ClearSpeed Programming Model: An Introduction
Overview

- PC host communication to ClearSpeed
- A first look at $C^n$
- Using the toolchain: “hello world”
- Lower level review of ClearSpeed architecture
- Comparing $C^n$ to C
- Detailed look at $C^n$
- Using the debugger
- First parallel implementation
PC host communication to ClearSpeed
Reminder of memory hierarchy

• First target:
• Move data from host to CSX DRAM
• PEs can then access data to perform compute
Data flow: Initialization

- Board and host communicate via Linux kernel module or Windows driver
- Create a handle and establish the connection
Data flow: Load program

- Register intent of using the first processor on the card
- Load the code onto the enabled processor
Data flow: Transfer data

- Transfer data from host to board
- Semaphores synchronize transfers between host and board
Data flow: Run program

- Run the code on the enabled processor
- Host can continue with other work
Data flow: Transfer results, shutdown

- Send results back to host
- Halt board program and clean up
Host / Board interaction basics

• The basic model for interaction between the host and ClearSpeed accelerator is very simple:
  – CS accelerator can signal and wait for semaphores
  – CS accelerator cannot initiate data transactions with the host

• Host pushes data to and pulls data from CS accelerator
  – Host can also signal and receive semaphores.
Connecting to the board

A host application needs to perform the following sequence to launch a process on the board:

- Create a CSAPI handle
  - CSAPI_new
- Establish a connection with the board
  - CSAPI_connect
- Register the host application with the driver
  - CSAPI_register_application
- Load the CSX application on the desired chip
  - CSAPI_load
- Run the CSX application on the desired chip
  - CSAPI_run
Interacting with the board

• Get board memory address of a known symbol
  – CSAPI_get_symbol_value
    • This must be done after the application is loaded, if the dynamic load capability is to be used.

• Write/Read data to a retrieved memory address
  – CSAPI_write_mono_memory
  – CSAPI_read_mono_memory
    • Asynchronous variants of these routines also exist
    • A process does not need to be running for these operations to succeed, but the process needs to be loaded.
    • These should not be performed DURING process termination

• Managing semaphores
  – CSAPI_register_semaphore
    • This is required for each semaphore the host will wait for.
  – CSAPI_semaphore_wait
  – CSAPI_semaphore_signal
Cleaning up

- **Process termination**
  - `CSAPI_wait_on_terminate`
  - `CSAPI_get_return_value`

- **Clean-up**
  - `CSAPI_delete`

- **See CSX600 Runtime Software User Guide (document ID 06-UG-1345) for more details, including:**
  - managing multiple processes on the board/chip at once
  - managing board control registers
  - board reset
  - managing multi-threaded CSX applications
  - board memory allocation
  - managing multiple boards/chips
  - error handling
Multiple CSX600 processors, multiple accelerators

- Do not assume or hard-code any limits
- Discover number of boards
  - You may insert more accelerators later
  - Another host may have a different board count
- Discover number of processors on each board
  - Two per board (for now...)
- Code then scales with more accelerators
Two CSX600 processors on present boards

• **Remember to use both chips**
  – Call CSAPI_new first, target board 0
    • This loads the host library
  – Call CSAPI_num_processors
    • At present this is 2, but may change in future

• **CSAPI_new:**
  – Enable access to the requested CSX processor
  – Multiple processors means multiple calls to CSAPI_new
  – Each instance can run…
    …the same program (on different data), or
    …a different program altogether
Multiple ClearSpeed Advance accelerators

• **How many cards are available?**
  – Don’t assume – make code discover this
  – Your code with then scale with more accelerators

• **Use CSAPI to discover:**
  – Call CSAPI_new first – this loads the host library
  – CSAPI_num_cards returns number of accelerator cards
  – If just interrogating, call CSAPI_delete to clean up state
  – Can now call CSAPI_new
A first look at $C^n$
ClearSpeed’s CSX architecture extends SIMD

- Multi-threaded
- Unified serial/parallel processing
  - No host
  - Single instruction set
- Per processing element (PE) memory addressing
  - Different address per PE
- Per PE I/O
  - PEs opt to take part in operation or not
- Scalable
  - Memory, number of PEs, bandwidth
The CSX architecture is simple to program:

- Single program for serial and parallel operations
- Architecture and Compiler co-designed
- Instruction and Data caches
- Simple, regular 32-bit instruction set
- Large, flexible register file
- Fast thread context switching
- Built-in debug support
- Same development process as traditional architectures: compile, assemble, link...
- \( C^n \) is a simple parallel extension of C
Reminder $C^n$ — C with vector extensions for CSX

- **New Keywords**
  - mono and poly storage qualifiers
    - mono is a serial (single) variable
    - poly is a parallel (vector) variable
- **Mono variables in 1 GB DRAM**
- **Poly variables in 6 kB SRAM of each PE**
- **We’ll concentrate on mono, initially**
Mono execution unit

• **Summary of mono:**
  - Serial execution
  - Can branch
  - Communication with host

• **Example “hello world” code:**
  ```c
  #include <stdio.h>  // Output support
  int main()
  {
    printf("Hello world\n");
    return 0;
  }
  ```

• **As standard C – no parallel extensions (yet!)**
Useful mono code example: “hello_world.cn” for 3.0

- **3.0 has argc/argv support:**

```c
#include <stdio.h> // Output support
#include <stdlib.h> // String to integer support

int main(int argc, char *argv[])
{
    if( argc == 2 )
    {
        int instance = atoi(argv[0]); // Simple host -> mono comms
        int proc_inx = atoi(argv[1]);
        printf("Hello world from processor#%i on board%i\n", proc_inx, instance);

        int result = instance + proc_inx;
        return result; // Simple mono -> host comms
    }
    return 0;
}
```
Real-world usage

• **Usual approach:**
  – Use global variable in mono section of C
    • Such as a statically allocated data buffer
  – Load program onto accelerator
  – Search for each global variable’s symbol via CSAPI
  – Store data into buffer indicated by variable
  – Signal mono to start running (via semaphore)
  – Host waits for mono to signal completion semaphore
    • Or carries out work in parallel
  – Processing complete, mono signals semaphore
  – Host reads data from known buffer

• **Using argv and return codes…**
  – …rather limiting for large datasets, useful for simple tests
Using the toolchain: “hello world”
First step: compiling $C^n$

- **cscn**: $C^n$ compiler and linker
  - Similar to cc or gcc (C compiler and linker)
- **To build earlier “hello world” example:**
  
  ```bash
  cscn -g -o hello_world.csx hello_world.cn
  ```
- **We now need to write and compile the host code**
  - Note: Linux or Windows, link to ClearSpeed libraries, include ClearSpeed header path
  - Refer to `runtime_user_guide.pdf` for configuration of SDK, paths, etc.
Host code to support "hello_world.cn"

```c
#include <csapi.h>
#define CSX_FILE_NAME "hello_world.cn"

void main() // NOTE: V3 SPECIFIC!!! Otherwise, v.nasty
{
    DRIVErno return_code;
    struct CSAPIState *s;
    int csx_exit_code;
    char parameters[10];
    int num_cards, num_processors;

    return_code = CSAPI_num_cards(&num_cards);
    for (int instance=0; instance < num_cards; instance++)
    {
        s = CSAPI_new( CM_Direct );
        return_code = CSAPI_connect( s , NULL , instance ); // Connect to board
        return_code = CSAPI_num_processors( s, &num_processors );

        for (int proc_inx=0; proc_inx < num_processors; proc_inx++)
        {
            CSAPIProcess process = CSAPI_load( s , proc_inx , CSX_FILE_NAME ); // Load compiled C
            sprintf(parameters, "%i %i", instance, proc_inx); // Parameters – white space delimited
            return_code = CSAPI_run( s , process, parameters ); // Run on ClearSpeed Advance
            return_code = CSAPI_wait_on_terminate( s , proc_inx ); // Wait for completion
            return_code = CSAPI_get_return_value( s , proc_inx , &csx_exit_code ); // Note return value
            printf("ClearSpeed Advance board #%i processor#%i returned %i\n", instance, proc_inx, csx_exit_code);
            CSAPI_delete( s );
        }
    }
}
```
Running host code

• Now compile previous slide’s code
  – Using Visual C++ or gcc on Linux

• Run your compiled program
  – It will grab the compiled C\textsuperscript{n} and run this as required
  – Note: On all available CSX processors on the machine
    • Such as 2 CSX processors on an Advance e620/X620

• Output produced (2 boards):
  ClearSpeed Advance board #0 processor#0 returned 0
  ClearSpeed Advance board #0 processor#1 returned 1
  ClearSpeed Advance board #1 processor#0 returned 1
  ClearSpeed Advance board #1 processor#1 returned 2
Code scalability

• Note interrogation to discover number of boards
  – And related number of CSX processors on each board
• Code automatically scales to additional ClearSpeed accelerators
• Note: this code is a simple example
  – Launches each ClearSpeed processor serially
  – Waits for each processor to terminate
  – No overlap of compute between boards / processors
  – Not an optimal use of resources
    • But makes a simple example with consistent output
Code scalability (continued)

• **Related CSAPI calls:**
  – CSAPI_get_free_mem – available DRAM for processor
  – CSAPI_num_pes – number of PEs on processor

• **Use these to avoid hard-wiring your code**
  – Can automatically adapt to later editions of the architecture or boards
Lower level review of ClearSpeed architecture
What this section covers

- **We have given a simple introduction to:**
  - Host API
  - “mono” usage within $C^n$
- **We have not covered parallel with “poly”**
  - Will now review the parallel parts of the architecture
  - Will provide context for “poly” in $C^n$
Reminder of memory hierarchy

- **Fastest access:**
  - Poly register
- **Largest capacity:**
  - Host PC
- **Hence trade-off between speed and capacity**
  - Akin to multiple levels of cache
  - But user has control
Architecture of CSX600 processor

- **Single instruction stream**
  - Fed into mono and poly execution units
- **Mono execution unit:**
  - Serial computation
  - Carry out branching
- **Poly execution unit:**
  - 96 units in parallel
  - Conditional execution
  - Local 6kB SRAM

![Diagram of CSX600 processor architecture]
Highlights of mono execution unit

- **Dedicated to processing mono data**
  - Scalar / non-parallel data
- **Handles program flow control**
- **Conditional execution**
  - Branching (conditionals, subroutines)
- **Multi-threaded execution**
  - Rapid thread switch
  - Semaphore support
    - Such as completion of I/O
    - Wait on semaphore causes thread switch etc.
Highlights of poly execution unit

- **Multiple functional units**
  - Floating point unit (FPU)
  - Integer multiply-add (MAC)
  - Integer arithmetic/logic unit (ALU)
  - Load/store to private memory
  - All work in parallel – e.g. integer operation performed simultaneously with floating point operation

- **Conditional behaviour**
  - Cannot branch due to SIMD nature
  - Can enable / disable individual PEs
  - Akin to predicated instruction execution

- **Enable state**
- **I/O mechanisms**
Highlights of poly execution unit (continued)

- **Swazzle Data Movement**
- **Register-to-register transfer between neighboring PE’s**
  - ~160 GB/sec bandwidth *per processor*
Comparing $C^n$ to $C$
Reminder $C^n$ — C with vector extensions for CSX

- **New Keywords**
  - `mono` and `poly` storage qualifiers
    - `mono` is a serial (single) variable
    - `poly` is a parallel (vector) variable
- **Mono variables in 1 GB DRAM**
- **Poly variables in 6 Kbyte SRAM of each PE**
C^n differences from C

• Quick overview
  – Will cover C^n in detail later

• New data type multiplicity modifiers:
  – mono: denotes serial variable
    • resident in “mono” memory (i.e., DDR)
    • mono is the default multiplicity
  – poly: denotes parallel/vector variable
    • resident in “poly” memory (i.e., local to each PE)
Multiplicity applies to pointers, doubly so:
- mono int * poly foo;
  - foo is a pointer in poly memory to an int in mono memory
- poly int * mono bar;
  - bar is a pointer in mono memory to an int in poly memory
- int * poly *mono good_grief;
  - as you would expect....

Pointer sizes:
- mono int *
  - 4 bytes (32-bit addressable space, 512 MB)
- poly int *
  - 2 bytes (16-bit addressable space, 6 kB)
• **Conditions based on mono expressions**
  - Expression has same value on all PEs
    - Code block selected according to expression and branch instruction executed
  - Jump occurs as in a traditional architecture

```c
mono int i, j;
i = j = 1;
if( i == j ) {
    // this block executed on all PEs
} else {
    // this block branched over on all PEs
}
```
Conditional expressions: poly-if

- **Conditions based on poly expressions**
  - Expression may have different values on different PEs
  - But SIMD model: PEs execute same instruction simultaneously
    - All branches must be executed on all PEs
  - PE enabled if conditional expression is true – disabled otherwise
    - Like predicated instructions

```c
poly int i;
i = get_penum();
if( i < 48 ) {
    // PEs 0, 1, 2, ... execute instructions
    // PEs 48, 49, ... instructions issued but ignored
} else {
    // PEs 0, 1, 2, ... instructions issued but ignored
    // PEs 48, 49, ... execute instructions
}
```
Conditional expressions: poly-while

- **While-loops based on poly expressions**
  - Loop continues execution until condition is false on all PEs
  - PEs will be disabled one by one until while condition is false on all PEs
  - `count` keeps track of total number of iterations (96 in this case)

```plaintext
mono int count = 0;
poly int me;
me = get_penum();
while( me > 0 ) {
    --me;
    ++count;
}
```
Function return types: mono or poly

- Function multiplicity affects how returns are handled
  - Analogous to handling of conditionals

- Function with poly return type:
  - Will return once all of the PEs have returned a value and all remaining mono code has executed
  - In effect, hitting a return statement early will disable a PE until the end of the function
  - If there is no mono code and all PEs have returned:
    - Compiler may be able to optimize by returning early.

- Function with mono return type:
  - Will return immediately a return is executed
Use of mono variables inside poly conditionals

- Implications for mono objects inside poly flow control:

  poly short penum = get_penum();
  mono int i = 0;
  if (penum < 32) { ... /* Do some work */
   i ++; /* Increment mono variable */
  } else { ... /* Do some other work */
   i ++; /* Increment mono variable */
  }

- Both branches are executed (as poly conditional)
  - But mono variable in both branches
  - Value of \( i \) post-branch: 2!
  - All mono commands are also executed
    - Otherwise, above we’d have \( \frac{1}{2} \) of mono branching…
Mixing mono and poly conditions (1 of 3)

• Conditional execution where conditional behaviour not explicit
  – Example: use of boolean operators
  – Typically to combine conditional expressions

• && and the || operators are guaranteed to do lazy evaluation
  – Only evaluate the right hand sub-expression if it is necessary to determine the truth or falsehood of the expression
  – The evaluation of the second sub-expression is conditional

• Implications when the expression mixes mono and poly sub-expressions
Mixing mono and poly conditions (2 of 3)

- Consider:

  ```
  mono int m;
  poly int p;
  ...
  if ((m == 0) && (p == 0))
  ```

- **If** \( m \neq 0 \) \( \Rightarrow \) **whole expression is false**
  - Value of \( p \) is never tested
  - Mono can branch over the code

- **\( p \) is then tested**
  - Individual PE's are disabled if \( p = 0 \) (ignore instructions)
  - Mono processor will execute code in the block
    - As mentioned before: we can’t have \( \frac{1}{2} \) of mono executing…
Mixing mono and poly conditions (3 of 3)

- What happens when the conditional expression is rewritten as:
  
  \[ \text{if } (p == 0) \&\& (m == 0) \]

- Value of \( p \) is tested first; if this is false…
  - Simply disables the execution of further poly code
  - Processor will continue executing mono code
  - Evaluates second expression to determine the value of \( m \).

- The second expression is always evaluated!
  - Any side-effects of its evaluation will always occur.

- Note: Overall condition is still a poly expression
  - “Else” statement: mono will execute both branches
\( \text{C}^n \text{ extensions: labeled loop constructs} \)

- Akin to Java’s labeled `break` and `continue`

```c
// Label for_i associated with for loop
for_i: for (i = 0; i < 10000; i++) {
    while (j > 100) {
        do { // ...
            if (foo == bar) { break for_i; } // ...
        } while (a != b); }
}
```

- `break` will break out of the deepest point in the nested loop to the outermost level
  - Most of the flexibility of `goto`, but in a more structured way

- Support in poly and mono
  - Not often supported in parallel architectures
Restrictions / omissions compared to C

• No goto statement in poly context
  – Use labeled break/continue instead

• switch/case on mono data only
  – Poly: use multiple “if” statements

• Structs must reside in a single memory space
  – Can’t have both mono and poly contents

• No dereferencing across memory spaces
  – Supported via library calls
Conditional expressions on poly

• To re-iterate:
  – Poly conditionals are a convenience
  – Not true branching
  – All PEs execute ALL instructions
  – Just that some PEs ignore the instructions
    • Akin to predicated execution
  – Mono cannot ignore mono instructions inside poly conditional

• So remember:
  – You will pay the total cycle count for both halves of a poly if-then-else construct
Porting C to C\(^n\) (Example 1)

**C code**

```c
int i, j;
for( i=0; i<96; i++ )
{
    j = 2*i;
}
```

**Similar C\(^n\) code**

```c
poly int i, j;
i = get_penum(); // i=0 on PE0, i=1 on PE1 etc.
j = 2*i;         // j=0 on PE0, j=2 on PE2 etc.
```
Porting C to \( C^n \) (Example 2)

C code

```c
int i;
for( i=0; i<N; i++ ) {
    ...
}
```

Similar \( C^n \) code

```c
poly int me, i;
mono int npes;
me = get_penum(); // me=0 on PE0, me=1 on PE1...
npes = get_num_pes(); // npes = 96
// i=0,96,192, ...; 1,97,193, ... etc.
for( i=me; i<N; i+=npes ) {
    ...
}
```
Detailed look at C^n
Implicit broadcast

- Implicit broadcast from mono to poly by assignment

```c
mono int m = 7;
poly int p;
p = m;  // implicit broadcast to all PEs
```

- Reverse operation has ambiguous meaning and is not permitted

```c
mono int m;
poly int p = get_penum();
m = p;  // m receives different values from each PE
       // (so whoever wrote last would "win")
```
Explicit data movement

- Memory copy of $n$ bytes from mono to poly
  `memcpym2p()`
  - Source is a poly pointer to mono memory, which can have a different value for each PE
  - Destination is a mono pointer to poly memory, that is destination address is the same for all PEs

![Diagram showing memory copy](image.png)
Explicit data movement

- Memory copy of $n$ bytes from poly to mono
  `memcpyp2m()`
  - Source is a mono pointer to poly memory, that is source address is the same for all PEs
  - Destination is a poly pointer to mono memory, which can have a different value for each PE
Using **mono** and **poly** with pointers

- mono int * mono mPmi  
  mono pointer to mono int
- poly int * mono mPpi  
  mono pointer to poly int
- mono int * poly pPmi  
  poly pointer to mono int
- poly int * poly pPpi  
  poly pointer to poly int

Most commonly used is mono pointer to poly

- poly <type> * mono <variable_name>

Further reading:

- Introductory Programming Manual (document 06-UG-1117), Section 5.3 “Mono and poly specifiers”
• **mono** pointer to **mono** int
  
  \[
  \text{mono int} \ast \text{mono mPmi}
  \]
\( C^n \) — mono and poly pointers

- **mono** pointer to **poly** int
  
  \[
  \text{poly int} * \text{mono pPmi}
  \]

Note: Points to same location in each PE
C^n — mono and poly pointers

- **poly** pointer to **poly** int
  
poly int * poly pPpi

Note: Pointer stored in same location in each PE
C^n — mono and poly pointers

• **poly** pointer to **mono** int

\[\text{mono int * poly pPmi}\]

Note: Pointer stored in same location in each PE
Dereferencing mono*poly pointers

- In general, a mono value can be assigned to a poly variable
- Cannot dereference (implicitly or explicitly) a mono * poly pointer
  - Each PE’s instance of the pointer can point to a different location in mono memory
  - Dereferencing the pointer would require a complex operation
  - Different memory location is copied to each PE
  - Can be done, but requires explicit use of library functions such as memcpym2p()
Dereferencing mono*poly pointers

• Example:

```c
mono int array[96];
mono int * poly p;
poly int x, n;
int i;
for (i = 0; i < 96; i++) {
    array[i] = i; // initialize array contents
    x = i;       // a legal mono to poly assignment
}

n = get_penum(); // different value on each PE
p = array + n;    // different address on each PE
x = *p;           // illegal explicit dereference
x = array[n];     // illegal (implicit) dereference
```

• Illegal dereference will generate a compiler error.
Pointer effect on program performance

• **Different pointer types run at different speeds**
  – We recommend (in order of speed):
    • mono * mono
    • mono * poly
    • poly * poly

• **Note that a poly pointer takes up poly memory**
  – A scarce resource
Runtime libraries

• Supports Standard C Runtime Libraries including:
  – malloc
  – printf
  – sqrt
  – memcpy

$C^n$ extensions including:
  – sqrtp
  – memcpym2p / memcpyp2m

$C^n$ specific routines including:
  – get_penum
  – swazzle
  – any / all
Asynchronous I/O

- For most efficient use of limited PE memory:
- Overlap data transfers between mono memory and poly
  - Refer to \textit{C^n} Standard Library Reference Manual
    - Section 3.2: Asynchronous memory transfer (string\_ext.h)
Asynchronous explicit data movement

- Asynchronous memory copy of \( n \) bytes from mono to poly or poly to mono
  - `async_memcpym2p()`, `async_memcpyp2m()`
  - Computation continues during data copy

- Mono memory data cache NOT flushed
  - If you’ve touched data by mono variable access
    - Change could still be in the cache
  - Then write over / read it via async memcpy
  - It may not be synchronized!
    - So, if mono then flushes its cache… could overwrite memcpy
  - Use `dcache_flush`
Asynchronous explicit data movement (cont.)

- **Restrictions on alignment of data**
  - Mono address needs to be 8 byte aligned
    - 32 byte alignment for maximum performance
  - Poly address needs to be 4 byte aligned

- **Use semaphores to wait for completion of copy**

- **Much higher bandwidth than synchronous versions**
  - Sync versions do not have any alignment restrictions

- **Example:**
  
  ```c
  dcache_flush();
  async_memcpym2p( semaphore, ... );
  // computation continues
  sem_wait( semaphore );
  // use data that has been transferred from mono memory
  ```
Swazzle operations

- Register-to-register transfer between neighboring PE’s
  - 8 bytes can be moved per cycle
  - \( \sim 160 \text{ GB/s} \) bandwidth \textit{per processor}
- \( C^n \) versions operate on data and include implicit data movement from memory to registers
  - Assembly language versions operate directly on register file
- Variants
  - \texttt{swazzle\_up( poly\ int\ src );} // copy to higher numbered PE
  - \texttt{swazzle\_down( poly\ int\ src );} // copy to lower numbered PE
  - \texttt{swazzle\_up\_generic( poly\ void\ *dst, poly\ void\ *src, unsigned\ int\ size );}
  - \texttt{swazzle\_down\_generic( … );}
  - Similar swazzles operating on other data types
  - Functions to set data copied into ends of swazzle chain
Using the Debugger
csgdb Debugger (Shown with ddd Front-end)

- On-chip poly array contents displayed
- Real time plot of contents of PE memory
- Cn source-level breakpoint, watch points, single step, etc.
- Register contents
- Disassembly, breakpoint, watch points, single step, etc.
csgdb/ddd debugger and csvprof integration

- x86 gdb port enables standard GUIs
  - Multi core
  - Multi thread
  - Single step, breakpoint, watchpoint

- csgdb port consistent with x86 gdb
  - Enables standard GUIs with the CSX600
  - Single step, breakpoint, watchpoint
  - Full symbolic debugging

- csgdb port is multi-everything
  - Card, processor, thread, PE
  - Profiler control integrated via new csgdb command set.

- Visualize all the state in everything
- Visualize all data movement PCI bus
Debugging with a host application

- **To enable debugging:**
  - Set environment variables (UNIX bash example):
    export CS_CSAPI_DEBUGGER=1
    - Initializes the debug interface within the host application
    export CS_CSAPI_DEBUGGER_ATTACH=1
    - Host application will then write a port number to stdout
  - Host will wait for <Return/Enter> to be pressed
    - csgdb can be manually attached to the connected board process

- **Launch the host application**
  - This can be done with or without a debugger.
• **Launch csgdb in a new shell**

  csgdb <csx_file_name> <port_number>
  
  • No need to “connect” as the host application did this already
  • Port number is required so gdb can connect to host process
  
  – Set desired breakpoints
  – Run
    
    • Note that the host is currently blocked waiting for <Return/Enter>, so card process may also be blocked waiting for the host.

• **Press the Enter key in the host shell for the host and card applications to proceed.**
Debugging without a host application

• If your program does not have a host application:
  – Can launch csgdb directly on the CSX executable
    • No need to link the host via a port number:
      csgdb <csx_file_name>
  – But you do need to connect to the accelerator
    • Example:
      (gdb) connect
      main() at myapp.cn:94
      94 int main() {
      (gdb)
Worked example using csgdb

• Refer to Introductory Programming Manual (document 06-UG-117)
  – Section 7: Debugging C
  – Uses Mandelbrot.cn as an example

• Useful csgdb commands:
  list <function name>
  • Lists 10 lines around the start of the given function
  break <function name>
  • Stops execution at the given function
  next
  • Steps to next program line (steps over function calls)
  print <variable name>
  • Displays the contents of the given variable
  run
  • Starts your program
  help
  • Shows command help within csgdb

• Alternatively:
  – Use a GUI front-end, such as DDD
csgdb Command-line example

% cscn foo.cn –g –o foo.csx
% csgdb ./foo.csx
(gdb) connect
0x80000000 in __FRAME_BEGIN_MONO__ ()
(gdb) break 109
Breakpoint 1 at 0x800154c0: file foo.cn, line 109.
(gdb) run
Starting program: /home/kris/my_app/foo.csx
Breakpoint 1, main () at foo.cn:109
(gdb) next
110 y = MINY + (get_penum() * STEPY);
(gdb) print y
$1 = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
First parallel implementation
Simple test application

- Apply 2×2 matrix to N two-element vectors and record the normalized result
  - Stored as a simple 1D array of double

- First:
  \[
  \begin{pmatrix}
  M_0 & M_1 \\
  M_2 & M_3
  \end{pmatrix}
  \begin{pmatrix}
  A_i \\
  A_{i+1}
  \end{pmatrix}
  =
  \begin{pmatrix}
  M_0 . A_i + M_1 . A_{i+1} \\
  M_2 . A_i + M_3 . A_{i+1}
  \end{pmatrix}
  \]

- Then: normalize the resulting vector
  - \( \text{length} = \sqrt{(A_i \times A_i) + (A_{i+1} \times A_{i+1})} \)
  - \([B_i, B_{i+1}] = [A_i / \text{length}, A_{i+1} / \text{length}]\)

- Where matrix “M” multiples values of array “A”
  - Read elements of “A” in pairs

- Store results in array “B”
  - Again, paired elements
Initial C code – also compiles as “mono” C

#define SUB_BUFFER_SIZE 256
#define BUFFER_SIZE 96*SUB_BUFFER_SIZE
// Assume preloaded with sensible values
double A[BUFFER_SIZE];
double M[4];
// Not required to be initialized
double B[BUFFER_SIZE];
double scratch0, scratch1, length;

int main()
{
    int i;
    for (i=0; i<BUFFER_SIZE; i+=2)
    {
        scratch0 = (M[0] * A[i]) + (M[1] * A[i+1]);
        length = sqrt((scratch0 * scratch0) + (scratch1 * scratch1));
        B[i] = scratch0 / length;
        B[i+1] = scratch1 / length;
    }
}
Required host side to support $C^n$ code

- No need to send down processor / board number
  
  - You can if you want! :^)
- Instead, populate $A[\ ]$ & $M[\ ]$ with initial values
- Need to read back from shared $B[\ ]$ array
- We'll initially target a single processor on a single board
Host side of matrix code

```c
#include <csapi.h>
#define CSX_FILE_NAME "matrix_example.cn"

void main()
{
    DRVErrno return_code;
    struct CSAPIState *s;
    int csx_exit_code;
    double *mono_A, *mono_B, *mono_M;

    return_code = CSAPI_num_cards(&num_cards);
    for (int instance=0; instance < num_cards; instance++)
    {
        s = CSAPI_new( CM_Direct );
        return_code = CSAPI_connect( s , NULL , instance ); // Connect to board
        return_code = CSAPI_num_processors( s, &num_processors );

        for (int proc_inx=0; proc_inx < num_processors; proc_inx++)
        {
            CSAPIProcess process = CSAPI_load( s , proc_inx , CSX_FILE_NAME ); // Load compiled Cn
            status = CSAPI_get_symbol_value( s, CSX_FILE_NAME, "A", mono_A); // Board memory: A[]
            status = CSAPI_get_symbol_value( s, CSX_FILE_NAME, "B", mono_B); // Board memory: B[]
            status = CSAPI_get_symbol_value( s, CSX_FILE_NAME, "M", mono_M); // Board memory: M[]
            // Stash useful values into A[] and M[]
            *cs_instance = instance;
            *cs_proc_inx = proc_inx;
            return_code = CSAPI_run( s , process ); // Run on ClearSpeed Advance
            return_code = CSAPI_wait_on_terminate( s , proc_inx ); // Wait for completion
            // Read out useful values from B[]
            CSAPI_delete( s );
        }
    }
}
```
#define BUFFER_SIZE (96*256)
#define SUB_BUFFER_SIZE (BUFFER_SIZE / 96)
// Assume preloaded with sensible values
mono double A[BUFFER_SIZE];
mono double M[4];
// Not required to be initialized
mono double B[BUFFER_SIZE];
poly double scratch0, scratch1, length;
// Subset of work on each PE - localized poly copy
poly double polyA[SUB_BUFFER_SIZE], polyB[SUB_BUFFER_SIZE], polyM[4];

int main()
{
    memcplym2p(polyA, A+(SUB_BUFFER_SIZE*get_penum())), SUB_BUFFER_SIZE*sizeof(double));

    int i;
    for (i=0; i<SUB_BUFFER_SIZE; i+=2)
    {
        scratch0 = (polyM[0] * polyA[i]) + (polyM[1] * polyA[i+1]);
        length = sqrt((scratch0 * scratch0) + (scratch1 * scratch1));
        polyB[i] = scratch0 / length;
        polyB[i+1] = scratch1 / length;
    }
    memcppyp2m(B+(SUB_BUFFER_SIZE*get_penum())), polyB, SUB_BUFFER_SIZE*sizeof(double));
}
Introduction of poly: code description

- Array A[] is resident in mono memory
- Send A[] from mono to poly – "Scatter"
  - Subset of A[] resides in each PE
  - Hence memcpym2p uses SUB_BUFFER_SIZE
  - Not BUFFER_SIZE
  - Also note mono memory address – unique per PE
- Lazily send M[] – as it's so small
- Locally compute, storing in local B[]
- Send B[] from poly to mono – "Gather"
Further improvement

• **Asynchronous I/O**
  – At present, we wait for all the data, then compute

• **Alternative:**
  – Send subset of data
  – Start compute AND send further data
  – Should overlap compute with I/O
  – Hence “hide” wait for I/O

• **Further reading:**
  – Introductory Programming Manual (06-UG-1117)
    • Section 5.7.1: Data transfers between mono and poly
  – The C$^n$ Standard Library
    • Section 3.2: Asynchronous memory transfer (string_ext.h)
Asynchronous code design: Pseudo code

```plaintext
async send block A1
async send block A2
set semaphore for block B1 (pretend already sent)
set semaphore for block B2 (pretend already sent)

while more work to do
    wait for block B1 to be sent (don’t overwrite)
    wait for block A1 to arrive (awaiting work)
    block B1 = matrix M * block A1
    async send block B1 (work complete)
    async request block A1 (need more work)

    wait for block B2 to be sent (don’t overwrite)
    wait for block A2 to arrive (awaiting work)
    block B2 = matrix M * block A2
    async send block B2 (work complete)
    async request block A2 (need more work)
```
Asynchronous code design: Timeline

Block 1

A in  compute  B out

Block 2

A in  compute  B out

Increasing time →
Asynchronous code design: Final notes

• **Arrange PEs to read in consecutive memory**
  – Most efficient memory access
  – Rather than widely-spaced scatter over memory
Asynchronous code: C^n (part 1)

#include <string_ext.h> // for async_memcpy
#define BUFFER_SIZE (96*256)
#define NUM_ITERATIONS 16
// All PEs will work together on 1/16th of the data each iteration
#define SUB_BUFFER_SIZE (BUFFER_SIZE / NUM_ITERATIONS)
// Each PE will work on (1/16th)/96th of the data each iteration
#define SUB_BUFFER_BLOCK_SIZE (SUB_BUFFER_SIZE / 96)
// Assume preloaded with sensible values - variables as previous poly implementation...
mono double A[BUFFER_SIZE];
mono double M[4];
// Not required to be initialized
mono double B[BUFFER_SIZE];
poly double scratch0, scratch1, length;

#define SEMA_A_1 16
#define SEMA_B_1 17
#define SEMA_A_2 18
#define SEMA_B_2 19

// Subset of work on each PE - localised poly blocks
poly double polyA1[SUB_BUFFER_BLOCK_SIZE], polyB1[SUB_BUFFER_BLOCK_SIZE];
poly double polyA2[SUB_BUFFER_BLOCK_SIZE], polyB2[SUB_BUFFER_BLOCK_SIZE];
poly double polyM[4];

int main()
{
  // "Seed" the algorithm
  async_memcpy2p(SEMA_A_1, polyA1, // Code is verbose for clarity - see SUB_BUFFER_SIZE usage
                  A+(SUB_BUFFER_SIZE*0)+(SUB_BUFFER_BLOCK_SIZE*get_penum()),
                  SUB_BUFFER_BLOCK_SIZE*sizeof(double));
  async_memcpy2p(SEMA_A_2, polyA2,
                  A+(SUB_BUFFER_SIZE*1)+(SUB_BUFFER_BLOCK_SIZE*get_penum()),
                  SUB_BUFFER_BLOCK_SIZE*sizeof(double));
int subBufferNum = 0;
for (subBufferNum = 0; subBufferNum < NUM_ITERATIONS; subBufferNum+=2) {
    // Wait on semaphores... work on block #1
    sem_wait(SEMA_A_1);
    sem_wait(SEMA_B_1);
    // Calculate block of work
    for (i=0; i<SUB_BUFFER_BLOCK_SIZE; i+=2) {
        scratch0 = (polyM[0] * polyA1[i]) + (polyM[1] * polyA1[i+1]);
        length = sqrt((scratch0 * scratch0) + (scratch1 * scratch1));
        polyB1[i] = scratch0 / length;
        polyB1[i+1] = scratch1 / length;
    }
    async_memcpyp2m(SEMA_B_1, // Send back results
        B+(SUB_BUFFER_SIZE*subBufferNum)+(SUB_BUFFER_BLOCK_SIZE*get_penum()),
        polyB1,
        SUB_BUFFER_BLOCK_SIZE*sizeof(double));
    async_memcpym2p(SEMA_A_1, polyA1, // Get new input
        A+(SUB_BUFFER_SIZE*(subBufferNum+2))+(SUB_BUFFER_BLOCK_SIZE*get_penum()),
        SUB_BUFFER_BLOCK_SIZE*sizeof(double));
    // Now, work on block #2
    sem_wait(SEMA_A_2);
    sem_wait(SEMA_B_2);
    // ...Calculate block of work...
    async_memcpyp2m(SEMA_B_2, // Send back results
        B+(SUB_BUFFER_SIZE*(subBufferNum+1))+(SUB_BUFFER_BLOCK_SIZE*get_penum()),
        polyB2,
        SUB_BUFFER_BLOCK_SIZE*sizeof(double));
    async_memcpym2p(SEMA_A_1, polyA1, // Get new input
        A+(SUB_BUFFER_SIZE*(subBufferNum+2))+(SUB_BUFFER_BLOCK_SIZE*get_penum()),
        SUB_BUFFER_BLOCK_SIZE*sizeof(double)); // ... etc. ... with final cleanup outside loop
Summary
Summary

- PC host communication to ClearSpeed
- A first look at $C^n$
- Using the toolchain: “hello world”
- Lower level review of ClearSpeed architecture
- Comparing $C^n$ to C
- Detailed look at $C^n$
- Using the debugger
- First parallel implementation