclusions, cont.

• Missing some features (device function calls, user-defined reductions, and critical regions).
• OpenACC ratio of development effort to performance is encouraging (630 vs 46 modified code lines).
• Authors view the move to directive-based accelerator programming to be essential for further growth and acceptance of accelerator devices.
References