Applied Parallel Computing

parallel-computing.pro

CUDA math libraries

Pavel Yakimov

© APC

CUDA Libraries

- http://developer.nvidia.com/cuda-tools-ecosystem
- CUDA Toolkit
  - CUBLAS – linear algebra
  - CUSPARSE – linear algebra with sparse matrices
  - CUFFT – fast discrete Fourier transform
  - CURAND – random number generation
  - Thrust – STL-like template library
  - NPP – signal and image processing
  - NVCUVENC/NVCUVID – video encoder and decoder libraries

3rd party libraries

- MAGMA – heterogeneous LAPACK and BLAS
- CUSP – algorithms for sparse linear algebra and graph computations
- ArrayFire – comprehensive GPU matrix library
- CULA Tools
- IMSL Fortran Numerial Library
- …
- GPU AI – path finding
- GPU AI for board games

CUBLAS

- BLAS interface implementation
- Column-major addressing, 0- and 1-based indexing
- C compatibility macros

Level	Complexity	Examples
1 (vector-vector)	$O(n)$	AXPY: $y = ax + y$ DOT: $s = (x, y)$
2 (matrix-vector)	$O(n^2)$	GEMV – matrix-vector multiplication
3 (matrix-matrix)	$O(n^3)$	GEMM – matrix-matrix multiplication

CUBLAS

- Naming convention: cublas<T><func>
  - <T> - data type
    - ✓ S – single precision, real number
    - ✓ D – double precision, real number
    - ✓ C – single precision, complex number
    - ✓ Z – double precision, complex number
  - <func> - BLAS literal
  - Example: cublasDgemm
- In API v.2 (CUDA 4.0+) handles are used for thread safety

CUBLAS

- Additional types:
  - `cuComplex, cuDoubleComplex`
  - `cublasHandle_t`
  - `cublasStatus_t`
- Helper functions
  - `cublasCreate() / cublasDestroy()`
  - `cublas{Get|Set}Stream()`
  - `cublas{Get|Set}{Vector|Matrix}[Async]()`

CUBLAS - workflow

- Initialize CUBLAS descriptor (`cublasCreate()`)
- Allocate GPU memory and upload data
- Call all the necessary CUBLAS functions
- Copy data from the GPU to host memory
- Free CUBLAS descriptor (`cublasDestroy()`)

Using CUBLAS

```
#include <stdlib.h>
#include <stdio.h>
#include “cublas.h”
main ()
{
        float *a, *b, *c;
        float *d_a, *d_b, *d_c;
        int lda, ldb, ldc;
        int i, j, n;
        struct timeval t1, t2, t3, t4;
        double dt1, dt2, flops;
        /* CUBLAS initialization */
        cublasInit();
        printf(" n t1 t2 GF/s GF/s\n”);

        for (n=512; n<5120; n*=512) {
            lda = ldb = ldc = 2*n;
```

```
/* Host page-locked memory allocation*/
cudaMallocHost( (void**)&a, n * lda * sizeof(float) );
cudaMallocHost( (void**|&b, n * ldb * sizeof(float) );
cudaMallocHost( (vold**)&c, n * ldc * sizeof(float) );


/*  Filling the matrices  */
for( j = 0; j < n;  j++) {
    for( i = 0; i < n; i++) {
            a[i + j * lda] = (float)rand() / (float)RAND_MAX;
            b[i + j * ldb] = (float)rand() / (float)RAND_MAX;
            c[i + j * ldc] = (float)rand() / (float)RAND_MAX;
    }
}
/* Allocating device memory */
cublasAlloc( n * lda, sizeof(float), (void**)&d_a );
cublasAlloc( n * ldb, sizeof(float), (void**)&d_b );
cublasAlloc( n * ldc, sizeof(float), (void**)&d_c );
```

8 APC

Using CUBLAS

```
/* Copying data from host to device */
gettimeofday (&t1, NULL);
cublasSetMatrix( n, n, sizeof(float), a, lda, d_a, lda);
cublasSetMatrix( n, n, sizeof(float), b, ldb, d_b, ldb);
gettimeofday (&t2, NULL);
/* Performing matrix multiplication */
cublasSgemm('N', 'N', n, n, n, 1.0, d_a, lda, d_b, ldb, 0.0, d_c, ldc);
/* Waiting for multiplication finish */
cudaDeviceSynchronize();
/* Copying data back to host */
gettimeofday (&t3, NULL);
cublasGetMatrlx( n, n, sizeof(float), d_c, ldc, c, ldc);
gettimeofday (&t4, NULL);
/* Clean up */
cublasFree( d_a);
cublasFree( d_b);
cublasFree( d_c);
```

```
    cudaFreeHost(a);
    cudaFreeHost(b);
    cudaFreeHost(c);

    tdiff1 = t4.tv_sec - t1.tv_sec + 1.0e-6 * (t4.tv_usec – t1.tv_usec);
    tdiff2 = t3.tv_sec - t2.tv_sec + 1.0e-6 * (t3.tv_usec - t2.tv_usec);
    flops = 2.0 * (double)n * (double)n * (double)n;
    /* Printing execution time */
    printf( "%4d %8.5f %8.5f %5.0f %5.0f\n", n, dt1, dt2,
                1.0e-9 * flops / tdiff1,1.0e-9 * flops/tdiff2);
  }

  cublasShutdown();
  return 0;
}
```

CUSPARSE

- BLAS-like interface implementation for sparse matrices
- Sparse = a lot of zero elements
- Formats:
  - Dense format (often ineffective)
  - COO: Coordinate
  - CSR/CSC: Compressed Sparse Row/Column
  - ELL: Ellpack-Itpack
  - HYB: Hybrid
  - BSR: Block Compressed Sparse Row

Sparse Formats: COO

$$A = \begin{pmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 0 & 9 & 0 & 6 \end{pmatrix}$$

- nnz = 9
- cooValA = [1 4 2 3 5 7 8 9 6]
- cooRowIndA = [0 0 1 1 2 2 2 3 3]
- cooColIndA = [0 1 1 2 0 3 4 2 4]

Sparse Formats: CSR

$$A = \begin{pmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 0 & 9 & 0 & 6 \end{pmatrix}$$

- nnz = 9
- cooValA = [1 4 2 3 5 7 8 9 6]
- cooRowIndA = [0 2 4 7 9]
- cooColIndA = [0 1 1 2 0 3 4 2 4]

Sparse Formats: CSC

$$A = \begin{pmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 0 & 9 & 0 & 6 \end{pmatrix}$$

- nnz = 9
- cooValA = [1 5 4 2 3 9 7 8 6]
- cooRowIndA = [0 2 0 1 1 3 2 2 3]
- cooColIndA = [0 2 4 6 7 9]

CUSPARSE – features

- 4 levels : cusparse<T><func>
  - Sparse and dense vectors
  - Sparse matrices and vectors
  - Sparse matrices and dense matrices
  - Format conversions
- Single/Double Precision, Real/Complex values

CUSPARSE – workflow

- Initialize descriptor (`cusparseCreate()`)
- Allocate GPU memory and upload data
- Call all the necessary CUSPARSE functions
- Copy data from the GPU to host memory
- Free CUBLAS descriptor(`cusparseDestroy()`)

CUFFT

- Interface similar to FFTW (FFTW compatibility)
- 1D, 2D and 3D forward and inverse DFT
- Single/Double Real/Complex
- Up to 128M single precision elements in each dimension, 64M for double precision
- CUDA Streams support (Asynchronous transforms)

CUFFT

```
#include <stdlib.h>
#include <stdio.h>
#include “cufft.h”
#define NX 256
#define NY 128
main()
{
        cufftHandle plan;
        cufftComplex *idata, *odata;
        int i;

        cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
        cudaMalloc((void**)&odata)sizeof(cufftComplex)*NX*NY);
        for( i=0; i<NX*NY;i++){
              idata[i].x = (float)rand() / (float)RAND_MAX;
              idata[i].y = (float)rand() / (float)RAND_MAX;
        }
```

```
/* Create a 2D FFT plan*/
cufftPlan2d(&plan, NX, NY, CUFFT_C2C);

/* Use the CUFFT plan to transform the signal out of place.
*  Note: idata != odata indicates an out of place
*  transformation to CUFFT at execution time. */
cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);
/* Inverse transform the signal in place */
cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);
/* Destroy the CUFFT plan*/
cufftDestroy(plan);
cudaFree(idata);
cudaFree(odata);
return 0;
}
```

CURAND

- Pseudo- and Quasi-Random Number Generation
- XORWOW, MRG32K3A, MTGP32 and SOBOL algorithms of generation
- Distributions:
  - Uniform
  - [Log]Normal
  - Poisson
- Has 2 interfaces: for device and for host

CURAND (Host API)

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand.h>

main()
{
        int i, n = 100;
        curandGenerator_t gen;
        float *devData, *hostData;

        /* Allocate n floats on host */
        hostData = (float *)calloc(n, sizeof(float));

        /* Allocate n floats on device*/
        cudaMalloc((void**)&devData, n * sizeof(float));

        /*Create pseudo-random number    generator*/
        curandCreateGenerator(&gen,
                CURAND_RNG_PSEUDO_DEFAULT);

        /* Set seed */
        curandSetPseudoRandomGeneratorSeed(gen, 1234ULL) ;
        /*Generate n floats on device */
        curandGenerateUniform(gen, devData, n);
        /* Copy device memory to host */
        cudaMemcpy(hostData, devData, n * sizeof(float),
                                cudaMemcpyDeviceToHost);


        /* Show result */
        for (i = 0; i < n; i++) {
                printf ("%1.4f ", hostData[i]);
        }

        printf("\n");

        /* Cleanup */
        curandDestroyGenerator(gen);
        cudaFree(devData);
        free(hostData);
        return 0;
}
```

CURAND (Run on CPU)

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand.h>

main()
{
        int i, n = 100;
        curandGenerator_t gen;
        float *hostData;

        /* Allocate n floats on host */
        hostData = (float *)calloc(n, sizeof(float));

        /*Create pseudo-random number    generator*/
        curandCreateGeneratorHost(&gen,
                CURAND_RNG_PSEUDO_DEFAULT);

        /* Set seed */
        curandSetPseudoRandomGeneratorSeed(gen, 1234ULL) ;
        /*Generate n floats on host */
        curandGenerateUniform(gen, hostData, n);


        /* Show result */
        for (i = 0; i < n; i++) {
                printf ("%1.4f ", hostData[i]);
        }

        printf("\n");

        /* Cleanup */
        curandDestroyGenerator(gen);
        free(hostData);
        return 0;
}
```

CURAND (Device API)

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h. >

__global__ void setup_kernel(curandState *state)
{
        int id = threadIdx.x + blockIdx.x * 64;
        /* Each thread gets same seed, a different
           sequence number, no offset */
        curand_init(1234, id, 0, &state[id]);
}

__global__ void generate_kemel(curandState ‘state, int *result)
{
        int id = threadIdx.x + blockIdx.x * 64;
        int count = 0;
        unsigned int x;
        /* Copy state to local memory for efficiency */
        curandState localState = state[id];

        /* Generate pseudo-random unsigned ints */
        for (int n = 0; n < 100000; n++) {
                x = curand(&localState);
                /* Check if low bit set */
                if (x & 1) count++;
        }

        /* Copy state back to global memory */
        state[id] = localState;

        /* Store results */
        result[id] += count;
}
```

CURAND (Device API)

```
main()
{
    int i, total;
    curandState *devStates;
    int *devResults, *hostResults;
    /* Allocate space for results on host */
    hostResults = (int*)calloc(64 * 64, sizeof(int));
    /* Allocate space for results on device */
    cudaMalloc((void**)&devResults, 64 * 64 * sizeof(int));
    /* Set results to 0 */
    cudaMemset(devResults, 0, 64 * 64 * sizeof(int));
    /* Allocate space for rng states on device */
    cudaMalloc((void**)&devStates, 64 * 64 *
                        sizeof(curandState));
    /* Setup prng states */
    setup_kernel<<<64, 64>>>(devStates);
    /* Generate and use pseudo-random */
    for (i = 0; i < 10; i++){
    generate_kernel<<<64, 64>>>(devStates, devResults);
}
    /* Copy device memory to host */
    cudaMemcpy(hostResults, devResults, 64 * 64 * sizeof(int),
                        cudaMemcpyDeviceToHost);
    /* Show result */
    total = 0;
    for (i = 0; i < 64 * 64; i++) {
        total += hostResults[i];
    }
    printf("Fraction with low bit set was %10.13f\n",
        (float)total / (64.0f * 64.0f * 100000.0f ^ 10.0f));

    /* Cleanup */
    cudaFree(devStates);
    cudaFree(devResults);
    free(hostResults);
    return 0;
}
```

NPP: Image & Signal Processing

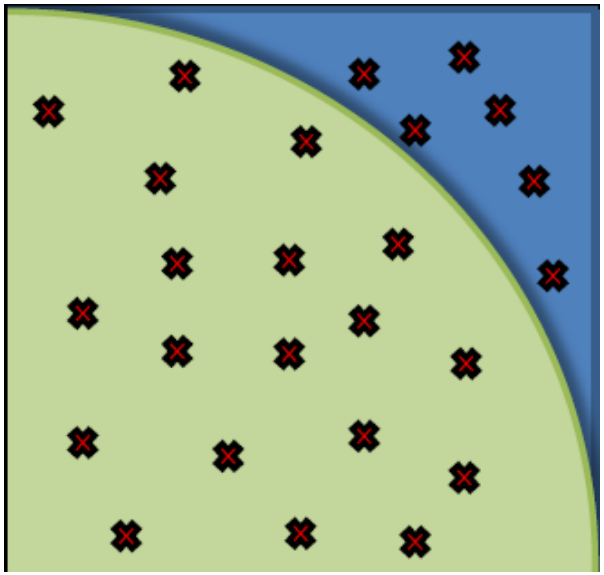
- Similar to IPP

- Arithmetic and logical operations
- Color model conversion
- Compression
- Filtering Functions
- Geometry transforms
- Statistics functions

- A comprehensive GPU matrix library:
  - Linear Algebra
  - Signal&image processing
  - Statistics
  - Code timing
  - Graphics
- Unified array container type:
  - Single/Double Real/Complex [Un]signed + Boolean
  - Easy index manipulation (Matlab-like)
  - Parallel for loops and multi-gpu scaling

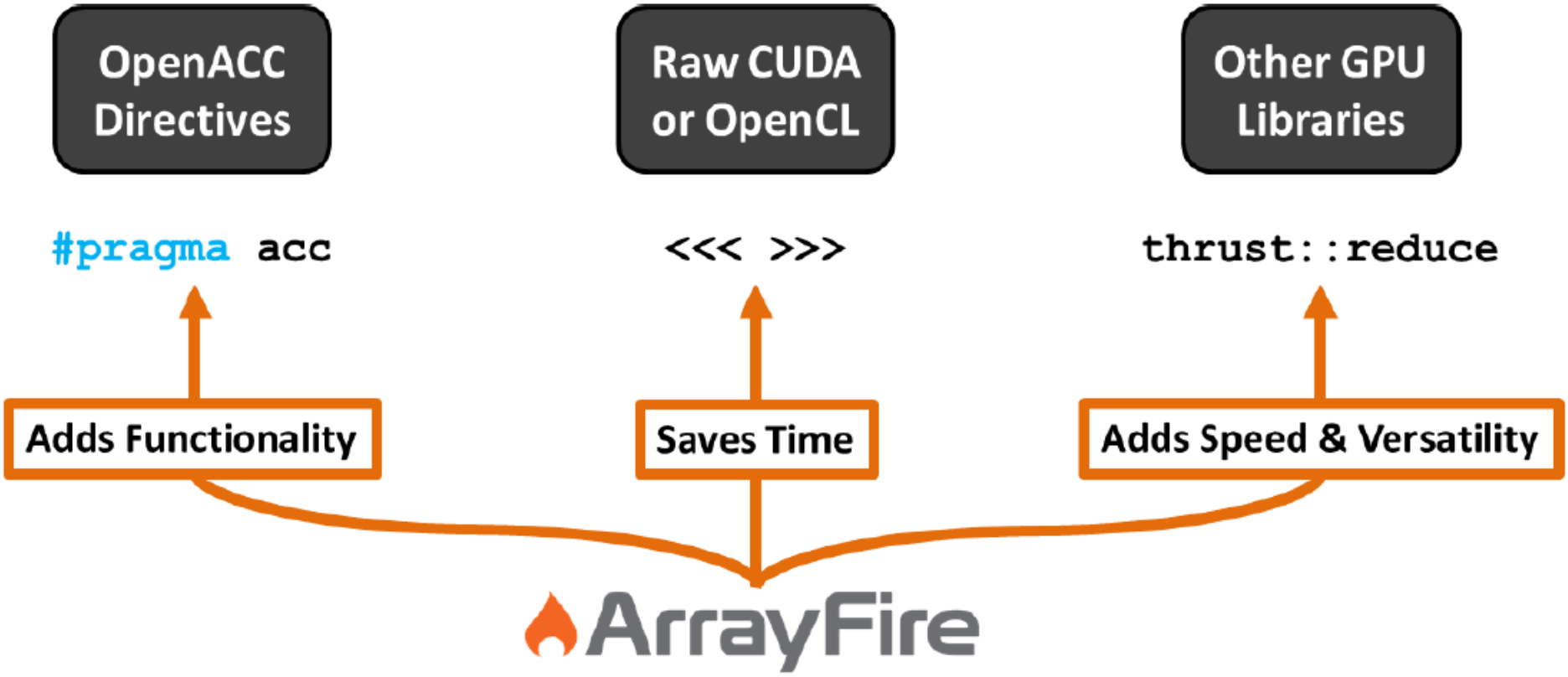
ArrayFire

Example: Pi value



```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main()
{
    // 20 million random samples
    int n = 20e6;
    array x = randu(n,1), y = randu(n,1);
    // how many fell inside unit circle?
    float pi = 4 * sum<float>(sqrt(mul(x,x)+mul(y,y))<1) / n;
    printf("pi = %g\n", pi);
    return 0;
}
```

ArrayFire


OpenACC Directives
Raw CUDA or OpenCL
Other GPU Libraries
#pragma acc
<<< >>>
thrust::reduce
Adds Functionality
Saves Time
Adds Speed & Versatility
ArrayFire

Conclusion

- If you are not a professional in some area – use libraries
- If you think you are a professional in particular area – use libraries at the beginning
- Do not worry if you cannot implement a routine more efficient than in library
- Sometimes everything above is wrong. But only sometimes.

Questions?

Pavel Yakimov

yakimov@parallel-computing.pro