INTRODUCTION TO GPU COMPUTING
Add GPUs: Accelerate Science Applications
ACCELERATED COMPUTING IS GROWING RAPIDLY

- 450+ Applications Accelerated
- 11x GPU Developers
- Available Everywhere

- >730M CUDA Enabled GPUs
- >2200 Universities Teaching CUDA
SMALL CHANGES, BIG SPEED-UP

Application Code

Compute-Intensive Functions

5% of Code

Rest of Sequential CPU Code

GPU

CPU
3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility
3 WAYS TO ACCELERATE APPLICATIONS

- Libraries
  - “Drop-in” Acceleration
- OpenACC Directives
  - Easily Accelerate Applications
- Programming Languages
  - Maximum Flexibility
LIBRARIES: EASY, HIGH-QUALITY ACCELERATION

EASE OF USE  Using libraries enables GPU acceleration without in-depth knowledge of GPU programming

“DROP-IN”  Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes

QUALITY  Libraries offer high-quality implementations of functions encountered in a broad range of applications

PERFORMANCE  NVIDIA libraries are tuned by experts
## GPU ACCELERATED LIBRARIES

“Drop-in” Acceleration for Your Applications

<table>
<thead>
<tr>
<th>DEEP LEARNING</th>
<th>SIGNAL, IMAGE &amp; VIDEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuDNN</td>
<td>cuFFT</td>
</tr>
<tr>
<td>TensorRT</td>
<td>NVIDIA NPP</td>
</tr>
<tr>
<td>DeepStream SDK</td>
<td>CODEC SDK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LINEAR ALGEBRA</th>
<th>PARALLEL ALGORITHMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuBLAS</td>
<td>nvGRAPH</td>
</tr>
<tr>
<td>CUDA Math library</td>
<td>Thrust</td>
</tr>
<tr>
<td>cuSPARSE</td>
<td></td>
</tr>
<tr>
<td>cuSOLVER</td>
<td>NCCL</td>
</tr>
<tr>
<td>cuRAND</td>
<td></td>
</tr>
</tbody>
</table>

- cuDNN
-campaign
- TensorRT
- DeepStream SDK
- cuFFT
- NVIDIA NPP
- CODEC SDK
- cuBLAS
- CUDA Math library
- cuSPARSE
- cuSOLVER
- cuRAND
- nvGRAPH
- Thrust
- NCCL
3 STEPS TO CUDA-ACCELERATED APPLICATION

**Step 1:** Substitute library calls with equivalent CUDA library calls

```c
saxpy ( ... ) ➤ cublasSaxpy ( ... )
```

**Step 2:** Manage data locality

- with CUDA: `cudaMalloc()`, `cudaMemcpy()`, etc.
- with CUBLAS: `cublasAlloc()`, `cublasSetVector()`, etc.

**Step 3:** Rebuild and link the CUDA-accelerated library

```
gcc myobj.o -l cublas
```
DROP-IN ACCELERATION (STEP 1)

int N = 1 << 20;

// Perform SAXPY on 1M elements: y[] = a*x[] + y[]
saxpy(N, 2.0, d_x, 1, d_y, 1);
int N = 1 << 20;

// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

Add “cublas” prefix and use device variables
DROP-IN ACCELERATION (STEP 2)

```c
int N = 1 << 20;
cublasInit();

// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasShutdown();
```

- Initialize cuBLAS
- Shut down cuBLAS
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);

// Perform SAXPY on 1M elements: \( d_y[i] = a \cdot d_x[i] + d_y[i] \)
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements: d_y[i]=a*d_x[i]+d_y[i]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);
cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
ACCELERATING OCTAVE
Scientific Programming Language

Mathematics-oriented syntax
Drop-in compatible with many MATLAB scripts
Built-in plotting and visualization tools
Runs on GNU/Linux, macOS, BSD, and Windows
Free Software

Source: http://www.gnu.org/software/octave/
NVBLAS
Drop-in GPU Acceleration

<table>
<thead>
<tr>
<th>Routines</th>
<th>Types</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>gemm</td>
<td>S,D,C,Z</td>
<td>Multiplication of 2 matrices</td>
</tr>
<tr>
<td>syrk</td>
<td>S,D,C,Z</td>
<td>Symmetric rank-k update</td>
</tr>
<tr>
<td>herk</td>
<td>C,Z</td>
<td>Hermitian rank-k update</td>
</tr>
<tr>
<td>syr2k</td>
<td>S,D,C,Z</td>
<td>Symmetric rank-2k update</td>
</tr>
<tr>
<td>her2k</td>
<td>C,Z</td>
<td>Hermitian rank-2k update</td>
</tr>
<tr>
<td>trsm</td>
<td>S,D,C,Z</td>
<td>Triangular solve, mult right-hand</td>
</tr>
<tr>
<td>trmm</td>
<td>S,D,C,Z</td>
<td>Triangular matrix-matrix mult</td>
</tr>
<tr>
<td>symm</td>
<td>S,D,C,Z</td>
<td>Symmetric matrix-matrix mult</td>
</tr>
<tr>
<td>hemm</td>
<td>C,Z</td>
<td>Hermitian matrix-matrix mult</td>
</tr>
</tbody>
</table>
PERFORMANCE COMPARISON
CPU (openblas) vs GPU (NVBLAS)

Dell C4130 | 128 GB | 36-core, E5-2697 v4 @ 2.30GHz | 4x NVIDIA Tesla P100-SXM2 + NVLINK

SGEMM (GFLOPS)

DGEMM (GFLOPS)
3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility
OpenACC is a directives-based programming approach to parallel computing designed for performance and portability on CPUs and GPUs for HPC.

Add Simple Compiler Directive

```c
main()
{
    <serial code>
    #pragma acc kernels
    {
        <parallel code>
    }
}
```
TOP HPC APPS ADOPTING OPENACC

OpenACC - Performance Portability And Ease of Programming

- ANSYS Fluent
- VASP
- Gaussian

3 of Top 10 Apps

- GTC
- XGC
- ACME
- FLASH
- LSDalton

5 ORNL CAAR Codes

- COSMO
- ELEPHANT
- RAMSES
- ICON
- ORB5

5 CSCS Codes

ANSYS Fluent R18.0 Radiation Solver

- Fluent Native Solver
- Fluent HTC Solver K80 GPU

CPU: (Haswell EP) Intel(R) Xeon(R) CPU E5-2695 v3 @2.30GHz, 2 sockets, 28 cores
GPU: Tesla K80 12+12 GB, Driver 346.46
LSDalton: Quantum Chemistry; 12X speedup in 1 week

Numeca: CFD; 10X faster kernels; 2X faster app

PowerGrid: Medical Imaging; 40 days to 2 hours

INCOMP3D: CFD; 3X speedup

NekCEM: Computational Electromagnetics; 2.5X speedup; 60% less energy

COSMO: Climate Weather; 40X speedup; 3X energy efficiency

CloverLeaf: CFD; 4X speedup; Single CPU/GPU code

MAESTRO: Astrophysics; 4.4X speedup; 4 weeks effort
2 BASIC STEPS TO GET STARTED

Step 1:

```fortran
!$acc data copy(util1,util2,util3) copyin(ip,scp2,scp2i)
!$acc parallel loop
...
!$acc end parallel
!$acc end data
```

Step 2:

```bash
pgf90 -ta=nvidia -Minfo=accel file.f
```
OpenACC DIRECTIVES EXAMPLE

```c
!$acc data copy(A,Anew)
iter=0
do while ( err > tol .and. iter < iter_max )

iter = iter +1
err=0._fp_kind

!$acc kernels
do j=1,m
do i=1,n
Anew(i,j) = .25_fp_kind *( A(i+1,j ) + A(i-1,j ) & +A(i ,j-1) + A(i ,j+1))
err = max( err, Anew(i,j)-A(i,j))
end do
doi
end do

!$acc end kernels
IF(mod(iter,100)==0 .or. iter == 1)    print *, iter, err
A= Anew
end do

!$acc end data```

Copy arrays into GPU memory within data region

Parallelize code inside region

Close off parallel region

Close off data region, copy data back
HETEROGENEOUS ARCHITECTURES

Unified Memory
## OPENACC FOR EVERYONE

New PGI Community Edition Now Available

### PROGRAMMING MODELS

OpenACC, CUDA Fortran, OpenMP, C/C++/Fortran Compilers and Tools

### PLATFORMS

X86, OpenPOWER, NVIDIA GPU

### UPDATES

1-2 times a year | 6-9 times a year | 6-9 times a year

### SUPPORT

User Forums | PGI Support | PGI Professional Services

### LICENSE

Annual | Perpetual | Volume/Site
RESOURCES

FREE Compiler
Success stories
Guides
Tutorials
Videos
Courses
Code Samples
Talks
Books Specification
Teaching Materials
Slack&StackOverflow

Success stories: https://www.openacc.org/success-stories
Resources: https://www.openacc.org/resources
Free Compiler: https://www.pgroup.com/products/community.htm
CUDA PROGRAMMING LANGUAGES
GPU PROGRAMMING LANGUAGES

- **Numerical analytics**: MATLAB, Mathematica, LabVIEW, Octave
- **Fortran**: CUDA Fortran, OpenACC
- **C, C++**: CUDA C++, OpenACC
- **Python**: CUDA Python, PyCUDA, Numba, PyCulib
- **C#**: Altimesh Hybridizer, Alea GPU
- **Other**: R, Julia
void saxpy_serial(int n,
    float a,
    float *x,
    float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);

__global__
void saxpy_parallel(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];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);

CUDA C++ features enable sophisticated and flexible applications and middleware.

Class hierarchies
__device__ methods
Templates
Operator overloading
Functors (function objects)
Device-side new/delete
More...

template<typename T>
struct Functor {
  __device__ Functor(_a) : a(_a) {}  
  __device__ T operator(T x) { return a*x; }
  T a;
};

template<typename T, typename Oper>
__global__ void kernel(T *output, int n) {
  Oper op(3.7);
  output = new T[n]; // dynamic allocation
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n)
    output[i] = op(i); // apply functor
}
CUDA FORTRAN

- Program GPU using Fortran
  - Key language for HPC
- Simple language extensions
  - Kernel functions
  - Thread / block IDs
  - Device & data management
  - Parallel loop directives
- Familiar syntax
  - Use allocate, deallocate
  - Copy CPU-to-GPU with assignment (=)

module mymodule contains
attributes(global) subroutine saxpy(n,a,x,y)
  real :: x(:), y(:), a,
  integer n, i
  attributes(value) :: a, n
  i = threadIdx%x+(blockIdx%x-1)*blockDim%x
  if (i<=n) y(i) = a*x(i) + y(i);
end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  real, device :: x_d(2**20), y_d(2**20)
  x_d = 1.0; y_d = 2.0
  call saxpy<<<4096,256>>>(2**20,3.0,x_d,y_d,)
  y = y_d
  write(*,*) 'max error=', maxval(abs(y-5.0))
end program main
• Numba, a just-in-time compiler for Python functions (open-source!)
  • Numba runs inside the standard Python interpreter
  • Can compile for GPU or CPU!
• Includes Pyculib

```python
import numpy as np
from numba import vectorize

@vectorize(['float32(float32, float32)'], target='cuda')
def Add(a, b):
    return a + b

# Initialize arrays
N = 100000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on GPU
C = Add(A, B)
```
PYTHON - PYCULIB

• Python interface to CUDA libraries:
  
  • cuBLAS (dense linear algebra), cuFFT (Fast Fourier Transform), and cuRAND (random number generation)

• Code generates a million uniformly distributed random numbers on the GPU using the “XORWOW” pseudorandom number generator

```python
import numpy as np
from pyculib import rand as curand

prng = curand.PRNG(rndtype=curand.PRNG.XORWOW)
rand = np.empty(100000)
prng.uniform(rand)
print rand[:10]
```
JULIA

• Up and coming scientific language
• Cross between Python and Matlab
• Interpreter (like Python) -or- compiled (like C/Fortran)
• New approach to multi-processing/multi-node
  • Or use MPI
• Easy to combine it with other languages
• Works with Jupyter Notebooks!
JULIA - SIMPLE EXAMPLE

• Simple matrix multiplication example (integers)
  • Double precision (Int64)
• Can also do elementwise multiplication (just like Matlab)
  • A .* B

julia> A = [1 2 ; 3 4]  
    2x2 Array{Int64,2}:
    1  2
    3  4
julia> B = [10 11 ; 12 13]  
    2x2 Array{Int64,2}:
    10 11
    12 13
julia> A * B  
    2x2 Array{Int64,2}:
    34 37
    78 85
JULIA - GPU EXAMPLE

- **Options:**
  - JuliaGPU (github)
  - Native CUDA (new)
  - GPUarrays
- **Simple native GPU example**

```julia
using CUDAdrv, CUDA

function kernel_vadd(a, b, c)
    # from CUDA: (implicit) CuDeviceArray type, #
    # and thread/block intrinsics
    i = (blockIdx().x-1) * blockDim().x + threadIdx().x

    c[i] = a[i] + b[i]
    return nothing
end

dev = CuDevice(0)
ctx = CuContext(dev)

# generate some data
len = 512
a = rand(Int, len)
b = rand(Int, len)

# allocate & upload on the GPU
d_a = CuArray(a)
d_b = CuArray(b)
d_c = similar(d_a)

# execute and fetch results
@cuda (1,len) kernel_vadd(d_a, d_b, d_c)    # from CUDA

using Base.Test
@test c == a + b

destroy(ctx)
```
JULIA - GPU EXAMPLE

• GPUarrays example
  • Convolution

using GPUArrays, Colors, FileIO, ImageFiltering
using CLArrays
using GPUArrays: synchronize_threads
import GPUArrays: LocalMemory
using CLArrays

img = RGB{Float32}.(load(homedir()*/"/Desktop/backround.jpg"));

a = CLArray(img);
out = similar(a);
k = CLArray(Float32.(collect(Kernel.gaussian(3))));
imgc = similar(img)

# convolution!(a, out, k);
# Array(out)
# outc = similar(img)
# copy!(outc, out)
• Very popular Statistics language
  • Used heavily in Machine Learning
• gpuR package
R - GPUR

- gpuR package
- Simple integer addition of two vectors with 1,000 values

```r
A <- seq.int(from=0, to=999)
B <- seq.int(from=1000, to=1)
gpuA <- gpuVector(A)
gpuB <- gpuVector(B)

C <- A + B
gpuC <- gpuA + gpuB

all(C == gpuC)
```
MATLAB

• Native support for most operations/functions
• Next speaker will cover this 😊
GET STARTED TODAY

These languages are supported on all CUDA-capable GPUs.
You might already have a CUDA-capable GPU in your laptop or desktop PC!

CUDA C/C++

CUDA Python
http://developer.nvidia.com/how-to-cuda-python

Thrust C++ Template Library
http://developer.nvidia.com/thrust

CUDA Fortran

MATLAB
http://www.mathworks.com/discovery/matlab-gpu.html

Mathematica
THANK YOU
SIX WAYS TO SAXPY

Programming Languages for GPU Computing
SINGLE PRECISION ALPHA X PLUS Y (SAXPY)

Part of Basic Linear Algebra Subroutines (BLAS) Library

\[ z = \alpha x + y \]

\( x, y, z \): vector
\( \alpha \): scalar

GPU SAXPY in multiple languages and libraries

A menagerie\(^*\) of possibilities, not a tutorial

\(^*\)technically, a program chrestomathy: http://en.wikipedia.org/wiki/Chrestomathy
**OpenACC COMPILER DIRECTIVES**

*Parallel C Code*

```c
void saxpy(int n,
            float a,
            float *x,
            float *y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

*Parallel Fortran Code*

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    !$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$acc end kernels
end subroutine saxpy
...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

int N = 1<<20;

...  

// Use your choice of blas library

// Perform SAXPY on 1M elements
blas_saxpy(N, 2.0, x, 1, y, 1);

int N = 1<<20;

// Parallel cuBLAS Code

cublasInit();
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements

cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasShutdown();

You can also call cuBLAS from Fortran, C++, Python, and other languages

http://developer.nvidia.com/cublas
void saxpy(int n, float a, 
    float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);

__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];
}

int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);
cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
int N = 1<<20;
std::vector<float> x(N), y(N);
...

// Perform SAXPY on 1M elements
std::transform(x.begin(), x.end(),
               y.begin(), y.end(),
               2.0f * _1 + _2);

---

int N = 1<<20;
thrust::host_vector<float> x(N), y(N);
...

thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

// Perform SAXPY on 1M elements
thrust::transform(d_x.begin(), d_x.end(),
                  d_y.begin(), d_y.end(),
                  2.0f * _1 + _2)
CUDA FORTRAN

**Standard Fortran**

```fortran
module mymodule contains
    subroutine saxpy(n, a, x, y)
        real :: x(:), y(:), a
        integer :: n, i
        do i=1,n
            y(i) = a*x(i)+y(i)
        enddo
    end subroutine saxpy
end module
```

```fortran
program main
    use mymodule
    real :: x(2**20), y(2**20)
    x = 1.0, y = 2.0
    ! Perform SAXPY on 1M elements
    call saxpy(2**20, 2.0, x, y)
end program main
```

**Parallel Fortran**

```fortran
module mymodule contains
    attributes(global) subroutine saxpy(n, a, x, y)
        real :: x(:), y(:), a
        integer :: n, i
        attributes(value) :: a, n
        i = threadIdx%x+(blockIdx%x-1)*blockDim%x
        if (i<=n) y(i) = a*x(i)+y(i)
    end subroutine saxpy
end module
```

```fortran
program main
    use cudafor; use mymodule
    real, device :: x_d(2**20), y_d(2**20)
    x_d = 1.0, y_d = 2.0
    ! Perform SAXPY on 1M elements
    call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)
end program main
```

import numpy as np

def saxpy(a, x, y):
    return [a * xi + yi for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)

import numpy as np
from numba import vectorize

@vectorize([['float32(float32, float32, float32)']], target='cuda')
def saxpy(a, x, y):
    return a * x + y

N = 1048576

# Initialize arrays
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays onGPU
C = saxpy(2.0, X, Y)

http://numpy.scipy.org
https://numba.pydata.org
ENABLING ENDLESS WAYS TO SAXPY

- Build front-ends for Java, Python, R, DSLs
- Target other processors like ARM, FPGA, GPUs, x86

CUDA Compiler Contributed to Open Source LLVM
GPU-ACCELERATED LIBRARIES
cuBLAS

Dense Linear Algebra on GPUs

Complete BLAS Library Plus Extensions

Supports all 152 standard routines for single, double, complex, and double complex

Supports half-precision (FP16) and integer (INT8) matrix multiplication operations

Batched routines for higher performance on small problem sizes

Host and device-callable interface

XT interface supports distributed computations across multiple GPUs

https://developer.nvidia.com/cublas

Up To 5x Faster DeepBench SGEMM Than CPU

- CUDA 8 (cuBLAS 8.0.88); Driver 375.66; P100 (PCIe, 16GB, Base Clocks). ECC OFF
- Host System: Intel Xeon Broadwell Dual E5-2690v4 with Ubuntu 14.04.5 and 256GB DDR4 memory
- MKL 2017.3, Compiler v17.0.4; FP32 Input, Output and Compute
- CPU system; Intel Xeon Broadwell Dual E5-2699v4 (Turbo Enabled) with Ubuntu 14.04.5 and 256GB DDR4 memory
cuFFT
Complete Fast Fourier Transforms Library

Complete Multi-Dimensional FFT Library

“Drop-in” replacement for CPU FFTW library
Real and complex, single- and double-precision data types
Includes 1D, 2D and 3D batched transforms
Support for half-precision (FP16) data types
Supports flexible input and output data layouts
XT interface now supports up to 8 GPUs

https://developer.nvidia.com/cufft

2x Faster Image & Signal Processing than CUDA 8

<table>
<thead>
<tr>
<th>Data Size</th>
<th>1D</th>
<th>2D</th>
<th>3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>64</td>
<td>1.5x</td>
<td>1.5x</td>
<td>1.5x</td>
</tr>
<tr>
<td>16384</td>
<td>2.0x</td>
<td>2.0x</td>
<td>2.0x</td>
</tr>
<tr>
<td>4194304</td>
<td>2.5x</td>
<td>2.5x</td>
<td>2.5x</td>
</tr>
</tbody>
</table>

* V100 and CUDA 9 (r384): Intel Xeon Broadwell, dual socket, E5-2698 v4@ 2.6GHz, 3.5GHz Turbo with Ubuntu 14.04.5 x86_64 with 128GB System Memory
* P100 and CUDA 8 (r361): For cublas CUDA 8 (r361): Intel Xeon Haswell, single-socket, 16-core E5-2698 v3@ 2.3GHz, 3.6GHz Turbo with CentOS 7.2 x86-64 with 128GB System Memory
NPP
NVIDIA Performance Primitives Library

GPU-accelerated Building Blocks for Image, Video Processing & Computer Vision

Over 2500 image, signal processing and computer vision routines

Color transforms, geometric transforms, move operations, linear filters, image & signal statistics, image & signal arithmetic, building blocks, image segmentation, median filter, BGR/YUV conversion, 3D LUT color conversion

Eliminate unnecessary copying of data to/from CPU memory

https://developer.nvidia.com/npp
cuSPARSE
Sparse Linear Algebra on GPUs

Optimized Sparse Matrix Library

- Optimized sparse linear algebra BLAS routines for matrix-vector, matrix-matrix, triangular solve
- Support for variety of formats (CSR, COO, block variants)
- Incomplete-LU and Cholesky preconditioners
- Support for half-precision (fp16) sparse matrix-vector operations

https://developer.nvidia.com/cusparse
CURAND
Random Number Generation (RNG) Library

High Performance Random Number Generation

Flexible interfaces for RNG on the host or within GPU kernels

Pseudo- and Quasi-RNGs — MRG32k3a, MTGP Mersenne Twister, XORWOW, Sobol

Supports several output distributions

Tested against well-known statistical test batteries (test results available in documentation)
cuSOLVER
Linear Solver Library

Library for Dense and Sparse Direct Solvers

Supports Dense Cholesky, LU, (batched) QR, SVD and Eigenvalue solvers (new in CUDA 8)

Sparse direct solvers & Eigen solvers

Includes a sparse refactorization solver for solving sequences of matrices with a shared sparsity pattern

Used in a variety of applications such as circuit simulation and computational fluid dynamics

https://developer.nvidia.com/cusolver

Sample Applications
• Computer Vision
• CFD
• Newton’s method
• Chemical Kinetics
• Chemistry
• ODEs
• Circuit Simulation
nvGRAPH
GPU Accelerated Graph Analytics

Parallel Library for Interactive and High Throughput Graph Analytics

Solve graphs with up to 2.5 Billion edges on a single GPU (Tesla M40)

Includes — PageRank, Single Source Shortest Path and Single Source Widest Path algorithms

Semi-ring SPMV operations provides building blocks for graph traversal algorithms

<table>
<thead>
<tr>
<th></th>
<th>PageRank</th>
<th>Single Source Shortest Path</th>
<th>Single Source Widest Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td></td>
<td>Robotic Path Planning</td>
<td>IP Routing</td>
</tr>
<tr>
<td>Recommendation</td>
<td></td>
<td>Power Network Planning</td>
<td>Chip Design / EDA</td>
</tr>
<tr>
<td>Engines</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Social Ad</td>
<td></td>
<td>Logistics &amp; Supply Chain Planning</td>
<td>Traffic sensitive routing</td>
</tr>
<tr>
<td>Placement</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
AmgX
Algebraic Multi-Grid Solvers

Flexible Solver Composition System

- Easy construction of complex nested solvers and preconditioners
- Flexible and simple high level C API that abstracts parallelism and GPU implementation
- Includes Ruge-Steuben, un-smoothed aggregation, Krylov methods and different smoother algorithms

> 15x Speedup vs HYPRE

- Florida Matrix Collection; Total Time to Solution
- HYPRE AMG Package (http://acts.nersc.gov/hypre) on Intel Xeon E5-2697 v4@2.3GHz, 3.6GHz Turbo, Hyperthreading off
- AmgX on K40, M40, P100 (SXM2); Base clocks
- Host system: Intel Xeon Haswell single-socket 16-core E5-2698 v3 @ 2.3GHz, 3.6GHz Turbo
- CentOS 7.2 x86-64 with 128GB System Memory

https://developer.nvidia.com/amgx