ALYA Multi-Physics System on GPUs: Offloading Large-Scale Computational Mechanics Problems

Vishal Mehta
Engineer, Barcelona Supercomputing Center
vishal.mehta@bsc.es
Training
• Build an education program on parallel programming using CUDA, OpenCL and OmpSs
• PUMPS summer school 2010-2016, courses at BSC and UPC

Research
• Porting applications for volcanic ash dispersal, Alya multiphysics, Interactive Drug Design etc on GPUs.
• Enabling GPU support for OmpSs task based model and GPU optimizations.
ALYA Multi-Physics System

Volcanic ash dispersal
Drug delivery
Bio-Mechanics

Solid Mechanics
Climate Modeling
Sniff

ALYA ShowReel
OmpSs Programming Model

- Extended OpenMP & Task based programming model
  - Mercurium Compiler
  - Nanos Runtime

- Forerunner for OpenMP
  - Tasking and tasks dependences are two examples of OmpSs influence

- OmpSs Current Accelerator Supports
  - Tasks to be executed on GPU programmed in CUDA or OpenCL
  - Runtime system takes care of data movement, overlapping and scheduling
  - Doesn’t generate gpu code

```cpp
#pragma omp task [ in (...) ] [ out (...) ] [ inout (...) ]
{
    <<.. function or code block ..>>
}
#pragma omp taskwait
```

To compute dependences

Wait for sons
Outline

1. ALYA multiphysics architecture
2. CUDA and Fortran
3. Algorithms for linear systems
4. Optimization challenges
5. OmpSs GPU
6. Performance Evaluation
7. Future Work
• Solves incompressible flows, turbulence models, compressible flows, solid mechanics, species transport, thermal flows, excitable media etc using finite element method.
• It scales upto 100,000 cores on NCSA blue waters using MPI + OpenMP.
• It uses generalized preconditioned linear solvers like GMRES, Deflated CG, MINRES etc.
• It also includes tools for mesh sub-division for improving simulation accuracy, smoothing of meshes and domain decompositions.
CUDA and Fortran : Example

Program p
Implicit none
Real*8 :: a(1:100), b(1:100), out(1:100)
Integer*8 :: d_a,d_b,d_out

Call gpmalloc(d_a,8*100)
Call gpmalloc(d_b,8*100)
Call gpmalloc(d_out,8*100)

Call memcpytogpu(a,d_a,8*100)
Call memcpytogpu(a,d_a,8*100)

Call GPUADD(d_a,d_b,d_out,100)

Call memcpyfromgpu(d_out,out,8*100)
End program p

#define devptr_t size_t
extern ``C'' void gpmalloc_(devptr_t *pt,int *N)
{void *x; cudaError_t err;err = cudaMalloc(&x,*N);
if(err != cudaSuccess){printf(\"malloc error %s\",cudaGetErrorString(err));exit(1);}
*pt = (devptr_t)x; return;}

extern ``C'' void memcpytogpu_ (devptr_t *dst, devptr_t *src, int *N)
{ cudaError_t err;err = cudaMemcpy((void*)(*dst),
(void*)src,*N, cudaMemcpyHostToDevice);
if(err != cudaSuccess){printf(\"memcpy error %s\",cudaGetErrorString(err));exit(1);
} return;}

extern ``C'' void memcpyfromgpu_ (devptr_t *dst,devptr_t *src, int *N)
{ cudaError_t err;err = cudaMemcpy((void*)dst,
(void*)src,*N, cudaMemcpyDeviceToHost);
if(err != cudaSuccess){printf(\"memcpy error %s\",cudaGetErrorString(err));exit(1);
}return;}

__global__ void gpuaddkernel(double *a,double *b,double *c,int N)
{ int tx = threadIdx.x + blockIdx.x*blockDim.x;
if(tx<N){ c[tx] = a[tx] + b[tx];}}

extern ``C'' void gpumalloc_(devptr_t *a,devptr_t *b,devptr_t *c, int *N){
gpumalloc<<<((int)((*N)/threads)+1, threads>>>((*a),(*b),(*c),(c)*N); }
Algorithm: Pipelined Conjugate gradient

- This algorithm provides a good test case with parallelism in terms of overlapping CPU, GPU and MPI communications.

- High computation efficiency of GPUs, can be overlapped with MPI.
Algorithm: GMRES

- This algorithm is computationally heavy owing to presence of global orthogonalization in Krylov Subspace.

- CPU, GPU and MPI tasks are interleaved.
The performance optimization in hybrid CPU+GPU codes, require platform specific optimizations as well as non-platform specific optimizations.

**Platform Specific Optimizations**
- Overlapping memory copies, compute kernels and independent compute kernels using CUDA streams.
- Managing CUDA memories and computation kernels across multiple GPUs using CUDA streams.

**Non-Platform Specific Optimizations**
- Overlapping GPU work load with CPU threads using OpenMP.
- Overlapping MPI communication with CPU and GPU workloads.
OmpSs GPU

- OmpSs provides compiler and runtime support for GPU based tasks.
- Analyzes dependencies between tasks independent of the platform.
- Move data across the platform as required by the tasks based on dependency graph.
- Schedules parallel tasks across multiple platforms including multi-GPUs and multicore CPUs.
- Overlaps data transfers with GPU tasks as well as among independent GPU tasks.
- Provides support for NVIDIA libraries like cuSparse and cuBLAS.
OmpSs GPU Support: Compiler

Program p
implicit none

interface
  |OMP TARGET DEVICE(CUDA) NDRANGE(1,N,128)
  FILE(ker.cu) COPY_DEPS
  OMP TASK IN(A,B) INOUT(C)
SUBROUTINE vecadd(N,a,b,c)
  IMPLICIT NONE
  INTEGER*4, VALUE :: N
  real*8 :: a(N),b(N),c(N)
END SUBROUTINE vecadd

|OMP TARGET DEVICE(CUDA) FILE(hker.cu)
NO_COPY_DEPS COPY_IN(a,b)
|OMP TASK IN(a,b) OUT(ans)
SUBROUTINE dotproduct(N,a,b,ans)
  Implicit none
  integer*4 , value ::N
  real*8 :: a(N), b(N), ans
end SUBROUTINE dotproduct
end interface

Real*8 :: a(1:100), b(1:100), out(1:100), dotans
call vecadd(100,a,b,c) !No data copy needed
call dotproduct(100,a,b,dotans) !cublas call
End program p

__global__ void vecadd(int N,double *x,double *y,double *out)
{
  int tx = threadIdx.x + blockIdx.x*blockDim.x;
  if(tx<N){   out[tx]=y[tx]+x[tx]; }
  return;
}
void ddot(int size,double *x,double *y,double *out)
{
  cublasHandle_t handle = nanos_get_cublas_handle();
cublasDdot(handle,size,x,1,y,1,1,al); //using cuBLAS
Performance improvement by changing values of **NX_GPUS** env variable
CUDA accelerates ALYA by a factor of 12 to 36. OmpSs performance ranges between 40 to 60% of native CUDA. Multi GPU performance on OmpSs achieved by modifying env variable.
CUDA acceleration can significantly improve the performance computational sections.

OmpSs allows flexible programming for hybrid architectures.

OmpSs allows easy portability to multiGPUs and provides environment variables for controlling GPU features.

The performance gap between CUDA and OmpSs can be bridge by designing better scheduling policies.
Future Work

- To port mesh assembly, various pre-conditioners in iterative solvers and other direct solvers in CUDA.
- Bridge performance gap of OmpSs and CUDA using new schedulers.
- Use GPU direct on new BSC cluster with K80 and scale ALYA+GPU further.