Cell-MPI
Mastering the Cell Broadband Engine architecture through a Boost based parallel communication library

Sebastian Schaetz, Joel Falcou, Lionel Lacassagne

Digiteo Foundation, LRI - University Paris South XI, CEA LIST

May 17, 2011
Why a talk about a library for the Cell in 2011

- Heterogeneous and composable architectures are not uncommon, powerful and worth studying.
Why a talk about a library for the Cell in 2011

- Heterogeneous and composable architectures are not uncommon, powerful and worth studying.
- We present useful concepts that apply to all of them.
Why a talk about a library for the Cell in 2011

- Heterogeneous and composable architectures are not uncommon, powerful and worth studying.
- We present useful concepts that apply to all of them.
- We illustrate the lessons we learned as we used Boost libraries on a constricted platform and
Why a talk about a library for the Cell in 2011

- Heterogeneous and composable architectures are not uncommon, powerful and worth studying.
- We present useful concepts that apply to all of them.
- We illustrate the lessons we learned as we used Boost libraries on a constricted platform and
- elaborate what choices we had to make and why we made them as we created a Boost-like library for this platform.
Cell Broadband Engine - Schematic

Element Interconnect Bus

PPU
PPU Memory

SPU Memory
SPU

SPU Memory
SPU

SPU Memory
SPU

SPU Memory
SPU

SPU Memory
SPU

I/O Interface
Uplink
A similar architecture - Multi-GPU Schematic
Cell Broadband Engine - The good stuff

- Power architecture core paired with up to 8 streamlined vector co-processors: 204.8 GFlops/s (single) 102.4 GFlops/s (double)
- High data transfer bandwidth: theoretical 204.8 GB/s
- Good performance/watt (0.87 double precision GFlops/s per Watt for IBM BladeCenter QS22)

Due to these advantages, the CBE is a good fit for multimedia and vector processing applications as well as scientific computation.
Cell Broadband Engine - The bad stuff

- Distributed system on one chip, explicit communication necessary
- SPE Memory limitations
  - 256kB for code and data per SPE
  - no overflow detection
- Communication intricacies
  - packet size
  - address alignment
  - explicit DMA
- Optimization for speed
  - SIMD (assembler-like)
  - convoluted pipeline mechanism

Due to these restrictions, the complexity of programming the CBE is comparable to writing code for embedded systems.
Writing code for the CBE

- PPE and SPE entry points in separate main functions

```c
/* DMA control block information from system memory. */
mfc_get((void*)&parms, parm_ptr, (sizeof(parms)+15)&~/zero.noslashxF, tag, td, rd);
mfc_write_tag_mask(1<<tag);
mfc_read_tag_status_all(); /* Wait for DMA to complete */
```
Writing code for the CBE

- PPE and SPE entry points in separate main functions
- Compilers: ppu-gcc, spu-gcc

```c
/* DMA control block information from system memory. */
mfc_get((void*)&parms, parm_ptr, (sizeof(parms)+15)&~/zero.noslashxF, tag, td, rd);
mfc_write_tag_mask(1<<tag);
mfc_read_tag_status_all(); /* Wait for DMA to complete */
```
Writing code for the CBE

- PPE and SPE entry points in separate main functions
- Compilers: ppu-gcc, spu-gcc
- SPE object file passed to ppu-embedspu to generate library exports symbol that is accessible from PPE code
Writing code for the CBE

- PPE and SPE entry points in separate main functions
- Compilers: ppu-gcc, spu-gcc
- SPE object file passed to ppu-embedspu to generate library exports symbol that is accessible from PPE code

- PPE creates thread for each SPE and loads the symbol

    ```c
    /* DMA control block information from system memory. */
    mfc_get((void*)&parms, parm_ptr, (sizeof(parms)+15)&~/zero.noslashxF, tag, td, rd);
    mfc_write_tag_mask(1<<tag);
    mfc_read_tag_status_all(); /* Wait for DMA to complete */
    ```
Writing code for the CBE

- PPE and SPE entry points in separate main functions
- Compilers: ppu-gcc, spu-gcc
- SPE object file passed to ppu-embedspu to generate library exports symbol that is accessible from PPE code

- PPE creates thread for each SPE and loads the symbol
- Argument passed to thread is accessible in SPE through argument vector
Writing code for the CBE

- PPE and SPE entry points in separate main functions
- Compilers: ppu-gcc, spu-gcc
- SPE object file passed to ppu-embedspu to generate library
  exports symbol that is accessible from PPE code
- PPE creates thread for each SPE and loads the symbol
- Argument passed to thread is accessible in SPE through argument vector
- Usual approach: argument is pointer to structure in main memory; structure is loaded to SPE through explicit DMA call:

```c
/* DMA control block information from system memory. */
mfc_get((void*)&parms, parm_ptr, (sizeof(parms)+15)&~/zero.noslashxF, tag, td, rd);
mfc_write_tag_mask(1<<tag);
mfc_read_tag_status_all(); /* Wait for DMA to complete */
```
Writing code for the CBE

- PPE and SPE entry points in separate main functions
- Compilers: ppu-gcc, spu-gcc
- SPE object file passed to ppu-embedspu to generate library
  exports symbol that is accessible from PPE code

- PPE creates thread for each SPE and loads the symbol
- Argument passed to thread is accessible in SPE through argument vector
- Usual approach: argument is pointer to structure in main memory;
  structure is loaded to SPE through explicit DMA call:

```c
/* DMA control block information from system memory. */
mfc_get((void*)&parms, parm_ptr, (sizeof(parms)+15)&~0xF, tag, td, rd);
mfc_write_tag_mask(1<<tag);
mfc_read_tag_status_all(); /* Wait for DMA to complete */
```
"Getting started" can be tedious when developing for the CELL since compilation procedure and startup are not trivial.
Writing code for the CBE - continued

- "Getting started" can be tedious when developing for the CELL since compilation procedure and startup are not trivial

- CMake to the rescue:
Writing code for the CBE - continued

- "Getting started" can be tedious when developing for the CELL since compilation procedure and startup are not trivial

- CMake to the rescue:
  - Great tool to simplify basically any build-related steps
"Getting started" can be tedious when developing for the CELL since compilation procedure and startup are not trivial.

CMake to the rescue:
- Great tool to simplify basically any build-related steps
- Find all required libraries and binaries on the system
Writing code for the CBE - continued

- "Getting started" can be tedious when developing for the CELL since compilation procedure and startup are not trivial

- CMake to the rescue:
  - Great tool to simplify basically any build-related steps
  - Find all required libraries and binaries on the system
  - Low-level macros: ACTIVATE_PPE_COMPILER(), ACTIVATE_SPE_COMPILER()
Writing code for the CBE - continued

- "Getting started" can be tedious when developing for the CELL since compilation procedure and startup are not trivial

- CMake to the rescue:
  - Great tool to simplify basically any build-related steps
  - Find all required libraries and binaries on the system
  - Low-level macros: ACTIVATE_PPE_COMPILER(), ACTIVATE_SPE_COMPILER()
  - ADD_SPE_MODULE(target symbol file0 file1 ... fileN)
Writing code for the CBE - continued

- "Getting started" can be tedious when developing for the CELL since compilation procedure and startup are not trivial

- CMake to the rescue:
  - Great tool to simplify basically any build-related steps
  - Find all required libraries and binaries on the system
  - Low-level macros: ACTIVATE_PPE_COMPILER(), ACTIVATE_SPE_COMPILER()
  - ADD_SPE_MODULE(target symbol file0 file1 ... fileN)

- C++ and Boost to the rescue:
Writing code for the CBE - continued

- "Getting started" can be tedious when developing for the CELL since compilation procedure and startup are not trivial.

- CMake to the rescue:
  - Great tool to simplify basically any build-related steps
  - Find all required libraries and binaries on the system
  - Low-level macros: ACTIVATE_PPE_COMPILER(), ACTIVATE_SPE_COMPILER()
  - ADD_SPE_MODULE(target symbol file0 file1 ... fileN)

- C++ and Boost to the rescue:
  - Wrap recurring boilerplate code in clearly laid out functions and classes
Writing code for the CBE - continued

- "Getting started" can be tedious when developing for the CELL since compilation procedure and startup are not trivial.

- CMake to the rescue:
  - Great tool to simplify basically any build-related steps
  - Find all required libraries and binaries on the system
  - Low-level macros: `ACTIVATE_PPE_COMPILER()`, `ACTIVATE_SPE_COMPILER()`
  - `ADD_SPE_MODULE(target symbol file0 file1 ... fileN)`

- C++ and Boost to the rescue:
  - Wrap recurring boilerplate code in clearly laid out functions and classes
  - A kernel function should be declared and behave like a free function
Cell-MPI Bootstrapping

- Launching a kernel function passing a data structure

```c
struct mydatastruct {
    int x;
    int y;
    int z;
};
```
Cell-MPI Bootstrapping

- Launching a kernel function passing a data structure
  ```c
  struct mydatastruct { int x; int y; int z; }
  ```

- Kernel is defined with:
  ```c
  BEGIN_CELL_KERNEL()
  {
    mydatastruct * ptr;
    SPE_Custom(ptr);
    RETURN((ptr->x + ptr->y) * ptr->z);
  }
  END_CELL_KERNEL()
  ```
Cell-MPI Bootstrapping

- Launching a kernel function passing a data structure
  ```
  struct mydatastruct {
    int x;
    int y;
    int z;
  };
  ```

- Kernel is defined with:
  ```
  BEGIN_CELL_KERNEL()
  {
    mydatastruct * ptr;
    SPE_Custom(ptr);
    RETURN((ptr->x + ptr->y) * ptr->z);
  }
  END_CELL_KERNEL()
  ```

- In PPE code the kernel is registered with
  ```
  PPE_REGISTER_KERNEL(kernel);
  ```
Cell-MPI Bootstrapping

- Launching a kernel function passing a data structure
  ```c
  struct mydatastruct { int x; int y; int z; }
  ```

- Kernel is defined with:
  ```c
  BEGIN_CELL_KERNEL()
  {
    mydatastruct * ptr;
    SPE_Custom(ptr);
    RETURN((ptr->x + ptr->y) * ptr->z);
  }
  END_CELL_KERNEL()
  ```

- In PPE code the kernel is registered with
  ```c
  PPE_REGISTER_KERNEL(kernel);
  ```

- The runtime is initialized with `PPE_Init();`
Cell-MPI Bootstrapping - continued

- The kernel is then called asynchronously:

```c
1    mydatastruct mydata(1, 5, 7);
2    PPE_Run(kernel, mydata);
```
Cell-MPI Bootstrapping - continued

- The kernel is then called asynchronously:

```c
1    mydatastruct mydata(1, 5, 7);
2    PPE_Run(kernel, mydata);
```

- The PPE can wait for kernel completion: `PPE.Sync();`
Cell-MPI Bootstrapping - continued

- The kernel is then called asynchronously:

  ```
  1 int mydatastruct mydata(1, 5, 7);
  2 PPE_Run(kernel, mydata);
  ```

- The PPE can wait for kernel completion: `PPE_Sync();`

- and access the kernels return value:

  ```
  1 int returnvalues[CBE_MPI_NUM_SPE];
  2 PPE_Return(&returnvalues[0]);
  ```
Cell-MPI Bootstrapping - continued

- The kernel is then called asynchronously:
  ```c
  1  mydatastruct mydata(1, 5, 7);
  2  PPE_Run(kernel, mydata);
  ```

- The PPE can wait for kernel completion: `PPE_Sync();`

- and access the kernels return value:
  ```c
  1  int returnvalues[CBE_MPI_NUM_SPE];
  2  PPE_Return(&returnvalues[0]);
  ```

- The runtime is finalized with `PPE_Finalize();`
Cell-MPI Bootstrapping Mechanism

PPE_Init
- allocate thread- and control block

PPE_Run
- load SPE program
- init control block incl. user data
- create and start threads

PPE_Sync
- wait for return value, check for error

PPE_Finalize
- send message to shutdown kernel

SPE_Init
- load control block
- load user data

SPE_Finalize
- send return value

Synchronization after synchronization a kernel can be started again

fork

join

kernel loop

run

shutdown

kernel user code
A kernel can be declared in both PPE and SPE code with:

```c
SPE_FUNCTION(kernel, kernel, (int x) (int y) (int z));
```
Cell-MPI Bootstrapping - Boostified

- A kernel can be declared in both PPE and SPE code with:

```c
SPE_FUNCTION(kernel_, kernel, (int x) (int y) (int z));
```

- and implemented as a free function in SPE code

```c
int kernel(int x, int y, int z) {
    return (x+y) * z;
}
```
Cell-MPI Bootstrapping - Boostified

- A kernel can be declared in both PPE and SPE code with:

```c
DE_FUNCTION(kernel_, kernel, (int x) (int y) (int z));
```

- and implemented as a free function in SPE code

```c
int kernel(int x, int y, int z)
{
    return (x+y) * z;
}
```

- It can then be called as a free function from PPE code:

```c
int * returnvalues = kernel(2, 5, 7);
```
Cell-MPI Bootstrapping - Boostified

- A kernel can be declared in both PPE and SPE code with:

```c
SPE_FUNCTION(kernel_, kernel, (int x) (int y) (int z) );
```

- and implemented as a free function in SPE code

```c
int kernel(int x, int y, int z)
{
    return (x+y) * z;
}
```

- It can then be called as a free function from PPE code:

```c
int * returnvalues = kernel(2, 5, 7);
```

- or asynchronously:

```c
kernel_async(2, 5, 7);
PPE_Sync();
```
So we do C++ but...

The architecture forces some restrictions especially on the SPE part of the library:

- Compilation without run-time type information
So we do C++ but...

The architecture forces some restrictions especially on the SPE part of the library:

- Compilation without run-time type information
- No dynamic memory allocation for predictable footprint
So we do C++ but...

The architecture forces some restrictions especially on the SPE part of the library:

- Compilation without run-time type information
- No dynamic memory allocation for predictable footprint
- Custom, lightweight STL compatible allocators
So we do C++ but...

The architecture forces some restrictions especially on the SPE part of the library:

- Compilation without run-time type information
- No dynamic memory allocation for predictable footprint
- Custom, lightweight STL compatible allocators
- Exception handling deactivated
Exception emulation

Due to architecture limitations we emulate exceptions:

- An exception stops the kernel and notifies the PPE

```c
#define THROW(errno) {spe_errno = errno; SPE_Finalize(-1); exit(0);}
```

```c
struct spe_error_bundle
{
    std::vector<spe_error_data> exception_info;
};
```

```c
typedef boost::error_info<struct tag_spe_error_info_bundle,
    spe_error_bundle> spe_error_info_bundle;
```
Exception emulation

Due to architecture limitations we emulate exceptions:

- An exception stops the kernel and notifies the PPE
- Only an error code is "thrown":

```c
#define THROW(errno) {spe_errno = errno; SPE_Finalize(-1); exit(0);}
```
Exception emulation

Due to architecture limitations we emulate exceptions:

- An exception stops the kernel and notifies the PPE
- Only an error code is "thrown":

```cpp
#define THROW(errno) {spe_errno = errno; SPE_Finalize(-1); exit(0);}  
```

- The PPE translates the error code into real exceptions:

```cpp
struct spe_error_bundle
{
    std::vector<spe_error_data> exception_info;
};  
```
Exception emulation

Due to architecture limitations we emulate exceptions:

- An exception stops the kernel and notifies the PPE
- Only an error code is "thrown":

  ```c
  #define THROW(errno) {spe_errno = errno; SPE_Finalize(-1); exit(0);}
  ```

- The PPE translates the error code into real exceptions:

  ```c
  struct spe_error_bundle
  {
    std::vector<spe_error_data> exception_info;
  };
  
  typedef boost::error_info<struct tag_spe_error_info_bundle,
                             spe_error_bundle> spe_error_info_bundle;
  ```
Exception emulation

Due to architecture limitations we emulate exceptions:

- An exception stops the kernel and notifies the PPE
- Only an error code is "thrown"

```c
#define THROW(errno) {spe_errno = errno; SPE_Finalize(-1); exit(0);}
```

- The PPE translates the error code into real exceptions:

```c
struct spe_error_bundle
{
  std::vector<spe_error_data> exception_info;
};
```

```c
typedef boost::error_info<struct tag spe_error_info_bundle,
  spe_error_bundle> spe_error_info_bundle;
```

```c
struct spe_runtime_exception : virtual boost::exception {};
```
Exception emulation - continued

- If desired SPE exceptions can interrupt PPE execution
Exception emulation - continued

- If desired SPE exceptions can interrupt PPE execution
- To define errors we use the same trick as in boostified bootstrapping:

```
1 ERROR(MPI_TAG_MISMATCH, 7, "Send receive tag mismatch")
2 ERROR(BOOST_FUNCTION_BAD_CALL, 12, "Bad boost function call")
3 ERROR(BAD_ALLOC, 14, "bad alloc")
```

Compiled with the SPE compiler (
```c
#ifdef _SPE_
```
```c
enum { MPI_TAG_MISMATCH = 7, BOOST_FUNCTION_BAD_CALL = 12,
      BAD_ALLOC = 14 };
```
```c
```
And with the PPE compiler generates a vector of objects:
```c
struct spe_error_struct
{ int id; const char * symbol; const char * message; 
```
Exception emulation - continued

- If desired SPE exceptions can interrupt PPE execution
- To define errors we use the same trick as in boostified bootstrapping:

1. `ERROR(MPI_TAG_MISMATCH, 7, "Send receive tag mismatch")`
2. `ERROR(BOOST_FUNCTION_BAD_CALL, 12, "Bad boost function call")`
3. `ERROR(BAD_ALLOC, 14, "bad alloc")`
Exception emulation - continued

- If desired SPE exceptions can interrupt PPE execution
- To define errors we use the same trick as in boostified bootstrapping:

```
1   ERROR(MPI_TAG_MISMATCH, 7, "Send receive tag mismatch")
2   ERROR(BOOST_FUNCTION_BAD_CALL, 12, "Bad boost function call")
3   ERROR(BAD_ALLOC, 14, "bad alloc")
```

- Compiled with the SPE compiler (#ifdef _SPE_) generates:

```
1   enum { MPI_TAG_MISMATCH = 7, BOOST_FUNCTION_BAD_CALL = 12,
2          BAD_ALLOC = 14 };
```
Exception emulation - continued

- If desired SPE exceptions can interrupt PPE execution
- To define errors we use the same trick as in boostified bootstrapping:

```cpp
1. ERROR(MPI_TAG_MISMATCH, 7, "Send receive tag mismatch")
2. ERROR(BOOST_FUNCTION_BAD_CALL, 12, "Bad boost function call")
3. ERROR(BAD_ALLOC, 14, "bad alloc")
```

- Compiled with the SPE compiler `(#ifdef _SPE_)` generates:

```cpp
enum { MPI_TAG_MISMATCH = 7, BOOST_FUNCTION_BAD_CALL = 12, BAD_ALLOC = 14 };
```

- And with the PPE compiler generates a vector of objects:

```cpp
struct spe_error_struct
{
    int id; const char * symbol; const char * message;
};
```
Unit Testing

Boostrap Test is great but builds don’t SPEs:

libboost_unit_test_framework.so.1.45./zero.noslash: 998kB

First idea:

boost/detail/lightweight_test.hpp misses a lot of the Boost Test goodness

Enter SPE-Unit:

Compromise between lightweight and feature-complete

Designed after Boost Test
Unit Testing

- Boost.Test is great but builds don’t fit SPEs:
  libboost_unit_test_framework.so.1.45.0: 998kB
Unit Testing

- Boost.Test is great but builds don’t fit SPEs:
  libboost_unit_test_framework.so.1.45.0: 998kB
- First idea: boost/detail/lightweight_test.hpp
Unit Testing

- Boost.Test is great but builds don’t fit SPEs:
  libboost_unit_test_framework.so.1.45.0: 998kB
- First idea: boost/detail/lightweight_test.hpp
  misses a lot of the Boost.Test goodness
Unit Testing

- Boost.Test is great but builds don’t fit SPEs:
  - libboost_unit_test_framework.so.1.45.0: 998kB
- First idea: boost/detail/lightweight_test.hpp misses a lot of the Boost.Test goodness

Enter SPE-Unit:

- Compromise between lightweight and feature-complete
Unit Testing

- **Boost.Test** is great but builds don’t fit SPEs:
  - `libboost_unit_test_framework.so.1.45.0`: 998kB

- **First idea**: `boost/detail/lightweight_test.hpp`
  misses a lot of the Boost.Test goodness

**Enter SPE-Unit:**

- Compromise between lightweight and feature-complete
- Designed after Boost.Test
Unit Testing - SPE Unit Features

- Only one test suite is available: `CBE_MPI_SPEUNIT_AUTO_TEST_SUITE();`
Unit Testing - SPE Unit Features

- Only one test suite is available: `CBE_MPI_SPEUNIT_AUTO_TEST_SUITE()`;
- Tests are started explicitly:

```c
1 uint32_t result = CBE_MPI_SPEUNIT_RUN_TEST_SUITE();
2 SET_RETURN_VALUE(result);
```
Unit Testing - SPE Unit Features

- Only one test suite is available: CBE_MPI_SPEUNIT_AUTO_TEST_SUITE();
- Tests are started explicitly:

  1. uint32_t result = CBE_MPI_SPEUNIT_RUN_TEST_SUITE();
  2. SET_RETURN_VALUE(result);

- The powerful AUTO_TEST_CASE_TEMPLATE(testname, T, typelist)
  and a normal template TEST_CASE_TEMPLATE(testname) are included
Unit Testing - SPE Unit Features

- Only one test suite is available: `CBE_MPI_SPEUNIT_AUTO_TEST_SUITE()`;
- Tests are started explicitly:
  ```c
  1  uint32_t result = CBE_MPI_SPEUNIT_RUN_TEST_SUITE();
  2  SET_RETURN_VALUE(result);
  ```
- The powerful `AUTO_TEST_CASE_TEMPLATE(testname, T, typelist)` and a normal template `TEST_CASE_TEMPLATE(testname)` are included
- Different test tool levels are supported: `WARN_*`, `CHECK_*`, `REQUIRE_*`
Unit Testing - SPE Unit Features

- Only one test suite is available: `CBE_MPI_SPEUNIT_AUTO_TEST_SUITE();`
- Tests are started explicitly:
  
  ```
  uint32_t result = CBE_MPI_SPEUNIT_RUN_TEST_SUITE();
  SET_RETURN_VALUE(result);
  ```

- The powerful `AUTO_TEST_CASE_TEMPLATE(testname, T, typelist)` and a normal template `TEST_CASE_TEMPLATE(testname)` are included

- Different test tool levels are supported: `WARN_*`, `CHECK_*`, `REQUIRE_*`

- Strings can be disabled to reduce overhead (silent mode)
Unit Testing - SPE Unit Features

- Only one test suite is available: `CBE_MPI_SPEUNIT_AUTO_TEST_SUITE()`;
- Tests are started explicitly:

```c
1 uint32_t result = CBE_MPI_SPEUNIT_RUN_TEST_SUITE();
2 SET_RETURN_VALUE(result);
```

- The powerful `AUTO_TEST_CASE_TEMPLATE(testname, T, typelist)` and a normal template `TEST_CASE_TEMPLATE(testname)` are included
- Different test tool levels are supported: `WARN_*`, `CHECK_*`, `REQUIRE_*`
- Strings can be disabled to reduce overhead (silent mode)
- Emulated SPE exceptions can be validated with test tools like `CBE_MPI_REQUIRE_THROW`
typedef boost::mpl::vector_c<int,1,2,4,8,16> aligned_alloc_alignments;

CBE_MPI_SPEUNIT_AUTO_TEST_SUITE();

CBE_MPI_SPEUNIT_AUTO_TEST_CASE_TEMPLATE( aligned_malloc_free_test, T,
    aligned_alloc_alignments )
{
    aligned_ptr<void,T::value> ptr = aligned_malloc<T::value>(T::value);
    CBE_MPI_SPEUNIT_REQUIRE_EQUAL(is_aligned<T::value>(ptr.get()),true);
    cbe_mpi::aligned_free(ptr);
    CBE_MPI_SPEUNIT_REQUIRE_EQUAL(ptr.get(),((void*)(/zero.noslash)));
}

int kernel(void)
{
    uint32_t result = CBE_MPI_SPEUNIT_RUN_TEST_SUITE();
    SET_RETURN_VALUE(result);
}
Data Transfer - Single Buffer

```cpp
ii = in.get();
oo = out.get();

for(int i=0; i<iterations; i++) {
    spe_ppe_get_c(in.get(), cd->inbuf1+(SPE_Rank()+i*SPE_Size())*slicesize*sizeof(float),
                  slicesize_padded*sizeof(float));

    harris_simd(ii, oo, cd->slice_dimx, cd->slice_dimy, 0, PADY, buf1.get(), buf2.get(), buf3.get());

    spe_ppe_put_c(cd->outbuf1+(SPE_Rank()+i*SPE_Size())*slicesize*sizeof(float) +
                  (cd->slice_dimx*PADY)*sizeof(float), oo, slicesize*sizeof(float));
}
```
Data Transfer - Single Buffer

```c
ii = in.get();
oo = out.get();

for(int i=0; i<iterations; i++) {
  spe_ppe_get_c(in.get(), cd->inbuf1+(SPE_Rank()+i*SPE_Size())*slicesize*sizeof(float),
                slicesize_padded*sizeof(float));

  harris_simd(ii, oo, cd->slice_dimx, cd->slice_dimy, 0, PADY, buf1.get(), buf2.get(), buf3.get());

  spe_ppe_put_c(cd->outbuf1+(SPE_Rank()+i*SPE_Size())*slicesize*sizeof(float) +
                (cd->slice_dimx*PADY)*sizeof(float), oo, slicesize*sizeof(float));
}
```
Data Transfer
Data Transfer

- Load 1
- Compute 1
- Store 1
- Load 2
- Compute 2
- Store 2
- Load 3
- Store 2
- Compute 3
- Load 4
- Store 3
- Compute 4
- Store 4
- Lead-in
- Optimal overlap
- Lead-out
Data Transfer - Double Buffering

```c
spe_ppe_get_async_c(in1.get(), cd->inbuf1+SPE_Rank()*slicesize*sizeof(float), slicesize_padded*sizeof(float), 9);

for(int i=0; i<iterations; i++) {
    if(i%2 == 0) {
        spe_ppe_get_async_c(in2.get(), cd->inbuf1+(SPE_Rank()+(i+1)*SPE_Size())*slicesize*sizeof(float),
                          slicesize_padded*sizeof(float), 10);
        dma_synchronize_c(9); dma_synchronize_c(11);
        ii = in1.get(); oo = out1.get();
    } else {
        spe_ppe_get_async_c(in1.get(), cd->inbuf1+(SPE_Rank()+(i+1)*SPE_Size())*slicesize*sizeof(float),
                          slicesize_padded*sizeof(float), 9);
        dma_synchronize_c(10); dma_synchronize_c(12);
        ii = in2.get(); oo = out2.get();
    }
}

harris_simd(ii, oo, cd->slice_dimx, cd->slice_dimy, 0, PADY, buf1.get(), buf2.get(), buf3.get());

if(i%2 == 0) {
    spe_ppe_put_async_c(cd->outbuf1+(SPE_Rank()+i*SPE_Size())*slicesize*sizeof(float) +
                        (cd->slice_dimx*PADY)*sizeof(float), out1.get(), slicesize*sizeof(float), 11);
} else {
    spe_ppe_put_async_c(cd->outbuf1+(SPE_Rank()+i*SPE_Size())*slicesize*sizeof(float) +
                        (cd->slice_dimx*PADY)*sizeof(float), out2.get(), slicesize*sizeof(float), 12);
}
}

spe_ppe_put_c(cd->outbuf1 + (SPE_Rank())+((iterations-1)*SPE_Size())) * slicesize*sizeof(float) +
               (cd->slice_dimx*PADY)*sizeof(float), oo, slicesize*sizeof(float));
```

23 of 49
Data Transfer - Double Buffering

```c
spe_ppe_get_async_c(in1.get(), cd->inbuf1+SPE_Rank()*slicesize*sizeof(float), slicesize_padded*sizeof(float), 9);

for(int i=0; i<iterations; i++) {
    if(i%2 == 0) {
        spe_ppe_get_async_c(in2.get(), cd->inbuf1+(SPE_Rank()+(i+1)*SPE_Size())*slicesize*sizeof(float),
            slicesize_padded*sizeof(float), 10);
        dma_synchronize_c(9); dma_synchronize_c(11);
        ii = in1.get(); oo = out1.get();
    } else {
        spe_ppe_get_async_c(in1.get(), cd->inbuf1+(SPE_Rank()+(i+1)*SPE_Size())*slicesize*sizeof(float),
            slicesize_padded*sizeof(float), 9);
        dma_synchronize_c(10); dma_synchronize_c(12);
        ii = in2.get(); oo = out2.get();
    }
}

harris_simd(ii, oo, cd->slice_dimx, cd->slice_dimy, 0, PADY, buf1.get(), buf2.get(), buf3.get());

if(i%2 == 0) {
    spe_ppe_put_async_c(cd->outbuf1+(SPE_Rank()+i*SPE_Size())*slicesize*sizeof(float) +
            (cd->slice_dimx*PADY)*sizeof(float), out1.get(), slicesize*sizeof(float), 11);
} else {
    spe_ppe_put_async_c(cd->outbuf1+(SPE_Rank()+i*SPE_Size())*slicesize*sizeof(float) +
            (cd->slice_dimx*PADY)*sizeof(float), out2.get(), slicesize*sizeof(float), 12);
}

spe_ppe_put_c(cd->outbuf1 + (SPE_Rank()+(iterations-1)*SPE_Size()) * slicesize*sizeof(float) +
            (cd->slice_dimx*PADY)*sizeof(float), oo, slicesize*sizeof(float));
```

Double Buffering - Operations - Input Segment
Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)
Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)
- Start loading next segment
Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)

- Start loading next segment

- Wait for segment to be ready for computation
Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)
- Start loading next segment
- Wait for segment to be ready for computation
- Signal that computation on current segment is finished
Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)
  \[ \text{operator} = () \]
- Start loading next segment
- Wait for segment to be ready for computation
- Signal that computation on current segment is finished
Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)
  \[ \text{operator } =() \]
- Start loading next segment
  \[ \text{operator } +\!(\text{int}) \]
- Wait for segment to be ready for computation
- Signal that computation on current segment is finished
Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)
  \texttt{operator \texttt{=}}()

- Start loading next segment
  \texttt{operator \texttt{++(int)}}

- Wait for segment to be ready for computation
  \texttt{operator \texttt{*()}}

- Signal that computation on current segment is finished
Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)
  operator =()

- Start loading next segment
  operator ++(int)

- Wait for segment to be ready for computation
  operator *()

- Signal that computation on current segment is finished
  operator ++(int)
Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)
  
  \texttt{operator =()}

- Start loading next segment
  
  \texttt{operator ++(int)}

- Wait for segment to be ready for computation
  
  \texttt{operator *()}

- Signal that computation on current segment is finished
  
  \texttt{operator ++(int)}

- Check if end of data is reached
Double Buffering - Operations - Input Segment

- Start loading first segment (lead-in)
  `operator =()`

- Start loading next segment
  `operator ++(int)`

- Wait for segment to be ready for computation
  `operator *()`

- Signal that computation on current segment is finished
  `operator ++(int)`

- Check if end of data is reached
  `operator ==()`
Double Buffered Segmented Input Iterator

template<typename T> struct remote_segmented_input_iterator
{
  // allocate required buffers
  remote_segmented_input_iterator(...) {} 

  // start loading first buffer
  void operator= (const addr64 & base_address_) {}

  // wait for current segment to arrive and return pointer to it
  T* operator *() {}

  // start loading new data and increment current segment
  inline void operator++(int) {}

  // check if iterator has reached a position
  bool operator ===(const addr64 & b) const {}
};
Double Buffered Segmented Iterator Example

```c
remote_segmented_input_iterator<float> it(depth, ssize, slicer(ssize));
remote_segmented_output_iterator<float> ot(depth, ssize, slicer(ssize));

for(it = input, ot = output; /* lead-in */
    it!=input+overall_size; /* check end */
    it++, ot++) // load next, store current
{
    float * in = *it; float * out = *ot; // synchronize
    harris simd(in, out, cd->slice_dimx, cd->slice_dimy, 0, PADY, buf1.get(), buf2.get(), buf3.get());
}
```
Double Buffered Segmented Iterator - Slicer
Double Buffered Segmented Iterator - Slicer
Double Buffered Segmented Iterator - Slicer
Double Buffered Segmented Iterator - Slicer
Double Buffered Segmented Iterator - Slicer

SPE0

SPE1

SPE2

SPE3
Double Buffered Segmented Iterator - Slicer
Double Buffered Segmented Iterator - Slicer
Double Buffered Segmented Iterator - Slicer

![Diagram of Double Buffered Segmented Iterator - Slicer]
Multi-Buffered Segmented Iterator - Features

- remote_vector<T> for more expressive code:

```c++
// PPE:
std::vector<float> v(1024*1024); kernel(v);
// SPE:
kernel(remote_vector<float> v) {
    remote_segmented_input_iterator<T> it(depth, ssize, slicer(ssize));
    for(it = v.begin(); it!=v.end(); it++) {
        float * in = *it;
        /* computation */
    }
}
```
Multi-Buffered Segmented Iterator - Features

- remote_vector<T> for more expressive code:

```cpp
// PPE:
std::vector<float> v(1024*1024); kernel(v);

// SPE:
kernel(remote_vector<float> v) {
    remote_segmented_input_iterator<T> it(depth, ssize, slicer(ssize));
    for(it = v.begin(); it!=v.end(); it++) {
        float * in = *it;
        /* computation */
    }
}
```

- Read, write- and read-write Iterators with minimum buffer depth of 3
Multi-Buffered Segmented Iterator - Features

- remote_vector<T> for more expressive code:

```cpp
// PPE:
std::vector<float> v(1024*1024); kernel(v);

// SPE:
kernel(remote_vector<float> v) {
    remote_segmented_input_iterator<T> it(depth, ssize, slicer(ssize));
    for(it = v.begin(); it!=v.end(); it++) {
        float * in = *it;
        /* computation */
    }
}
```

- Read, write- and read-write Iterators with minimum buffer depth of 3
- Various slicers
2D Multi-Buffered Segmented Iterator

- Native 2D data transfer support through DMA lists
2D Multi-Buffered Segmented Iterator

- Native 2D data transfer support through DMA lists
- Difference to regular iterator:

  - Slice size is 2D
  - Supports remote_vector_2D
  - Slicer takes 2D arguments:
    
    ```c
    slicer_2D(size_2d_t vector_dim, size_2d_t slice_dim);
    ```
2D Multi-Buffered Segmented Iterator

- Native 2D data transfer support through DMA lists
- Difference to regular iterator:
  - Slice size is 2D
2D Multi-Buffered Segmented Iterator

- Native 2D data transfer support through DMA lists
- Difference to regular iterator:
  - Slice size is 2D
  - Supports remote_vector_2D
2D Multi-Buffered Segmented Iterator

- Native 2D data transfer support through DMA lists
- Difference to regular iterator:
  - Slice size is 2D
  - Supports remote_vector_2D
  - Slicer takes 2D arguments:
    ```c
    slicer_2D(size_2d_t vector_dim, size_2d_t slice_dim);
    ```
2D Multi-Buffered Segmented Iterator

- Native 2D data transfer support through DMA lists
- Difference to regular iterator:
  - Slice size is 2D
  - Supports remote_vector_2D
  - Slicer takes 2D arguments:
    slicer_2D(size_2d_t vector_dim, size_2d_t slice_dim);
- Ideal for image processing:
2D Multi-Buffered Segmented Iterator

- Native 2D data transfer support through DMA lists
- Difference to regular iterator:
  - Slice size is 2D
  - Supports `remote_vector_2D`
  - Slicer takes 2D arguments:
    `slicer_2D(size_2d_t vector_dim, size_2d_t slice_dim);`
- Ideal for image processing:
2D Multi-Buffered Segmented Iterator

- Native 2D data transfer support through DMA lists
- Difference to regular iterator:
  - Slice size is 2D
  - Supports `remote_vector_2D`
  - Slicer takes 2D arguments:
    ```
    slicer_2D(size_2d_t vector_dim, size_2d_t slice_dim);
    ```
- Ideal for image processing:
High-Level Inter-SPE Communication: MPI

- Interprocess communication by message passing, SPEs send and receive messages
- API specification, used in high performance computing

**Features:**
- Virtual topology of processes
- Synchronization
- Point to point communication
- Collective communication
High-Level Inter-SPE Communication: MPI

- Interprocess communication by message passing, SPEs send and receive message
High-Level Inter-SPE Communication: MPI

- Interprocess communication by message passing, SPEs send and receive message
- API specification, used in high performance computing
High-Level Inter-SPE Communication: MPI

- Interprocess communication by message passing, SPEs send and receive message
- API specification, used in high performance computing

Features:
- Virtual topology of processes
High-Level Inter-SPE Communication: MPI

- Interprocess communication by message passing, SPEs send and receive message
- API specification, used in high performance computing

Features:
- Virtual topology of processes
- Synchronization
High-Level Inter-SPE Communication: MPI

- Interprocess communication by message passing, SPEs send and receive message
- API specification, used in high performance computing

Features:
- Virtual topology of processes
- Synchronization
- Point to point communication
High-Level Inter-SPE Communication: MPI

- Interprocess communication by message passing, SPEs send and receive message
- API specification, used in high performance computing

Features:
- Virtual topology of processes
- Synchronization
- Point to point communication
- Collective communication
MPI Collectives - Broadcast

\[ \begin{array}{cccc}
V_1 & V_2 & V_3 & V_4 \\
P_1 & A_1 & \square & \square & \square \\
P_2 & \square & \square & \square & \square \\
P_3 & \square & \square & \square & \square \\
P_4 & \square & \square & \square & \square \\
\end{array} \quad \text{broadcast} \quad \begin{array}{cccc}
W_1 & W_2 & W_3 & W_4 \\
P_1 & A_1 & \square & \square & \square \\
P_2 & \square & \square & \square & \square \\
P_3 & \square & \square & \square & \square \\
P_4 & A_1 & \square & \square & \square \\
\end{array} \]
MPI Collectives - Scatter and Gather

\[
\begin{array}{cccc}
V_1 & V_2 & V_3 & V_4 \\
\hline
P_1 & A_1 & A_2 & A_3 & A_4 \\
P_2 & & & & \\
P_3 & & & & \\
P_4 & & & & \\
\end{array}
\]

\[
\begin{array}{cccc}
W_1 & W_2 & W_3 & W_4 \\
\hline
P_1 & A_1 & & & \\
P_2 & A_2 & & & \\
P_3 & A_3 & & & \\
P_4 & A_4 & & & \\
\end{array}
\]

\[
\begin{array}{cccc}
V_1 & V_2 & V_3 & V_4 \\
\hline
P_1 & A_1 & & & \\
P_2 & B_1 & & & \\
P_3 & C_1 & & & \\
P_4 & D_1 & & & \\
\end{array}
\]

\[
\begin{array}{cccc}
W_1 & W_2 & W_3 & W_4 \\
\hline
P_1 & A_1 & B_1 & C_1 & D_1 \\
P_2 & & & & \\
P_3 & & & & \\
P_4 & & & & \\
\end{array}
\]

scatter

gather
MPI Collectives - Reduce and All to All

\[ \begin{array}{cccc}
V_1 & V_2 & V_3 & V_4 \\
P_1 & A_1 & & \\
P_2 & B_1 & & \\
P_3 & C_1 & & \\
P_4 & D_1 & & \\
\end{array} \quad \quad \begin{array}{cccc}
W_1 & W_2 & W_3 & W_4 \\
P_1 & X & & \\
P_2 & & & \\
P_3 & & & \\
P_4 & & & \\
\end{array} \quad \text{reduce} \quad \begin{array}{cccc}
\end{array} \]

\[ X = Op(A_1, Op(B_1, Op(C_1, D_1))) \]

\[ \begin{array}{cccc}
V_1 & V_2 & V_3 & V_4 \\
P_1 & A_1 & A_2 & A_3 & A_4 \\
P_2 & B_1 & B_2 & B_3 & B_4 \\
P_3 & C_1 & C_2 & C_3 & C_4 \\
P_4 & D_1 & D_2 & D_3 & D_4 \\
\end{array} \quad \quad \begin{array}{cccc}
W_1 & W_2 & W_3 & W_4 \\
P_1 & A_1 & B_1 & C_1 & D_1 \\
P_2 & A_2 & B_2 & C_2 & D_2 \\
P_3 & A_3 & B_3 & C_3 & D_3 \\
P_4 & A_4 & B_4 & C_4 & D_4 \\
\end{array} \quad \text{alltoall} \quad \begin{array}{cccc}
\end{array} \]
MPI Interface - Example

```c
communicator world;

if (world.rank() == 0) {
    char s1[] = "Hello";
    world.send(1, 0, s1, sizeof(s1));
    char s2[6];
    world.recv(1, 1, s2, sizeof(s2));
}
else if (world.rank() == 1) {
    char s1[6];
    world.recv(0, 0, s1, sizeof(s1));
    char s2[] = "world";
    world.send(0, 1, s2, sizeof(s2));
}

// Hello world from SPE 0, Hello world from SPE 1
```
class communicator
{
    void barrier();

    template <typename T> void send(int dst, int tag, const T& value);
    template <typename T> void send(int dst, int tag, const T* values, int n);
    template <typename T> request isend(int dst, int tag, const T& value);

    ... 

    template <typename T> status recv(int source, int tag, T& value);
    template <typename T> status recv(int source, int tag, T* values, int n);
    template <typename T> request irecv(int source, int tag, T& value);

    ... 

    communicator include(uint16_t first, uint16_t last);
    communicator exclude(uint16_t first, uint16_t last);
    friend bool operator==(const communicator& c1, const communicator& c2);
};
MPI Interface - Request and Status

1 // represents current request
class request
{
  request() {};
  status wait();
  boost::optional<status> test();
};

// represents status of a request
class status
{
  int32_t source() const;
  int32_t tag() const;
  int32_t error() const;
};
MPI Interface - Collectives Interface

1. template<typename T, typename Op>
2. void reduce(const communicator & comm, const T & in,
   T & out, Op op, int root);

3. template<typename T, typename Op>
4. void reduce(const communicator & comm, const T & in,
   Op op, int root);

5. template<typename T, typename Op>
6. void reduce(const communicator & comm, const T * in,
   int n, T * out, Op op, int root);

7. template<typename T, typename Op>
8. void reduce(const communicator & comm, const T * in,
   int n, Op op, int root);
### MPI Header

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>communicator identifier</td>
</tr>
<tr>
<td>8</td>
<td>tag</td>
</tr>
<tr>
<td>16</td>
<td>datatype</td>
</tr>
<tr>
<td>16</td>
<td>datasize</td>
</tr>
<tr>
<td>24</td>
<td>controlflags</td>
</tr>
<tr>
<td>24</td>
<td>opflags</td>
</tr>
<tr>
<td>24</td>
<td>prependsize</td>
</tr>
<tr>
<td>32</td>
<td>appendsize</td>
</tr>
<tr>
<td>32</td>
<td>address</td>
</tr>
</tbody>
</table>
MPI Protocol

SEND

- preprocess data
- prepare header

received header?

- yes
  - send header
  - SYN

- no
  - received header?

- yes
  - header match?

- yes
  - put data async
  - send ACK async
  - FINISHED

- no
  - reset SYN
  - DMA sync

- ERROR

- reset ACK
  - send RDY

- FINISHED

RECEIVE

- preprocess data
- prepare header

received header?

- yes
  - send header

- no
  - received header?

- yes
  - header match?

- yes
  - get data async
  - DMA sync
  - postprocess data
  - send ACK

- no
  - reset SYN
  - FINISHED

- ERROR

- reset ACK
  - send RDY

- FINISHED
MPI Types

We don’t do Boost.Serialization but

- you may register your POD type:

```cpp
struct gps_position { /* POD */ };  
namespace cbe_mpi
{
    CBE_MPI_USER_POD_DATATYPE(gps_position);
}
```

- or you may specialize send/receive methods:

```cpp
template<typename T>
request isend(cbe_mpi::communicator & comm, int dst, 
    int tag, T data, int n);

request irecv(cbe_mpi::communicator & comm, int src, 
    int tag, T data, int n);
```
Registering POD Types

How we identify your type:

```cpp
template<typename T>
struct cbe_mpi_user_pod_type_id { static void get() {} };

#define CBE_MPI_USER POD_DATATYPE(CppType) \
template<> \
struct is_mpi_datatype< CppType > \
: boost::mpl::bool_<true> {};
\ninline int get_mpi_datatype(const CppType &) \
{ \
    return 0x80000000 | \
    (int)(cbe_mpi_user_pod_type_id< CppType >::get); \
}
```
Sending std::vector

```cpp
template <typename T>
request isend(cbe_mpi::communicator com,
    int dest, int tag, const std::vector<T> * values, int)
{
    int vectorsize = values->size();
    com.send(dest, tag, &vectorsize, 1);
    return com.isend(dest, tag, &(*values)[0], vectorsize);
}

template <typename T>
request irecv(cbe_mpi::communicator com,
    int source, int tag, std::vector<T> * values, int)
{
    int vectorsize;
    com.recv(source, tag, &vectorsize, 1);
    values->resize(vectorsize);
    return com.irecv(source, tag, &(*values)[0], vectorsize);
}
```
MPI - Sending Unaligned Data

$s_{begin}$ $s_{aligned}$

source

$S_{aligned\_end}$ $S_{end}$

main block

destination

$d_{begin}$ $d_{aligned}$

$D_{aligned\_end}$ $D_{end}$
MPI - Sending Unaligned Data

source

32 byte buffer

destination

main block

D_begin D_aligned D_end

Daligned_end

S_begin S_aligned copy to buffer Saligned_end S_end

47 of 49
MPI - Sending Unaligned Data

Source: $S_{\text{begin}}$ $S_{\text{aligned}}$

Destination: $D_{\text{begin}}$ $D_{\text{aligned}}$

Copy to buffer:

Main block:

Align DMA put:

Main block:

Buffer copy to buffer:

Aligned DMA put:

32 byte buffer:

aligned DMA put buffer
MPI - Sending Unaligned Data

- **Source**
  - \( S_{\text{begin}} \) to \( S_{\text{aligned}} \)
  - Copy to buffer
  - \( S_{\text{aligned\_end}} \) to \( S_{\text{end}} \)
  - 32 byte buffer

- **Destination**
  - \( D_{\text{begin}} \) to \( D_{\text{aligned}} \)
  - Aligned DMA put main block
  - \( D_{\text{aligned\_end}} \) to \( D_{\text{end}} \)

- **Main Block**
  - \( S_{\text{aligned}} \) to \( D_{\text{aligned}} \)

- **In-Place Copy**
  - \( D_{\text{aligned}} \) to \( S_{\text{aligned}} \)

- **At Destination**
  - \( D_{\text{aligned\_end}} - S_{\text{aligned\_end}} \)
MPI - Sending Unaligned Data

S\textsubscript{begin} S\textsubscript{aligned}

\textbf{source}

\begin{itemize}
\item copy to buffer
\item S\textsubscript{aligned\_end} S\textsubscript{end}
\end{itemize}

\begin{itemize}
\item 32 byte buffer
\item aligned DMA put main block
\end{itemize}

S\textsubscript{aligned\_end} - S\textsubscript{aligned\_end}

\textbf{destination}

\begin{itemize}
\item D\textsubscript{begin} D\textsubscript{aligned}
\item aligned DMA put main block
\item D\textsubscript{aligned\_end} - D\textsubscript{end}
\item copy from buffer
\item in place copy
\item at destination
\end{itemize}

D\textsubscript{aligned\_end} - S\textsubscript{aligned\_end}
Conclusion

- Build process can be simplified with CMake

- Boilerplate code can be simplified with the help of Boost (e.g. PP)

- Ambiguity of functions or macros in different compilation units can be exploited

- Optimal Boost solutions have to be adapted to embedded architecture

- Sweet spot between generic code and efficiency must be found

- Difficult low-level code can be wrapped nicely in C++ Concepts

- C++ Concepts can be even more powerful on special purpose hardware
Conclusion

- Build process can be simplified with CMake
- Boilerplate code can be simplified with the help of Boost (e.g. PP)
Conclusion

- Build process can be simplified with CMake
- Boilerplate code can be simplified with the help of Boost (e.g. PP)
- Ambiguity of functions or macros in different compilation units can be exploited
Conclusion

- Build process can be simplified with CMake
- Boilerplate code can be simplified with the help of Boost (e.g. PP)
- Ambiguity of functions or macros in different compilation units can be exploited
- Optimal Boost solutions have to be adapted to fit embedded architecture
Conclusion

- Build process can be simplified with CMake
- Boilerplate code can be simplified with the help of Boost (e.g. PP)
- Ambiguity of functions or macros in different compilation units can be exploited
- Optimal Boost solutions have to be adapted to fit embedded architecture
- Sweet spot between generic code and efficiency must be found
Conclusion

- Build process can be simplified with CMake
- Boilerplate code can be simplified with the help of Boost (e.g. PP)
- Ambiguity of functions or macros in different compilation units can be exploited

- Optimal Boost solutions have to be adapted to fit embedded architecture
- Sweet spot between generic code and efficiency must be found

- Difficult low-level code can be wrapped nicely in C++ Concepts
Conclusion

- Build process can be simplified with CMake
- Boilerplate code can be simplified with the help of Boost (e.g. PP)
- Ambiguity of functions or macros in different compilation units can be exploited

- Optimal Boost solutions have to be adapted to fit embedded architecture
- Sweet spot between generic code and efficiency must be found

- Difficult low-level code can be wrapped nicely in C++ Concepts
- C++ Concepts can be even more powerful on special purpose hardware
Conclusion

- Build process can be simplified with CMake
- Boilerplate code can be simplified with the help of Boost (e.g., PP)
- Ambiguity of functions or macros in different compilation units can be exploited

- Optimal Boost solutions have to be adapted to fit embedded architecture
- Sweet spot between generic code and efficiency must be found

- Difficult low-level code can be wrapped nicely in C++ Concepts
- C++ Concepts can be even more powerful on special purpose hardware
Thank you for your kind attention.