USING AND PROGRAMMING NVIDIA GPUS
Max Katz, April 19, 2017
TESLA P100
New GPU Architecture to Enable the World’s Fastest Compute Node

- Pascal Architecture: Highest Compute Performance
- NVLink: GPU Interconnect for Maximum Scalability
- CoWoS HBM2: Unifying Compute & Memory in Single Package
- Page Migration Engine: Simple Parallel Programming with Virtually Unlimited Memory Space
GIANT LEAPS IN EVERYTHING
P100: PERFORMANCE OF LOTS OF SERVERS

Performance Equivalency of Single GPU Server vs Multiple CPU Servers

MOLECULAR DYNAMICS
LAMMPS

QUANTUM CHEMISTRY
VASP

PHYSICS
QUDA

GEOPHYSICS
SPECFEM3D

For benchmark details, input models, ask your sales rep for P100 Performance Guide
# TESLA P100 ACCELERATORS

<table>
<thead>
<tr>
<th></th>
<th>Tesla P100 with NVLink</th>
<th>Tesla P100 for PCIe</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute</strong></td>
<td>5.3 TF DP · 10.6 TF SP · 21.2 TF HP</td>
<td>4.7 TF DP · 9.3 TF SP · 18.7 TF HP</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>HBM2: 732 GB/s · 16 GB</td>
<td>HBM2 16GB: 732 GB/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HBM2 12GB: 549 GB/s</td>
</tr>
<tr>
<td><strong>Interconnect</strong></td>
<td>NVLink (160 GB/s) + PCIe Gen3 (32 GB/s)</td>
<td>PCIe Gen3 (32 GB/s)</td>
</tr>
</tbody>
</table>
| **Programmability** | Page Migration Engine  
Unified Memory                                             | Page Migration Engine  
Unified Memory                                             |
| **Power**        | 300W                                                                                   | 250W                                                                                  |

Interconnect Speed is measured bi-directional.
Directive-based approaches:

OpenMP 4.5

OpenACC

CUDA C/C++/Fortran

OpenCL

Python
#pragma omp target parallel for
for (int i = 0; i < N; ++i)
    y[i] = a * x[i] + y[i];

#pragma acc kernels
for (int i = 0; i < N; ++i)
    y[i] = a * x[i] + y[i];

Let the compiler figure out how to parallelize the work
_cuda C/C++/FORTRAN
Programmer explicitly controls parallelism

__global__ void saxpy(int n, float a, float* x, float* y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) y[i] = a * x[i] + y[i];
}

...

saxpy<<<4096,256>>>(N, 2.0, x, y);
Managed memory allocation is a drop-in replacement for malloc()

On-demand data migration (no need to explicitly offload data to GPU)

Allocate as much memory as there is available system memory

Data motion can be optimized with hints to CUDA runtime
COMPILING GPU CODE

C/C++: nvcc (CUDA), clang, XL (OpenMP)

nvcc is a compiler driver that handles the CUDA parts of your code, passes host code to other compiler (gcc, XL, etc.)

Fortran: PGI (CUDA, OpenACC), XL (CUDA)

CUDA Fortran is compiled directly by the PGI and XL compilers
CUDA LIBRARIES
Drop-in optimized function calls

Linear algebra: cuBLAS / NVBLAS, cuSPARSE, cuSOLVER

Machine learning: cuDNN

Graph analytics: nvGRAPH

C++ STL: Thrust

Collective operations: NCCL

Random number generation: cuRAND

Fast Fourier Transforms: cuFFT
$ nvprof ./saxpy.o

==55960== NVPROF is profiling process 55960, command: ./saxpy.o

==55960== Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>63.44%</td>
<td>1.1495ms</td>
<td>2</td>
<td>574.76us</td>
<td>574.14us</td>
<td>575.39us</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>31.38%</td>
<td>568.67us</td>
<td>1</td>
<td>568.67us</td>
<td>568.67us</td>
<td>568.67us</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>5.18%</td>
<td>93.856us</td>
<td>1</td>
<td>93.856us</td>
<td>93.856us</td>
<td>93.856us</td>
<td>saxpy(int, float, float*, float*)</td>
</tr>
</tbody>
</table>

==55960== API calls:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>96.75%</td>
<td>213.38ms</td>
<td>2</td>
<td>106.69ms</td>
<td>213.38us</td>
<td>213.16ms</td>
<td>cudaMalloc</td>
</tr>
<tr>
<td>0.94%</td>
<td>2.0662ms</td>
<td>3</td>
<td>688.72us</td>
<td>450.80us</td>
<td>967.91us</td>
<td>cudaMemcpy</td>
</tr>
</tbody>
</table>
NVIDIA VISUAL PROFILER
Examine compute kernels and memory transfer
GUIDED ANALYSIS
GUIDED ANALYSIS
DEPENDENCY ANALYSIS

Easily find the critical kernel to optimize

The longest running kernel is not always the most critical optimization target.

CPU

A

wait

B

wait

5%

40%

GPU

Kernel X

Kernel Y

 Timeline

Optimize Here
DEPENDENCY ANALYSIS

Visual profiler

Launch copy_kernel  MemCopy HtoD [sync]  MemCopyDtoH [sync]

Inbound dependencies

Outbound dependencies
NVIDIA TOOLS EXTENSION

API for adding custom profiling ranges/markers
CUDA-GDB
CUDA-aware extension to GDB debugger

$ cuda-gdb ./bitreverse
(cuda-gdb) break main
Breakpoint 1 at 0x18e1: file bitreverse.cu, line 25.
(cuda-gdb) run
Starting program: bitreverse
Breakpoint 1, main () at bitreverse.cu:25
25 void *d = NULL; int i;
CUDA-MEMCHECK
Detect illegal access, race conditions

(cuda-gdb) set cuda memcheck on
(cuda-gdb) run
Starting program: memcheck_demo
[Launch of CUDA Kernel 0 (memset32_post<<<(1,1,1),(64,1,1)>>>)]
Memcheck detected an illegal access to address (@global)0x400100001
Program received signal CUDA_EXCEPTION_1, Lane Illegal Address.
NVIDIA SYSTEM MANAGEMENT INTERFACE

Tools for monitoring GPU status

<table>
<thead>
<tr>
<th>NVIDIA-SMI 384.08</th>
<th>Driver Version: 384.08</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Name</td>
<td>Persistence-M</td>
</tr>
<tr>
<td>Fan</td>
<td>Temp</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>0 Tesla P100-SXM2...</td>
<td>Off</td>
</tr>
<tr>
<td>1 Tesla P100-SXM2...</td>
<td>Off</td>
</tr>
<tr>
<td>2 Tesla P100-SXM2...</td>
<td>Off</td>
</tr>
<tr>
<td>3 Tesla P100-SXM2...</td>
<td>Off</td>
</tr>
</tbody>
</table>

**Processes:**

<table>
<thead>
<tr>
<th>GPU</th>
<th>PID</th>
<th>Type</th>
<th>Process name</th>
<th>GPU Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>No running processes found</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CUDA TOOLS ON CORAL EA SYSTEMS

Executables: /usr/local/cuda/bin/
Libraries: /usr/local/cuda/lib64/
Includes: /usr/local/cuda/include/
CUDA Samples: /usr/local/cuda/samples/
USEFUL ONLINE RESOURCES

Parallel Forall: An Even Easier Introduction to CUDA, Six Ways to SAXPY, etc.

NVIDIA Qwiklabs (CUDA, OpenACC, Deep Learning, etc.)

CUDA Toolkit Documentation

PGI CUDA Fortran Reference Guide

Targeting GPUs with OpenMP 4.5 Device Directives (GTC 2016)

Introduction to OpenACC Online Course (October 2016)