User-Space Scheduler User Guide

Version 1.0

Daniel Welp, Tobias Beisel

November 28, 2011

1 Guideline

To utilize the *user-space scheduler (USS)* several steps have to be taken to prepare an application. These steps are listed below.

1. Apply the checkpointing scheme

   a) Create user-defined application-dependent structures for a meta checkpoint (`meta_checkpoint`) as well as for additional data (`meta_data`). They allow for interleaving and migration to other accelerators. Therefore they can contain many different technology-specific variables.

   b) Restructure the algorithm code to fit into three functions `init()`, `main()`, and `free()`. The `main()` function may be called multiple times by the USS and performs work corresponding to the current checkpoint.

2. Prepare meta scheduling information (by filling struct `meta_sched_info`)

3. Start USS entrance call (`libuss_start()`)

4. Clean up meta scheduling information

The second part of this documentation demonstrates how to implement or port an existing application in order to use the interface offered by the USS. A simple exemplary algorithm that increases vector elements is introduced, with and without USS scheduling support.

The last part provides an overview over the installation process. This includes compiling the source code using `make` and installing the shared library functionality.

2 Exemplary implementation flow

This guide shows how to port a very simple test application to use the interface provided by the USS. This application takes a vector of size `size` as input and increments each component by one. A CPU-implementation and an accelerated version for NVIDIA CUDA GPUs reflect the heterogeneous nature. The latter will increase each vector component by two instead of one to help visualizing the results. The original vector `host_A` as well as the result vector `host_C` both have to be resident in the machine's host memory before and after the computation.

2.1 Original code before porting to USS

CPU

The CPU-implementation of the function/algorithm described above is very simple and straightforward. It takes two arrays that represent in input and output vector and iterates through all elements. It is shown in Listing 2.1.

```
int myalgo_cpu(float *host_A, float *host_C, size)
{
  int i;
  for(i = 0; i < size; i++)
  {
    host_C[i] = host_A[i] + 1;
  }

  return 0;
}
```

Listing 2.1: CPU-implementation for vector increment

NVIDIA CUDA

The native implementation for NVIDIA CUDA GPUs involves preparing the device by copying the input vector to it. A CUDA kernel has to be started that looks similar to the CPU implementation (see Listing 2.2).

```
__global__ void VecAdd(float* A, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + 1;
}
```

Listing 2.2: CUDA-implementation for vector increment (kernel)

The employed general-purpose GPU performs this operation many times in parallel. The computations on the device are controlled by CPU-code that can look like the one in Listing 2.3. The CUDA runtime interface will be used to perform device operations throughout this document.

```
int myalgo_cuda(float *host_A, float *host_C, size)
{
  float *dev_A;
  float *dev_C;

  cudaError_t retruntime = cudaSetDevice(0);
  if (retruntime != cudaSuccess) {printf("SetDevice Error");}

  //allocate memory on GPU
  retruntime = cudaMalloc(&(dev_A), size);
  if (retruntime != cudaSuccess) {printf("Error_malloc");}
  retruntime = cudaMalloc(&(dev_C), size);
  if (retruntime != cudaSuccess) {printf("Error_malloc");}

  //copy original data and modified vector completely
  retruntime = cudaMemcpy(dev_A, host_A, size, cudaMemcpyHostToDevice);
  if (retruntime != cudaSuccess) {printf("Error_Memcopy");}
  retruntime = cudaMemcpy(dev_C, host_C, size, cudaMemcpyHostToDevice);
  if (retruntime != cudaSuccess) {printf("Error_Memcopy");}

  //
  //run big kernel
  //
  int threadsPerBlock, blocksPerGrid;
  threadsPerBlock = 256;
  blocksPerGrid = (size + threadsPerBlock -1) / threadsPerBlock;
  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(dev_A, dev_C);

  //copy result back
  retruntime = cudaMemcpy(host_C,dev_C, size, cudaMemcpyDeviceToHost);
  if (retruntime != cudaSuccess) {printf("Error_Copy");}

  //free device memory
  if(dev_C) cudaFree(dev_C);
  return 0;
}
```

Listing 2.3: CUDA-implementation for vector increment (host code)

2.2 Modified code after porting to USS

2.2.1 Applying the checkpointing scheme

Create user-defined meta structures

The USS demands that an application, that desires to be scheduled by it, divides its functionality/algorithm into smaller parts. After each part a checkpoint indicates the overall progress or represents the current state. A structure called `meta_checkpoint` has been introduced by the USS that encapsulates all checkpoint information.
For this exemplary application the vector increment will be done in several steps instead of all at once. Therefore the checkpoint contains the next index *curr* to increment. The user-defined struct `meta_checkpoint` is shown in Listing 2.4.

```
struct meta_checkpoint
{
  int curr;
};
```

Listing 2.4: meta checkpoint

A structure called `meta_data` (see Listing 2.5) is application-dependent and thus again programmed by the end-user. It holds the computation data size and pointers to them. A user-defined incrementation granularity constant determines how many vector components are to be increased in one iteration of the main-function, i.e., it represents the checkpointing distance.
The `float *dev_*` pointers are used by the NVIDIA CUDA-implementation only and are meaningless for the CPU-implementation.

```
struct meta_data
{
  size_t size;
  float *host_A;
  float *host_C;
  float *dev_A;
  float *dev_C;
  int start, stop;
  int inc_granularity; //how many components to increment
  int is_finished; //flag indicates completion
};
```

Listing 2.5: meta data

Restructure the code

Another requirement to use the USS interface is to define three functions called `init()`, `main()`, and `free()` for each supported technology. Whenever a registered application is chosen by the USS to run on device X, the `X_init()` function. The `X_main()` function can be executed multiple times as long as the USS permits. The `X_free()` may destroy structures or free previously allocated memory on device X.

CPU

The CPU-algorithm has to be modified in order to comply to the checkpointing scheme. Listing 2.6 shows how this can be done for the `main()` function (`init()` and `free()` can be empty for the CPU-implementation).

```c
int myalgo_cpu_main(void *md_void, void *mcp_void, int api_device_id)
{
  struct meta_data *md = (struct meta_data*) md_void;
  struct meta_checkpoint *mcp = (struct meta_checkpoint*) mcp_void;

  int i;
  for(i = mcp->curr; i<(mcp->curr + md->inc_granularity)
              && i<(md->stop); i++)
  {
    md->host_C[i] = md->host_A[i] + 1;
  }

  mcp->curr = i;

  if(i == md->stop) {md->is_finished = 1;}

  return 0;
}
```

Listing 2.6: Modified CPU-implementation for vector increment (main)

The `main()` function always requires the most recent checkpoint or it will do the same task twice. The internal loop starts at the position specified in the `meta_checkpoint` struct.

NVIDIA CUDA

Since any computation data has to be transferred to a NVIDIA CUDA-enabled GPU, the `init()` method is used to allocate device memory and copy the complete vector to it (Listing 2.7). The variable `api_device_id` is used to identify the target device.

```
int myalgo_cuda_init(void *md_void, void *mcp_void, int api_device_id)
{
  struct meta_data *md = (struct meta_data*) md_void;

  cudaError_t retruntime = cudaSetDevice(api_device_id);
  if (retruntime != cudaSuccess) {printf("SetDevice Error");}

  //get space on cuda
  retruntime = cudaMalloc(&(md->dev_A), md->size);
  if (retruntime != cudaSuccess) {printf("Error_malloc");}
  retruntime = cudaMalloc(&(md->dev_C), md->size);
  if (retruntime != cudaSuccess) {printf("Error_malloc");}

  //copy original data and modified vector completely
  retruntime = cudaMemcpy(md->dev_A, md->host_A,
                          md->size, cudaMemcpyHostToDevice);
  if (retruntime != cudaSuccess) {printf("Error_Memcopy");}
  retruntime = cudaMemcpy(md->dev_C, md->host_C,
                          md->size, cudaMemcpyHostToDevice);
  if (retruntime != cudaSuccess) {printf("Error_Memcopy");}

  return 0;
}
```

Listing 2.7: Modified CUDA-implementation for vector increment (host code init)

The meta data strucutre is used to remember the location of the allocated device memory. Each `main()` function only has to start a kernel of a size equal to the checkpointing distance. The modified CUDA kernel is shown in Listing 2.8 and takes into account the offset for the current position inside the vector.

```
__global__ void VecAdd(float* A, float* C, int offset)
{
  int i = threadIdx.x;
  C[i+offset] = A[i+offset] + 2;
}
```

Listing 2.8: Modified CUDA-implementation for vector increment (kernel)

This CUDA kernel has an additional parameter (compared to Listing 2.2) that specifies the offset. As shown in Listing 2.9, this offset equals the current position in the vector.

```
int myalgo_cuda_main(void *md_void, void *mcp_void, int api_device_id)
{
  struct meta_data *md = (struct meta_data*) md_void;
  struct meta_checkpoint *mcp = (struct meta_checkpoint*) mcp_void;

  int threadsPerBlock, blocksPerGrid, N;
  //run for md->inc_granularity or less if we are close to stop
  if(mcp->curr + md->inc_granularity < md->stop)
    N = md->inc_granularity;
  else
    N = md->stop - mcp->curr + 1;

  //run small kernel
  threadsPerBlock = 256;
  blocksPerGrid = (N + threadsPerBlock -1) / threadsPerBlock;
  VecAdd<<<blocksPerGrid, threadsPerBlock>>>
        (md->dev_A, md->dev_C, mcp->curr);

  //update checkpoint
  if(mcp->curr + md->inc_granularity < md->stop)
    mcp->curr += (N);
  else
    mcp->curr += (N-1);

  if(mcp->curr == md->stop) {md->is_finished = 1;}

  return 0;
}
```

Listing 2.9: Modified CUDA-implementation for vector increment (host code main)

The checkpoint is modified each time at the end of the `main()` function. However, this does not include any device-specific operations. Since the device is controlled by CPU-code, it is sufficient to have the current checkpoint in host memory.

Only when the application is told to free the NVIDIA GPU or finishes all work, the incremented vector is transferred back.

```c
int myalgo_cuda_free(void *md_void, void *mcp_void, int api_device_id)
{
  struct meta_data *md = (struct meta_data*) md_void;
  cudaError_t retruntime;

  //copy result back
  retruntime = cudaMemcpy(md->host_C, md->dev_C,
               md->size, cudaMemcpyDeviceToHost);
  if (retruntime != cudaSuccess) {printf("Error malloc \n"); exit(1);}

  //free device memory
  if(md->dev_C) cudaFree(md->dev_C);

  return 0;
}
```

Listing 2.10: Modified CUDA-implementation for vector increment (host code cleanup)

2.2.2 Prepare meta scheduling information

To inform the user-space scheduler about an application's different available implementations and how well it performs on them, meta scheduling information are send to the USS during the registration process. The struct `meta_sched_info` has to be filled by the end-user during runtime or at compile-time. The USS library provides a function called `libuss_fill_msi()` to assist with this task. The function declaration is shown in Listing 2.11.

```c
int libuss_fill_msi(struct meta_sched_info *msi, int type,
                int affinity, int flags,
                int (*algo_init_function)(void*, void*, int),
                int (*algo_main_function)(void*, void*, int),
                int (*algo_free_function)(void*, void*, int));
```

Listing 2.11: Modified CUDA-implementation for vector increment (host code cleanup)

Each supported technology has an integer constant that is defined in the header file `uss.h`. It is given to `libuss_fill_msi()` as the second parameter. The third parameter reflects the affinity towards this technology. A value of 10 represents the best affinity and 1 the lowest, 0 tells the scheduler that the technology is not supported by the implementation. The full usage of this function is shown in the next section.

2.2.3 Start USS entrance call

A complete version of the test application described above is shown in Listing 2.12. It recaps the introduction of the two user-defined meta structures and shows the three functions for CPU and NVIDIA CUDA GPUs.

The main part of the exemplary code begins by filling the meta scheduling information structure that tells the USS what hardware is supported and conveys the function-pointers. The first checkpoint is initialized with zero and meta data are prepared by allocating host memory. A common granularity constant called `inc_granularity` is set to 10 in this example. In general, it is reasonable to use different values for different implementations to control the checkpointing distances of the respective technology.

After all structures have been filled, the main library routine is invoked by calling `libuss_start()` rendering this thread subject to the USS. The USS selects the offered functions and executes them. This continues until the last checkpoint is reached and `libuss_start()` returns. As a last operation, the host thread should free any memory allocations.

```
//USS
#include <uss.h>
#include <other.includes>

//////////////////////////////////////////////
//
// own user-defined USS structures
//
//////////////////////////////////////////////
#define MYEXAMPLEARRAY 100

struct meta_checkpoint
{
  int curr;
};

struct meta_data
{
  size_t size;
  float *host_A;
  float *host_C;
  float *dev_A;
  float *dev_C;
  int start, stop, inc_granularity;
  int is_finished;
};

//////////////////////////////////////////////
//
// CPU implementation
//
//////////////////////////////////////////////
```

```
int myalgo_cpu_init(void *md_void, void *mcp_void, int api_device_id)
{...}

int myalgo_cpu_main(void *md_void, void *mcp_void, int api_device_id)
{...}

int myalgo_cpu_free(void *md_void, void *mcp_void, int api_device_id)
{...}

//////////////////////////////////////////////
//
// CUDA implementation
//
//////////////////////////////////////////////
__global__ void VecAdd(float* A, float* C, int offset)
{
  int i = threadIdx.x;
  C[i+offset] = A[i+offset] + 2;
}

int myalgo_cuda_init(void *md_void, void *mcp_void, int api_device_id)
{...}

int myalgo_cuda_main(void *md_void, void *mcp_void, int api_device_id)
{...}

int myalgo_cuda_free(void *md_void, void *mcp_void, int api_device_id)
{...}

//////////////////////////////////////////////
//
// MAIN (fills msi and calls libuss_start)
//
//////////////////////////////////////////////
int main()
 {
  int i, id = 0;
  int inc_granularity = 10;

/************************************\
* fill meta sched info (MSI) struct
\************************************/
  int ret = 0;
  struct meta_sched_info msi;
  memset(&msi, 0, sizeof(struct meta_sched_info));

  //insert USS_ACCEL_TYPE_CPU
  ret = libuss_fill_msi(&msi, USS_ACCEL_TYPE_CPU, 2, 0,
           &myalgo_cpu_init,
           &myalgo_cpu_main,
           &myalgo_cpu_free);
  if(ret == -1) {printf("Error"); return -1;}


  //insert USS_ACCEL_TYPE_CUDA
    ret = libuss_fill_msi(&msi, USS_ACCEL_TYPE_CUDA, 3, 0,
          &myalgo_cuda_init,
          &myalgo_cuda_main,
          &myalgo_cuda_free);
  if(ret == -1) {printf("Error"); return -1;}

/*************************************\
* fill meta checkpoint (MCP)
\*************************************/
  struct meta_checkpoint mcp;

  mcp.curr = 0;

/*************************************\
* fill meta data (MD)
\*************************************/
  struct meta_data md;

  md.host_A = (float*) malloc(MYEXAMPLE_ARRAY*sizeof(float));
  memset(md.host_A, 0, MYEXAMPLE_ARRAY*sizeof(float));
  md.host_C = (float*) malloc(MYEXAMPLE_ARRAY*sizeof(float));
  memset(md.host_A, 0, MYEXAMPLE_ARRAY*sizeof(float));

  md.size = sizeof(float)*MYEXAMPLE_ARRAY;
  md.stop = MYEXAMPLE_ARRAY - 1;
  md.inc_granularity = inc_granularity;
  md.is_finished = 0;

/*************************************\
* call libuss_start to enter
\*************************************/
  int run_on;
  int api_device_id;
  libuss_start(&msi, (void*)&md, (void*)&mcp, &(md.is_finished),
               &run_on, &api_device_id);

/*************************************\
* free meta sched info (MSI) memory
\*************************************/
  libuss_free_msi(&msi);

  return 0;
 }
```

Listing 2.12: Modified code that uses USS

3 Installation and Makefile

The enclosed recursive Makefile controls the installation process. The list below explains all available options for the top-level Makefile.

- `make all`
  This creates the most important parts (library, daemon) of USS but does not perform an installation of the library.
- `make library`
  This compiles the USS library as an shared library .so file.
- `make daemon`
  The USS daemon is created as a binary file.
- `make testapp`
  To test the USS' functionality, several basic test application are created.
- `make testappprime`
  An accelerated test application that uses NVIDIA CUDA GPUs to perform prime factorizations. (requires `nvcc`)
- `make testappmd5`
  An accelerated test application uses NVIDIA CUDA GPUs to perform MD5 cracking. (requires `nvcc`)
- `make ticks`
  Some benchmarks use time measurements and have to be adjusted by estimating the current machines ticks per second.
- `make install`
  This installs the USS environment in two steps on the target machine and required root privileges. The first step copies the header file uss.h to /usr/include/ to support an interface for applications that desire to use the USS. Secondly, the previously created shared library is installed into /lib/ and /usr/lib/ or the respective 64-bit folder.