Future computing in particle physics: a report from a workshop

Dmitry Emeliyanov (RAL PPD)
Introduction

• The workshop was held in Edinburgh, 15-17 June 2011, organized by NESC and e-Science Institute

• The agenda
  • https://indico.cern.ch/conferenceDisplay.py?confId=141309

• The scope of the workshop:
  • Recent developments in computing and software architectures
  • Effective use of many-core and Graphics Processing Unit (GPU) architectures in a distributed computing environment
  • Utilising emerging I/O and storage technologies
  • Distributed data management in HEP
  • Tools for software performance optimization

• In my talk I will focus on the first two items ...
Current software challenges

Talks by D. Rousseau (for ATLAS) and G. Eulisse (for CMS)

Migration to 64-bit

- Provides larger address space
- Faster execution by 10-30%
But the problem is that 64-bit running causes 30-50% increase in memory consumption

Multi-core CPUs

- Event-level parallelism
  - N application/machine = N cores
  - Applications are independent
  - Memory = N cores x Memory/App
  - x2 if HyperThreading is ON

Dealing with pile-up

Increasing LHC luminosity gives multiple pp interaction pile-ups

- \( <\mu> \sim 20 \), more in 2012 Run
- CPU time scales non-linearly, increase dominated by tracking

More memory is needed but even 2GB/core could be prohibitively expensive

Memory crisis?

- CPU time vs. N of pp interactions

• Addressing these challenges quickly and effectively is not easy due to complexity and development model of the HEP software ...
ATLAS software at a glance

David Rousseau, LAL/CERN

- ATLAS reconstruction framework – Athena runs ~100 algorithms / event
- The software consists of 2000 packages:
  - LOC in millions: C++ : 4.0, Python : 1.4, Fortran : 0.1, Java : 0.1
  - Contributions from 1000 developers in the last 3 years, typically 25 updates/day
- Who are the ATLAS developers ? :
  - Physicists with various degree of s/w expertise
  - A few s/w experts
- It would be unrealistic to expect cutting edge quality in all 2000 packages
- Performance improvement via:
  - Core software (i.e. AthenaMP)
  - “Magic bullet” (compilation options)
  - Fixing hot spots in CPU time / memory
Using multi-core CPUs in ATLAS

Paolo Calafiura, LBNL

- Basic idea of AthenaMP (Multi-Process): split events from one file (LB) into batches, process one batch / CPU core, reduce memory footprint by Copy-On-Write technique:
  - Fork worker processes which see the same initial physical memory,
  - New allocations and touched pages created in private memory space

Maximize the shared memory!

init

OS-fork

Input Files

worker event loop

PARALLEL: WORKER 0:
Events: [0, 5, 8,…96]

PARALLEL: WORKER 1:
Events: [1, 7, 10,…,99]

PARALLEL: WORKER 2:
Events: [3, 6, 9,…,98]

PARALLEL: WORKER 3:
Events: [2, 4, 12,…,97]

Output Files

SERIAL: parent-init-fork

SERIAL: parent-merge and finalize

PPD Seminar, 05/10/2011
AthenaMP experience and plans

Andrew Washbrook, Edinburgh Univ.

- Tests have shown that
  - AthenaMP saves ~0.5Gb of RAM per process
  - Hyper-Threading increases event throughput by 25%
  - pinning a process to a core to prevent Linux scheduler from moving it between cores – gives 20% improvement in event processing rate

- Using AthenaMP in production on Grid:
  - not quite there yet – obviously requires the whole node running the same applications
  - requires an external framework for output files merging
  - In production in 3-6 months, “Whole-node job submission” Task Force is working on it – similar TF in CMS
Using multi-core CPUs in CMS

Giulio Eulisse, FNAL

- Approach similar to AthenaMP: use C-o-W
  - Most (all?) of the common const data / code can actually be brought in the application very early
  - If you fork at that point, the kernel is actually smart enough to share the common data memory pages between parent and the children
  - The kernel “un-shares” the memory pages only when one of the processes writes to them: copy-on-write (CoW)
  - New allocations (i.e. event data) end up in non-shared pages

- Test results:
  - Using reconstruction with 64bit software on 4 CPU, 8 core/CPU 2GHz AMD Opteron(tm) 6128
  - Shared memory per child proc. ~ 700MB
  - Private memory per child proc. ~ 375MB
  - Total memory used by 32 child procs. : 13GB
  - Total memory used by 32 separate jobs: 34GB

- Huge memory saving – but requires whole-node running while on Grid!
• Forking/C-o-W proves to be effective and simple for being considered a good strategy for the short-medium term

• Deployment of whole-node scheduling is a key to exploiting multi-core CPUs on Grid
  ▪ The new processing model requires a new model in computing resources allocation
  ▪ Experiments need to have control over a large quantum of resources as multi-core aware jobs require scheduling of multiple cores at the same time

• The effort which would be required to have module-level parallelism is not worth the actual gain in the current CMS offline software given the decomposition of algorithms:
  ▪ Basically, the problem is dependence of the algorithms which means they can be run only sequentially
News from the industry

- were presented by
  - Roger Goff, Dell LHC Team
  - Alistair Hart, CRAY Exascale Research Initiative
Co-processor architectures

Roger Goff, Dell LHC Team

CPUs+co-processors (GPU) => “New Era of Processor Performance”

(c) AMD
NVIDIA Fermi Architecture

- 3 billion of transistors, 512 cores
  - arranged into 32-core streaming multiprocessors (SM) @ 1.3 GHz
  - L2 (768K) and L1 (64K/SM) caches
  - 16 SMs – up to 16 parallel programs can be run concurrently
- NVIDIA provides CUDA programming environment for software developers

- 32-bit Integer ALU with 64-bit extensions
- Full IEEE-754 32-/64-bit precision

- Each core in a SM has 1K of 32-bit registers, shares up to 48K with other cores
AMD/ATI Cypress Architecture

- Firestream 9350/70 GPUs
  - 1600 FP cores arranged into 20 multiprocessors
  - Performance leader: 2.6 TF single-precision
- The best FLOPS/watt and FLOPS/price ratio
- Not supported by NVidia CUDA
  - developers should use OpenCL SDK (originally from Apple)
Emerging Intel MIC Architecture

• “Knights Ferry” Architecture is based on Many Integrated Cores (MIC) approach:
  • many cores with many threads per core
  • MIC core ~ Fermi multiprocessor but exec. model is closer to MIMD
  • Standard IA programming and memory model

• No actual hardware available yet

• “Knights Corner”: 1st production MIC co-processor in 2\textsuperscript{nd} half of 2012
  • Knowns:
    • 50+ cores
    • 22nm manufacturing process
  • Unknowns:
    • Core frequency, size of on-board memory, ECC support

Baseline “Knights Ferry” architecture: 32 Cores @ 1.2 GHz

✓ 4 threads/core, 128 total parallel threads
✓ 32KB i-cache, 32KB d-cache
✓ 256KB coherent L2 cache (8MB total)
✓ 512bit vector units
## Co-processor Comparison

<table>
<thead>
<tr>
<th></th>
<th>AMD Firestream</th>
<th>NVIDIA Fermi</th>
<th>Intel Knights Ferry</th>
<th>Intel Knights Corner Speculation</th>
<th>Intel Knights Corner Speculation2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cores</strong></td>
<td>1600</td>
<td>512</td>
<td>32*4 threads/core = 128</td>
<td>50*4 threads/core = 200</td>
<td>64*4 threads/core = 256</td>
</tr>
<tr>
<td><strong>Core Frequency</strong></td>
<td>700/825 MHz</td>
<td>1.3 GHz</td>
<td>1.2 GHz</td>
<td>1.2 GHz</td>
<td>2 GHz</td>
</tr>
<tr>
<td><strong>Thread Granularity</strong></td>
<td>fine</td>
<td>fine</td>
<td>coarse</td>
<td>coarse</td>
<td>coarse</td>
</tr>
<tr>
<td><strong>Single Precision Floating Point Capability GFLOPs</strong></td>
<td>2000/2640</td>
<td>1024</td>
<td>614</td>
<td>960</td>
<td>2048</td>
</tr>
<tr>
<td><strong>Double Precision Floating Point Capability GFLOPs</strong></td>
<td>400/528</td>
<td>512</td>
<td>307</td>
<td>480</td>
<td>1024</td>
</tr>
<tr>
<td><strong>GDDR5 RAM</strong></td>
<td>2/4 GB</td>
<td>3-6 GB</td>
<td>1-2 GB</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td><strong>L1 cache/processor</strong></td>
<td>64KB (16KB Shmem, 48KB L1 or 48KB Shmem, 16KB L1)</td>
<td>64KB (32KB icache, 32KB dcache)</td>
<td>64KB (32KB icache, 32KB dcache)</td>
<td>64KB (32KB icache, 32KB dcache)</td>
<td></td>
</tr>
<tr>
<td><strong>L2 cache/processor</strong></td>
<td>768KB shared L2</td>
<td>8MB coherent total (256KB/core)</td>
<td>12MB coherent total (256KB/core)</td>
<td>16MB coherent total (256KB/core)</td>
<td></td>
</tr>
<tr>
<td><strong>programming model</strong></td>
<td>CUDA kernels</td>
<td>posix threads</td>
<td>posix threads</td>
<td>posix threads</td>
<td>posix threads</td>
</tr>
<tr>
<td><strong>virtual memory</strong></td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><strong>memory shared with host</strong></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

The best of what you can buy now
Co-processor Adoption

• Commercial adoption:
  • Oil & Gas/seismic data processing
  • Financial services
  • Ray tracing
  • Molecular dynamics
  • Commercial applications: MATLAB, ANSYS

• Barriers to adoption
  • Lack of parallel programming skills
  • Immature software development environment & standards
    • CUDA vs. OpenCL vs. OpenMP
    • Waiting for the compiler or libraries to abstract the accelerator
  • Uncertainty of benefit vs. effort
    • Amdahl’s law is still the law! Maximum Speedup = \( \frac{1}{(1 - P) + \frac{P}{N}} \)
    • Huge investment in current codes
1. Co-processors are here to stay, but their architectures will continue to evolve.

2. Programming tools will get easier to use and will further integrate co-processing technology.

3. Further abstraction of the underlying co-processor hardware is necessary to achieve broad adoption.

4. Processors from Intel and AMD will integrate co-processors before the end of the decade.

5. Preparing applications for extreme parallelism will enable users to get the most out of future systems.
The new Cray XK6 based on:
- Next generation NVIDIA Fermi X2090 GPU
- 512 cores @ 1.3 GHz, 6GB of memory
- AMD Interlagos CPUs (up to 16 cores)
- Gray Gemini interconnect

XK6 includes Cray Unified x86/GPU programming environment based on OpenMP directives:
- Cray Compiler (C/C++, Fortran), performance analysis tools

Longer term, GPUs are template for Exascale HPC architectures:
- The goal is to achieve 1 EFlops by 2018
- It will require 10 Millions of processing elements (PE)
- Power consumption scales non-linearly – US DoE requirement is to keep it below 20 MW for the Exascale supercomputer

We need lower-power, higher-performing PE, e.g. GPUs
GPUs for HEP: Success story

- A few examples of using GPUs to accelerate HEP applications were presented and discussed
Accelerating ATLAS tracking

Christian Schmitt, Johannes Mattmann, Uni. Mainz

- Reconstruction time depends on hit multiplicity in the detector
- Track finding has worst combinatorial behaviour (as expected) and starts to dominate already at modest multiplicities
- Flowchart of ATLAS track reconstruction:

  - Formation of hits/Clustering
    - Local information only
    - low-medium CPU load
  - Seed finding using
    - Pix/SCT spacepoints
    - Information in small regions
    - high number of combinations
  - Track finding
    - (extension)
    - Information in small regions
    - Geometry and Field access
  - Ambiguity solving
    - Non-local information
    - medium/high CPU load
  - Final track fit
    - Local information, detailed
    - Geometry/Material/Field

- High locality of data
- High arithmetical density
- Ideal for GPU-based parallelization!
Fast ATLAS tracking using GPUs

Christian Schmitt, Johannes Mattmann, Uni. Mainz

- Work on-going to port the offline track finding algorithm to make use of GPUs
- First version that implements the seed finding using spacepoints is ready:
  - each combination of three spacepoints is tested by an individual GPU thread
  - cuts on pT, impact parameter, etc. are applied
- The first results:
  - Test setup: Xeon W3550 (3GHz) CPU vs. NVIDIA GTX460 (Fermi) 1GB video-card
  - GTX460 is a commodity video-card (~£100)
- 10x speed-up on tt-bar MC events
- Bigger speed-up factor for heavy-ion events reported.
GPUs for ATLAS Trigger

- R&D programme for ATLAS higher luminosity upgrade includes High-Level Trigger (HLT) software and hardware

- In general, various upgrade approaches are possible:
  - using more/better CPUs for HLT farms
  - vectorization of HLT software for better utilization of CPUs
  - using GPUs for time-critical parts of HLT code which are suitable for GPU-based parallelization

- The GPU-based option is possible since ATLAS HLT uses dedicated farms which can be, in principle, equipped with GPU cards.

- To study this feasibility a few GPU-accelerated algorithms for ATLAS Level 2 Trigger (LVL2) tracking have been developed.
Tracking at LVL2 Trigger

- LVL2 operates independently on Region-of-Interests (RoI) identified by the Level 1 Trigger:

  - Data preparation
  - Spacepoints
  - RoI
  - Level 1 Trigger

  - Interaction vertex finding:
    - ATLAS z-axis

  - Hough transform in $(\varphi_0, 1/p_T)$ space

  - Track finding by Hough transform
    - Combinatorial track finding

  - Track candidates

  - LVL2

  - Kalman track fit

  - Tracks
GPU-based Level 2 Data Preparation

Jacob Howard, University of Oxford

- ATLAS Pixel and SCT are modular detectors: thousands of modules being read out in parallel
- Readout handled in groups by Read Out Drivers (ROD) and Read Out Buffers (ROB)
- Output data encoded into bytestreams

- Three levels of parallelization:
  - Region-Of-Interest
  - ROB fragment
  - Data words in ROB frag.
- Parallelization at the data word level is the most suitable for GPUs
**Current Results:** speed-up factor for
- CPU: Westmere 2.4 GHz vs.
- GPU: NVIDIA Fermi C2050

<table>
<thead>
<tr>
<th>MC Data, 10^34 pile-up</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>pp min-bias</td>
<td>2.1</td>
</tr>
<tr>
<td>Z-&gt;mu mu</td>
<td>3.2</td>
</tr>
<tr>
<td>tt-bar</td>
<td>3.2</td>
</tr>
</tbody>
</table>

**Future Work**
- Further parallelize bytestream decoding to the word level (option B)
- Add GPU-based Pixel clusterization
- Combine with GPU-based SCT data preparation
GPU-based Trigger Algorithms

- The GPU-based IdScan zFinder (*Chris Jones, Andy Washbrook, University of Edinburgh*)
- Primary interaction vertex finding using histogramming (Hough trf.)
- Highly parallel task – ideal for GPUs
- Concurrent execution – more than one RoI can be processed simultaneously
- 35x speed-up achieved on Fermi GPU

LVL2 Track fitting on GPU (*Dmitry Emeliyanov, RAL*)
- Track-level parallelism: a GPU thread per track – all tracks are fitted in parallel
- 12x speed-up for 3000 tracks – GPU track fitting seems promising for the Kalman filter-based track finding in offline code
Trigger Track Fitter Optimization

Dmitry Emeliyanov, RAL

- A set of optimizations has been applied:
  1. Original code
  2. 32 threads/block
  3. Reduced memory footprint (fewer local variables, upper-triangular covariance matrix)
  4. Track state (cov. + parameters) stored in fast (shared) memory
  5. Jacobian in shared memory to speed-up calculations

- The optimized code gives ~20x speed-up w.r.t. the CPU
- 1.5 microsecond / track has been achieved!
Integration with ATLAS software

• So far our research have been focused mainly on the algorithms:
  – bits of existing code ported to a standalone (i.e. Athena-free) framework and then re-implemented for a GPU using NVIDIA CUDA
  – That’s fine if we want to assess feasibility of GPU-based approach but this is not the complete solution

• Two main problems need to be solved:
  – How the GPU-accelerated code can be used directly in Athena?
  – For more than one Athena application / CPU how the GPU can be shared between applications?
    • important issue for hosts with multi-core CPUs

• Clearly, the issue of integrating GPUs and existing (legacy) HEP software is not ATLAS-specific and must be addressed properly – but surprisingly few ideas on the market ...
“Client-server” architecture

- At RAL, we have developed a “Client-server” solution: Compute server based on NVIDIA CUDA, and CUDA-free clients.

  - **High-level abstraction** of GPU via AthenaComputeSvc which:
    - provides a set of high-level routines: e.g. track finding – ported CPU-intense code which could benefit from GPU acceleration.
    - ComputeSvc talks to the ComputeServer which, in turn, starts the corresponding parallel code (CUDA kernel) on GPU.
Conclusion

• Hardware architectures are evolving quickly:
  - CPUs with 16-20 cores – next year.
  - GPUs seem to be the solution-of-choice for the HPC
  - Integration of CPU and GPU cores on one die:
    - in fact, it’s already available – AMD Fusion CPUs
• HEP software is trying to keep up with this progress:
  - Main focus now is on memory usage optimization
  - Integration of GPUs will require changes in the existing software architectures
Backup slides
CPU+GPU integrated solution

- 1U system from SuperMicro: 2 12-core AMD Opterons + 2 ATI FireStream 9370 GPUs, 1.4 kW PSU. Peak ~ 5.5 TFLOPS
GPU Programming Model

- GPUs are designed for massive parallel SIMT calculations
- Software needs to be written in blocks ("kernels") of instructions
- Kernels are executed in individual threads
- Threads can communicate via shared memory if needed
- Example: matrix sum $C = A + B$ with (9x16) matrices
  - executed by a grid of thread blocks
  - 12 blocks with 12 threads each
  - each block runs on its own SM
  - each thread works with a unique $(i,j)$:
    - $i = \text{threadId.x} + \text{blockId.x} \times \text{blockSize.x}$
    - $j = \text{threadId.y} + \text{blockId.y} \times \text{blockSize.y}$
How to program a GPU?

```c
#include <stdio.h>
#include <cuda.h>

__global__ void square_array(float *a, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) a[idx] = a[idx] * a[idx];
}

int main(void)
{
    float *a_h, *a_d; // Pointer to host & device arrays
    const int N = 10; // Number of elements in arrays
    size_t size = N * sizeof(float);
    a_h = (float *)malloc(size); // Allocate array on host
    cudaMemcpy((void **)&a_d, size); // Allocate array on device
    // Initialize host array and copy it to CUDA device
    for (int i = 0; i < N; i++) a_h[i] = (float)i;
    cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
    // Do calculation on device:
    int block_size = 4;
    int n_blocks = N/block_size + (N % block_size == 0 ? 0 : 1);
    square_array <<< n_blocks, block_size >>> (a_d, N);
    // Retrieve result from device and store it in host array
    cudaMemcpy(a_h, a_d, size, cudaMemcpyDeviceToHost);
    // Print results
    for (int i = 0; i < N; i++) printf("%d %f
", i, a_h[i]);
    // Cleanup
    free(a_h); cudaFree(a_d);
}
```
GPU Programming Issues

• GPU memory can have very different latency:
  • on-card memory: huge (up to 6GB) but takes ~200-400 cycles to get data from
  • registers/shared memory: fast but small and must be used sparingly

• High arithmetic density required:
  • a number of operations per data volume transferred to GPU

• Flow control:
  • data-dependent branching incurs performance penalty as SM evaluates each branch sequentially
  • synchronization between threads on the same SM is possible but also results in performance losses

• Development a good algorithm for a GPU is not trivial but
  • a lot of examples in CUDA SDK – very helpful for grasping basic concepts