

Using OpenACC With CUDA Libraries

John Urbanic
with NVIDIA
Pittsburgh Supercomputing Center

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

Programming
Languages

CUDA Libraries are
interoperable with OpenACC

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

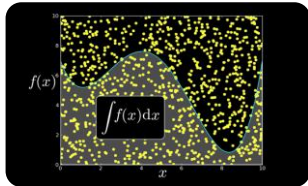
Programming
Languages

Maximum
Flexibility

CUDA Languages are
interoperable with OpenACC,
too!



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP

GPU VSIPL

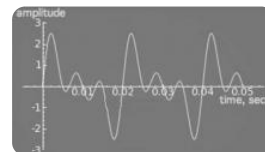
Vector Signal
Image Processing

CULA | tools

GPU Accelerated
Linear Algebra



Matrix Algebra on
GPU and Multicore



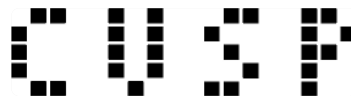
NVIDIA cuFFT



IMSL Library



Building-block
Algorithms for CUDA



Sparse Linear
Algebra



C++ STL Features
for CUDA



GPU Accelerated Libraries
“Drop-in” Acceleration for Your Applications

CUDA data in OpenACC

You have to allocate data memory on the host and device with `alloc/cudaMalloc`. `deviceptr()` lets OpenACC know that has happened.

```
float *a;
...
err = cudaMalloc(&a, sizeof(float)*n);
kernel<<<n/32,32>>>(a,...);
...
incr(a,n);

void incr(float* x, int n){
    #pragma acc parallel loop deviceptr(x)
    for (int i = 0; i < n; ++i)
        x[i] += 1.0f;
}
```

deviceptr Data Clause

`deviceptr(list)` Declares that the pointers in *list* refer to device pointers that need not be allocated or moved between the host and device for this pointer.

Example:

C

```
#pragma acc data deviceptr(d_input)
```

Fortran

```
$(acc data deviceptr(d_input))
```

host_data Construct

If the data is on the device - say it has been *create()*ed - then `host_data use_device()` allows us to grab that device pointer on the host so that we can pass it along to some CUDA routine elsewhere.

```
a = (float*)malloc(sizeof(float)*n);
#pragma acc data create(a[0:n])
{
    #pragma acc host_data use_device(a)
    {
        incr(a,n);
    }
}

----- separate file with CUDA code -----
__global__ inkernel(float* x, int n){ ... }

void incr(float* x, int n){
    inkernel<<<n/32,n>>>(x,n);
}
```

Example: 1D convolution using CUFFT

- Perform convolution in frequency space
 1. Use CUFFT to transform input signal and filter kernel into the frequency domain
 2. Perform point-wise complex multiply and scale on transformed signal
 3. Use CUFFT to transform result back into the time domain
- We will perform step 2 using OpenACC
- Code highlights follow. Code available with exercises in: `Exercises/OpenACC/Cuffft-acc`

Source Excerpt

Allocating Data

```
// Allocate host memory for the signal and filter
Complex *h_signal = (Complex *)malloc(sizeof(Complex) * SIGNAL_SIZE);
Complex *h_filter_kernel = (Complex *)malloc(sizeof(Complex) * FILTER_KERNEL_SIZE);
.
.
.

// Allocate device memory for signal
Complex *d_signal;
checkCudaErrors(cudaMalloc((void **)&d_signal, mem_size));
// Copy host memory to device
checkCudaErrors(cudaMemcpy(d_signal, h_padded_signal, mem_size, cudaMemcpyHostToDevice));

// Allocate device memory for filter kernel
Complex *d_filter_kernel;
checkCudaErrors(cudaMalloc((void **)&d_filter_kernel, mem_size));
```

Source Excerpt

Sharing Device Data (d_signal, d_filter_kernel)

```
// Transform signal and kernel
error = cufftExecC2C(plan, (cufftComplex *)d_signal, (cufftComplex *)d_signal, CUFFT_FORWARD);
error = cufftExecC2C(plan, (cufftComplex *)d_filter_kernel, (cufftComplex *)d_filter_kernel, CUFFT_FORWARD);

// Multiply the coefficients together and normalize the result
printf("Performing point-wise complex multiply and scale.\n");
complexPointwiseMulAndScale(new_size, (float *restrict)d_signal, (float *restrict)d_filter_kernel);

// Transform signal back
error = cufftExecC2C(plan, (cufftComplex *)d_signal, (cufftComplex *)d_signal, CUFFT_INVERSE);
```

OpenACC
Routine

CUDA
Routines

OpenACC Convolution Code

```
void complexPointwiseMulAndScale(int n, float *restrict signal,
                                float *restrict filter_kernel)
{
    // Multiply the coefficients together and normalize the result
    #pragma acc data deviceptr(signal, filter_kernel)
    {
        #pragma acc kernels loop independent
        for (int i = 0; i < n; i++) {
            float ax = signal[2*i];
            float ay = signal[2*i+1];
            float bx = filter_kernel[2*i];
            float by = filter_kernel[2*i+1];
            float s = 1.0f / n;
            float cx = s * (ax * bx - ay * by);
            float cy = s * (ax * by + ay * bx);
            signal[2*i] = cx;
            signal[2*i+1] = cy;
        }
    }
}
```

Implementation note: We cast the Complex* pointers to float* pointers and use interleaved indexing

Linking CUFFT

- `#include "cufft.h"`
- Compiler command line options:

```
CUDA_PATH = /opt/pgi/13.10.0/linux86-64/2013/cuda/5.0  
CCFLAGS = -I$(CUDA_PATH)/include -L$(CUDA_PATH)/lib64  
          -lcudart -lcufft
```

Must use
PGI-provided
CUDA toolkit paths

Must link lib cudart
and libcufft

Result

```
instr009@nid27635:~/Cufft> aprun -n 1 cufft_acc  
Transforming signal cufftExecC2C  
Performing point-wise complex multiply and scale.  
Transforming signal back cufftExecC2C  
Performing Convolution on the host and checking correctness
```

```
Signal size: 500000, filter size: 33  
Total Device Convolution Time: 6.576960 ms (0.186368 for point-wise convolution)  
Test PASSED
```



CUFFT + cudaMemcpy



OpenACC

Summary

- Use `deviceptr` data clause to pass pre-allocated device data to OpenACC regions and loops
- Use `host_data` to get device address for pointers inside acc data regions
- The same techniques shown here can be used to share device data between OpenACC loops and
 - Your custom CUDA C/C++/Fortran/etc. device code
 - Any CUDA Library that uses CUDA device pointers