Internship HMF Transmitter

Kai Husmann

November 2011

Electronic Vision(s), Kirchhoff-Institut für Physik,
Ruprecht-Karls-Universität Heidelberg

Internship HMF Transmitter

This internship covers the *HMF Transmitter*[1] tests. The aim of these tests was to find the maximum data rate at which it is possible to transfer HICANN[2] data from one process to another. Thereby using a shared memory area that will later be read autonomously by the network device itself. This document shows the results of the evaluated bandwidth and latency. Furthermore it describes the interprocess circular buffer which was programmed within these tests.

Praktikum HMF Transmitter

Dieser Praktikumsbericht behandelt die *HMF Transmitter*[1] Tests. Ziel dieser Tests war es die maximale Datenrate zu ermitteln, mit welcher HICANN[2] Daten von einem Prozess zu einem anderen übermittelt werden können. Dabei wurde ein „shared memory“ Bereich verendet welcher später autonom von der Netzwerk Karte direkt gelesen werden soll. Dieses Dokument liefert eine Einschätzung über die erreichbare Bandbreite und Latenz. Weiterhin beschreibt es den Interprozess-Ringpuffer (interprocess circular buffer) welcher im Rahmen dieser Tests entwickelt wurde.

[1]Hybrid Multiscale Facility (HMF)
[2]High Input Count Analog Neural Network (HICANN)

Contents

1. Introduction

1.1. Motivation

The “Electronic Vision(s) Group” at the Kirchhoff-Institut für Physik was founded in 1995. Initially research was focused on CMOS image sensors. Later, the research focused on tactile perception and displays.

In recent years, the group started to build neuromorphic hardware. A chip-based system called *Spikey* is in productive use and a successor called *HICANN*[1] [*Millner et al.*, 2010] is running as prototype system. To allow for larger scale neural networks which require high connectivity, a wafer-scale system that replicates ca. 384 HICANNs and their inter-connectivity onto a single wafer is under development.

Currently, the “Electronic Vision(s) Group” takes part at the “BrainScaleS project” which aims at understanding function and interaction of multiple spatial and temporal scales in brain information processing. One research task of the BrainScaleS project is to develop a non von-Neumann HMF:

> We will build a facility for the exploration of non von-Neumann computing architectures, in particular for multiscale emulations of neural systems. The Hybrid Multiscale Facility (HMF) combines a neuromorphic computing system composed of custom designed neural circuits in microelectronics with conventional high performance numerical computers. The neuromorphic system is a physical model of neural microcircuits featuring low energy consumption per neural event, fault tolerance, scalability and the capability to learn. Networks can be assembled from 1.6 million neurons and 0.4 billion dynamic synapses with user configurable parameters and network architectures. The merging of the two computational concepts into a hybrid system provides a new experimental platform suited to bridge temporal scales from milliseconds to years and at the same time to study spatial scales from the single cell level to functional brain areas in a single experiment at speeds far exceeding biological real-time. By virtue of the numerical computing infrastructure as an integral part of the HMF, the system will provide virtual environments and generate sensory inputs as well as motor feedback in order to realise multiscale closed-loop experiments addressing cognitive tasks. [...]
>
> *BrainScaleS* [2011]

This report is a part of the neuromorphic (non von-Neumann) hardware which is developed within the “BrainScaleS” HMF research task. The HMF consists of two mayor

[1]High Input Count Analog Neural Network

parts. One part is the so called HMF Conventional Hardware (HMFConvHw) consisting of (conventional) backbone computers on which the *MappingTool* [*Ehrlich et al.*, 2010; *Wendt et al.*, 2010; *Brüderle et al.*, 2011] and other software runs. The other part is the neuromorphic part (NP) consisting of the wafer-scale system: this system is made up of several so called wafer-scale integration units (*WSI-system*). The heart of each unit builds a printed circuit board called *mainPCB* which forms the connection to one wafer and to 12 customizable communication subgroups. These subgroups will manage the inter-wafer communication as well as the external connections to the backbone computers using an FPGA. The backbone computers are equipped with RDMA capable network interface cards (RDMA-NICs) for connection with each other and to the FPGA boards. The Remote Direct Memory Access (RDMA) technology comes in handy here. It allows the bundled computers to access each others RAM in a direct and cheap manner. And one can define RAM areas which can be accessed by the RDMA-NICs as well as by distributed software applications [*OpenFabrics*, 2011]. This is important for the HMFConvHw to be able handling enough data to satisfy the NPs needs.

As there is lots of data that has to be exchanged between the HMFConvHw and the NP, for example pulse data, as well as within the HMFConvHw the transfer rate at which we can offer data to the RDMA-NICs is of high interest.

1.2. Assignment

The aim of the *HMF Transmitter* internship was to evaluate the maximum transfer rate at which it is possible to

1. offer data (e.g. pulse data) through RAM to the RDMA-NIC
2. read data stored in RAM by the RDMA-NIC

As the RDMA-NICs were not available during this internship they have been excluded from this evaluation. But its expected that the RAM accessed by software and RDMA-NIC in reading and writing manner is the bottle neck. Therefore a simulation that transfers data through a Shared-Memory Area (shamem) had to be programmed (representing a software accessing a shared memory which is also available to an RDMA-NIC).

This simulation is the *HMF Transmitter*. And this report describes its production as well as its evaluation.

1.3. The simulator

The simulator consists of two independent processes transferring data from one to the other using a Shared-Memory Area (shamem). These processes named *Generator* and *Receiver* form – together with the Starter, which initialises the shamem and then starts a Generator-Receiver pair – the main classes of the HMF Transmitter. Additionally there is a shell script to start up multiple pairs/Starters to evaluate how parallel processes can influence the overall data rate.

The results are expected to give a reasonable upper bound at which we can offer data through RAM from a software interface to an RDMA-NIC.

2. General Design

Looking at the later use of *HMF Transmitter* namely providing HICANN data at high speed to a network device through RAM, it was clear that using the C++ Programming Language (C++) would bring best results: Apart from being widely spread, well known and fast, C++ allows precise control about data structures and how their content is put into RAM. This was found to be of great importance when passing data from software to hardware.

When writing C++ code the BOOST C++ Libraries[1] are fairly known. They are a collection of free open source libraries of well formed and frequently used C++ structures that aim at becoming C++ standards. Especially the BOOST interprocess library [*Boost.Interprocess*, 2011] seemed to have everything one needs to create intercommunicating processes. So it was decided on building the interprocess communication upon the BOOST interprocess library (`BOOST::Interprocess`).

The data should be transported through a circular buffer. It seems to be the best structure/pattern for our purpose. There is a circular buffer structure within `BOOST` but that unfortunately had problems interacting with interprocess routines. More on that in chapter 3. Since the `BOOST` circular buffer was not working within our constraints next choice was to create an interprocess functional circular buffer myself. But before the work on the circular buffer could begin the data structures had do be defined. They can be found in the file *structure.h*. The pulse data which is actually transferred is packed in a "Matryoshka doll" style. The smallest struct is the `pulse_t` (timestamp and label) and a load of these is packed into a `pulse_packet_t` (id, count and multiple pulses). Finally the `buffer_packet_t` holds a pulse packet together with a control flag and some padding. To avoid page alignment problems it was decided to align each `buffer_packet_t` at 8 KiB. Here were some problems too, more on that again in chapter 3.

Further *HMF Transmitter* settings, manageable through `#define`s, can be reviewed in the appendix A.3. Additionally there is an overview of the files contained within the *HMF Transmitter* package in appendix A.2.

2.1. Self-made circular buffer

A circular buffer is a memory area or vector which is circled through. That means – starting writing at `0` – we move ahead element by element and when reaching the end we jump back to the first element (`0`) overwriting it. As in our case we cannot accept loss of data we must assure that element `0` is read before the generator flips back there. And so on. That was archived by implementing a Receiver Read-head (r-head) and a Generator

[1] Available at `http://www.boost.org`

Write-head (g-head) and defining the circular structure such that g-head and r-head are never touching. Therefore the buffer size must at minimum be 3 where generator and receiver will access the circular buffer alternately. In the following ASCII example *b* denotes a blank/free element, *d* an element containing data and *g* and *r* the accordant head's position. If a head is followed by a *!* its blocked. Each word of chars is a possible follow up situation of the one before. But showing subsequent moves of the same head might have been omitted.

size 2: `bb` $\mapsto$ `gb` $\mapsto$ `dg!` $\mapsto$ `r!g!` and now both block each other

size 3: `bbb` $\mapsto$ `gbb` $\mapsto$ `ddg!` $\mapsto$ `rdg!` $\mapsto$ `br!g` (g flips back) $\mapsto$ `g!rd` (alternating blocks)

size 5: `bbbbb` $\mapsto$ `dddgb` $\mapsto$ `rddgb ...`

Normally either the generator or the receiver is faster and the situation will turn into something like this:

fast g: `...  rdddg!` $\mapsto$ `bbrdg` $\mapsto$ `dg!rdd` $\mapsto$ `dgbbr` $\mapsto$ `dddg!r` (only g blocks)

fast r: `...  bbr!gb` $\mapsto$ `bbrdg` $\mapsto$ `bbbr!g` $\mapsto$ `dgbrd` $\mapsto$ `r!gbbb` (only r blocks)

It is possible to program a circular buffer without this kind of gr-head structure. Then the receiver just tests if its next element has been fully written meanwhile the generator only tests if its next element has been read already. But that results in a less clear situation and in respect of race conditions and other interprocess complications the strict handling seems to be the better promise. And well since the final version not one packet was observed to be transmitted erroneously. And that without using any memory fences and even in a 3 weeks run (longest test performed). Furthermore the gr-head structure allows for a better and clearer observation of the circular buffers accessed positions.

Also some words have to be said about the start setting. As the generator needs writing first we have the possible situation of the generator filling the whole buffer before the receiver has started, leading to a possible situation of the generator outdistancing the receiver. This actually was an earlier bug that wasn't easy to trigger (as normal code uses high buffer sizes), hard to find but easy to fix. Let us finally take a look at the start-up process and the "master-while-loop" (working loop), which are illustrated as algorithms 1 and 2.

```
1: shamem is initialised:
           both heads stati = HEAD_INIT, both heads pos = POS_INIT = -42 < 0
2: generator (G) and receiver (R) are started (in any given order)
3: they both do some further initialisation..
4: and R sets its position to rpos = 0 and finally its status to
           rstate = HEAD_WAIT_TO_START
5: G waits until R leaves state HEAD_INIT:
           while (r_head_p->status == HEAD_INIT) {...}
     • and then puts its pos to gpos = 0, enters the master-while-loop and starts
       generating
           d[..d]gb..b
     • G can't pass gpos=0 a second time until the receiver has started (see algorithm 2)
6: Alongside R waits on G to have written its first elements
           while (gpos < 1) {...}
     • and then enters it's master-while-loop (see algorithm 2) and starts to receive
           r[d..d]g[b..b]
7: now both processes have entered their master-while-loop.
8: NB: The heads check to wait only prior to moving ahead. So if gpos=n and rpos=n+1
   G will block after having written n, remaining on n. The same accounts for R.
```

Algorithm 1: The start-up process

```
1: always entering loop with r-pos/g-pos = [0..BUFFER_SIZE]
2: write or read element on pos
3: check if other process stands at pos+1 and wait until it has continued
4: now increase own pos by one (and make sure that pos=BUFFER_SIZE≥pos=0)
```

Algorithm 2: The master-while-loop

3. Encountered problems and curious observations

As stated already there were some issues with the `BOOST` circular buffer, as well as with `BOOST` and aligned structs. Also the fact that we we did not need any memory fences is interesting.

3.1. Problems with BOOST circular buffer

Looking through the `BOOST` documentation I found a class "circular buffer" there. On first sight this seemed to be the perfect code concerning the assignment. But then the `BOOST` circular buffer was not aimed at interprocess assignments. Putting the circular buffer within a Shared-Memory Area (shamem) it throws a runtime error. But to that a solution can be found in the `BOOST` documentation. Search for `BOOST_CB_DISABLE_DEBUG` in the circular buffer documentation [*Boost.CircularBuffer*, 2011]. One must disable that debug flag to get the circular buffer to run within a shamem.

In our concern the use of the `BOOST` circular buffer is to simply check its size (the amount of elements within) and decide thereafter if a new element could be retrieved or generated. The *Generator* checks for `size` < `capacity` and only then it puts a new element into the circular buffer whereas the *Receiver* checks if `size` > `0` and only then retrieves the next element. Unfortunately the `BOOST` circular buffer size implementation is not interprocess safe. After lots of testing it became clear that nothing can be done to change that fact. There are race conditions between the value of size, the elements within and where the receiver or generator are accessing the circular buffer. After working fine for a short period of time bad elements fall in and eventually become more until one gets a failure rate of 100%. Even putting memory fences in all thinkable places – within some tests the code contained almost more fences then other statements – did not help.

So far I can conclude that the `BOOST` circular buffer implementation can not be used within an interprocess environment, period. The solution to this problem was programming an interprocess functional circular buffer on my own, on top of the `BOOST::Interprocess` vector which is – as expected – interprocess tested. This leads us to the next problem: alignment.

3.2. Problems with the alignment

The first definitions of buffer structs in *structure.h* were using alignment attribute statements and looked similar to the listing 3.1.

```c
#define BUFFER_PACKET_SZ      ( 8 KiB_IEC )

// buffer_packet_t
typedef struct {
  uint32_t flag;                            // header
  pulse_packet_t pulse_packet;              // data
} __attribute__ ((aligned (BUFFER_PACKET_SZ), packed)) buffer_packet_t;
```

Listing 3.1: buffer packet using alignment attribute

```c
#define BUFFER_PACKET_SZ        ( 8 KiB_IEC )

// expected size of uint32_t, see buffer_packet_t.flag
#define BUFFER_PACKET_HEADER  4

#define BUFFER_PACKET_PADDING \
        ( BUFFER_PACKET_SZ - BUFFER_PACKET_HEADER - PULSE_PACKET_SZ )

typedef struct {
  uint32_t flag;                            // header
  pulse_packet_t pulse_packet;              // data
  uint8_t padding[BUFFER_PACKET_PADDING];
} __attribute__ (( packed )) buffer_packet_t;
```

Listing 3.2: buffer packet using padding

But putting such aligned structs within a `BOOST::Interprocess` vector – which is the base of the self-made circular buffer – did not work. The `BOOST` allocator has its own thoughts upon alignment and they are not compatible with using the alignment attribute. The solution to this was to use padding instead. It was tried to avoid padding as it produces less readable and reusable code, but it seems that one has to live with it. The above code therefore became the following: listing 3.2.

3.3. About memory fences

As memory fences themselves are an enormous field apart from reading various papers and scanning through some books the best way to avoid errors is defensive coding, of course, and heavy testing. And this is what was done. After having a stable code and thinking of the right places for fences some tests where started without fences in expectance of errors... If then enabling the fences would remove these errors the fences would probably have been well placed. Due to these expectations it was rather disturbing that even a three weeks run lead to no errors. On the other hand this fail-free test is some evidence that either the self-made circular buffer has just a that stable implementation that no fences are needed or its just an effect of x86 architecture.. But well not every possible settings were tested. Just in case, the fences were left commented out in the correct places.

4. Performed Tests

During production of the *HMF Transmitter* various tests were performed. The results of these tests were not kept as their intention was not measuring the performance but to check the code for being bug free. The longest of these tests ran for three weeks and no failures arose. So I may say that the code can be considered as being bug free in means of typical usage. The last steps of this internship were the performance tests. On the way producing them the code was cleaned up a bit (e.g. removing `printf` statements and some older remarks) but nothing should have been changed that may introduce new bugs or influences performance measurable. Well there was one greater change: extracting the shamem-initialisation from the Generator to a newly introduced process – the *Starter*. But as the shamem-initialisation already was encapsulated within the `shamemInit.cpp` that was no big deal. Before going into further detail the test environment will be outlined in the following section.

4.1. Environment

4.1.1. code branches

Three different test settings (different code) were executed:

001 `INIT_TEST` performance testing of the creation and initialization process of the shamem[1]

002 `ONE_PAIR` performance testing of one Generator-Receiver pair

003 `MULTI_PAIR` performance testing of multiple Generator-Receiver pairs using different shamems[2]

The resulting code of test 3 is the most modular and clean one and should be used in further development. Of course – when one can run multiple pairs a multiplicand of one is also possible; so the creation of test 3 code actually obsoleted the test 2 code. Because test 3 produces a neater output the one-pair tests where run again using test 3. The results of test 2 are therefore not discussed any further. The code of the `INIT_TEST` is quite chaotic as most code was not supposed to execute. It is just a dirty branch of the *HMF Transmitter* with no further purpose than performing the `INIT_TEST`.

[1]test code and non-processed results: *tag/TEST_ 001_ INIT*
[2]test code and non-processed results: *tag/TEST_ 003_ MULTI_ PAIR*

4.1.2. `#define` options

Throughout all tests three values as `BUFFER_SIZE` were used. For better recognition they were arbitrarily named (after the power of some machines I worked on during this internship). All buffer sizes are prime numbers as this guards us for missing certain errors on a circular structure. For example, using a prime ring-size element `x` on run `y` is surely distinct from element `x, (y-1)`.[3] Like this we are able to assert correct reception.

1. `HACKOMAT 24989 (PRIME_2762)`
2. `FOXI 49999 (PRIME_5133)`
3. `DOPAMINE 74959 (PRIME_7393)`

No other values where altered.

4.1.3. compile flags

The *HMF Transmitter* makefile has (within others) two different master targets: make debug and make optimize. Additionally one can specify the so called `EXENV` which in effect passes a define to the compiler telling *structure.h* which `BUFFER_SIZE` to take. The following list shows 3 example make statements and their outcome:

```
make g++ -Wall -std=gnu++0x -DEXEC_HACKOMAT -c ...

make -e EXENV=FOXI debug g++ -Wall -std=gnu++0x -DEXEC_FOXI -g -O0 -c ...

make -e EXENV=DOPAMINE optimize g++ -Wall -std=gnu++0x -DEXEC_DOPAMINE
     -O3 -c ...
```

`-Wall` produces warnings on bad code style, without `-std=gnu++0x BOOST` does not work, `-g` enables debugging, but the difference between the tests is due to the specified `EXENV` and the different optimisation levels. The debug tests were especially performed to check if the optimisation might have spoiled the test by optimising away some test code. If the `-O3` results had turned out faster in orders of magnitude the results would have had to be double checked. Furthermore the receiver checks if all packages are received correctly.[4] Therefore I can say that all parts of the code have executed.

4.2. Results and plots

4.2.1. Initialisation of the Shared-Memory Area

The first test that was performed was the `INIT_TEST`. Its aim was to receive information on whether it is possible to start-up the *HMF Transmitter* on need and finally shut it

[3]each buffer packet is generated on base of the count of previous generated buffer packets; see `flagval/FLAG_MAX` which must be distinct from `BUFFER_SIZE`.

[4]We know the expected package as the generator uses a predictable non-random generation of pulse data.

down. As the initialisation takes a few tenths of a second and we expect the transportation of buffer packets in microseconds frequent initialisations should be avoided. It is favourable to call the initialisation only once and then reuse the buffer on demand. Further it was observed that in the `ONE_PAIR` test the first package (of 1,000 buffer packets) was also one of the slowest packages (high above the mean and mostly equal to the slowest package measured). This is explainable by the fact that on Linux, `malloc()` requests memory by calling `sbrk()` which expands the process' address space, but the actual assignment of memory pages happens upon the first write to this memory. This kind of lazy reservation of RAM should actually count to the initialisation time but it is not measurable by the initialisation test itself. The first/last package time is not represented in any plot but one can find these values in the appendix.

Furthermore it was checked if compiler optimisation would speed up the initialisation process. But figure 4.1 shows clearly that there is no noteworthy difference between compiling with debugging or optimising options. It could be observed that the average time per buffer packet (blue and purple boxes) decreases slightly with an increased buffer size. To make this visible to the observer, the y2 range (right side) was set to start at 3.5 µs – all measured values lie within 4 and 4.3 µs. So as long as the shamem is not to be initialised very often and if an increased buffer size brings along a better bandwidth there is nothing that indicates against increasing the buffer size.

All initialisation tests were performed 120,000 times and thereof the mean and standard deviation (sd) was calculated.

4.2.2. One pair

The following two test settings (one pair and multiple pairs) both measured the time after every 1,000 buffer packets being send (one test package). After 1,000,000 test packages being send the test stopped. Mean time and standard deviation were calculated out of the average of these measurements. Finally the bandwidth was calculated thereof.

Let us now take a look at the run of one pair comparing debug and optimise compiler setting. As one can see in figure 4.2 compiling with optimise (`-O3`) is measurable better. But one must also see that there is a significant covering of the sd margins of both tests. So one cannot expect much out of compiler optimisation considering the *HMF Transmitter* code. This is probably due to the actually quite clear and simple loop which is performed endless times and cannot be estimated and optimised by the compiler as he cannot know anything about the volatile variables. For the generator the r-head position is volatile as for the receiver it is vice versa. Also these results show clearly that all code was performed as otherwise we would see an extreme increase in speed within the optimised setting. In an earlier test there was a problem with the optimiser "optimising away" the code[5] that should actually be tested.

[5]Cf. the famous Linux 2.6.30 0-day exploit:
`http://lists.grok.org.uk/pipermail/full-disclosure/2009-July/069714.html`

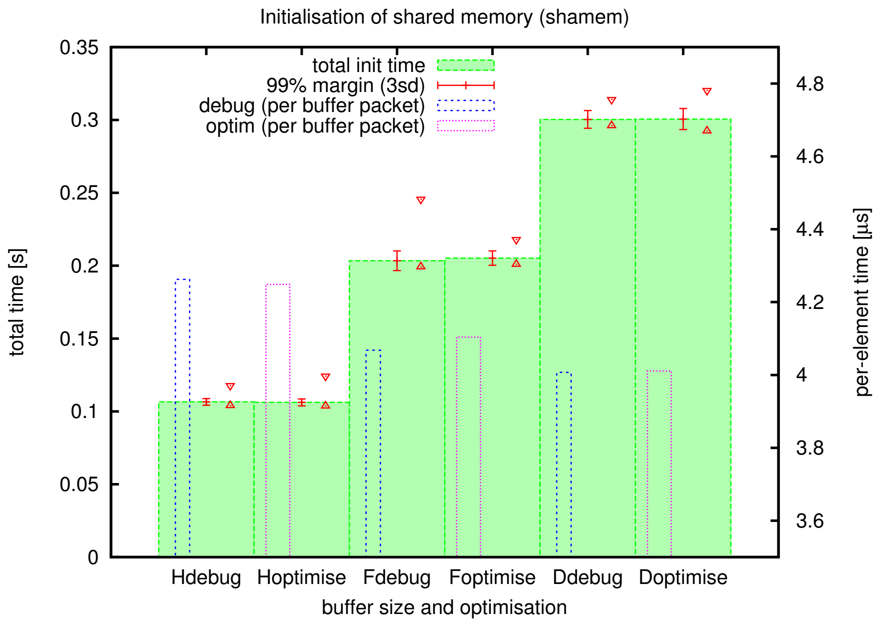

Figure 4.1.: Initialisation of shared memory circular buffer. The green bars show the total time the initialisation takes. The red error margin is 3sd wide so that one can see something and the red triangles show the minima and maxima measured. The blue and purple bars show the time per buffer packet.

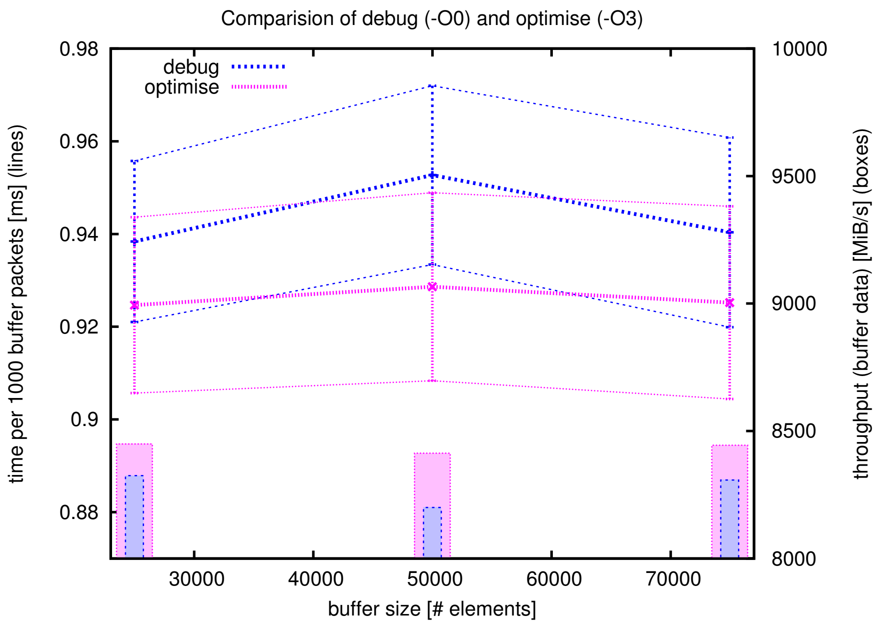

Figure 4.2.: One pair: comparing debug and optimise compile flag. As in figure 4.1 the blue and purple colours distinguish between the debug and optimise compile settings. The boxes at the bottom show the calculated bandwidth whereas the lines denote the mean time of the measured test packages (each 1,000 buffer packets). The error lines denote the 1sd margin.

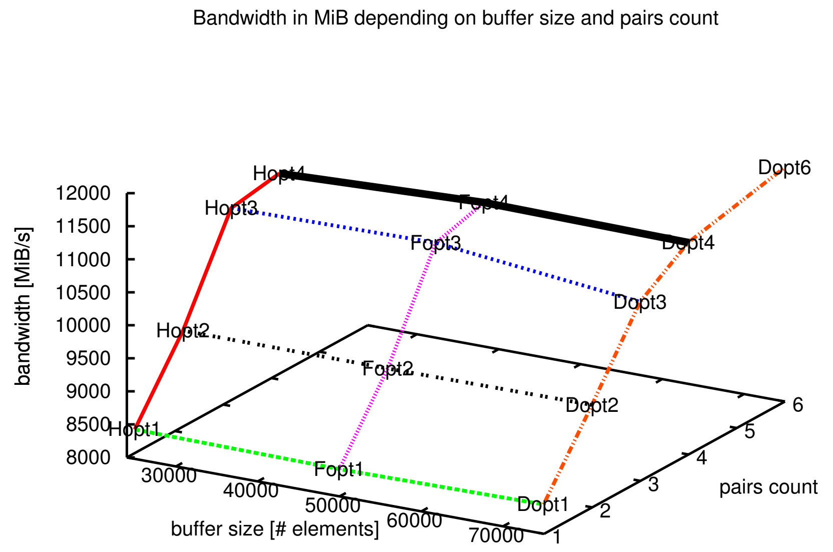

Figure 4.3.: Overview of the multiple pairs test scenario

4.2.3. Multiple pairs

The final and most interesting tests are the tests with multiple pairs. All these tests are using the same colour scheme. The scheme can be studied best in figure 4.3 which gives a quick and clean overview of the test results considering the total bandwidth. In the figures 4.4 and 4.5 all lines refer to the mean time per test package whereas the boxes denote the bandwidth and use the y2 axes (right side). All tests with 1, 2, 3, and 4 pairs where performed using the three available buffer sizes Hackomat, Foxi, and Dopamine. The 6 pairs test was performed using the Dopamine setting (biggest buffer size) only as its aim was to show what happens when we overpower the available machine – which had 8 CPUs whereas the 6 pairs test would actually need 12 (for each process having its own CPU).

In figure 4.4 we can see that with each additional pair the total bandwidth increases even though high costs are paid: The mean time transferring 1,000 buffer packets (one test package) is increased heavily. Also we see that with 3 and more pairs the reliability of a test package being written to the shared memory within a given time is lost. We can see this especially well in figure 4.5.

Apart from 3 pairs we cannot see any significant influence of the buffer size used. Taking a look at figure 4.5 we can see that the Foxi setting (buffer size 49999) gains some advantage over the Dopamine setting.

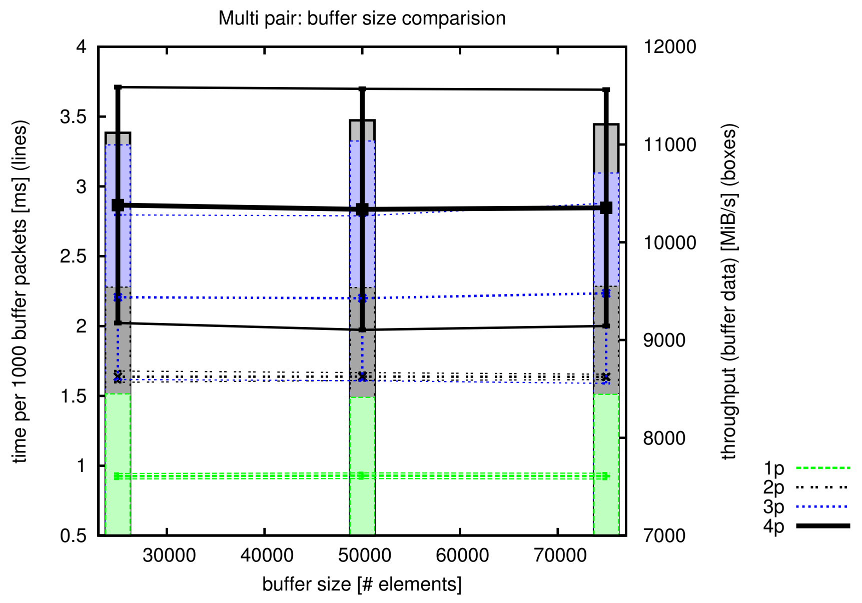

Figure 4.4.: This plot shows one line for all pairs (1 to 4) and the time their test packages needed in mean to be transmitted depending on the buffer size used. The 6 pairs result is not shown as it has only one buffer size setting. Furthermore its mean package time lies far above the other results and would not contribute to this plot's readability.

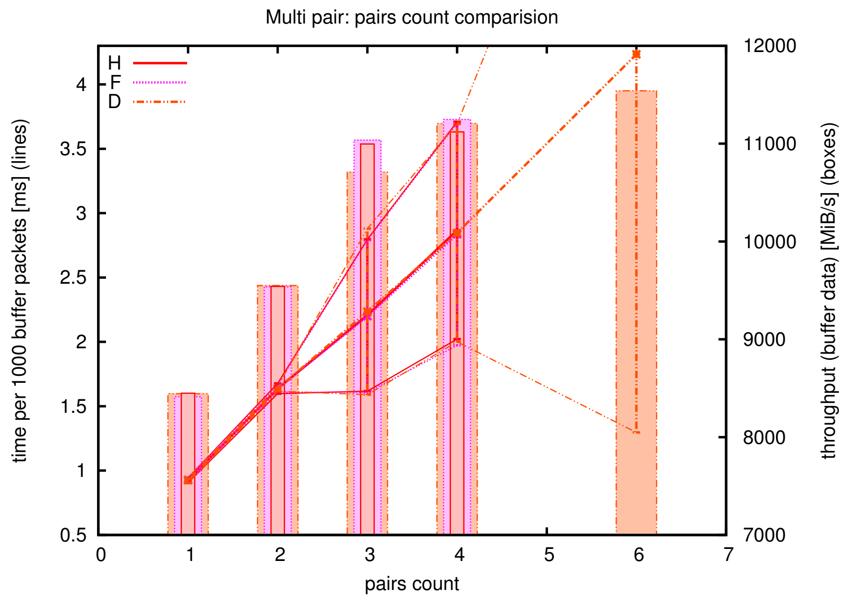

Figure 4.5.: This plot shows one line for every buffer size setting.

5. Conclusion

One might miss a chapter discussion, but as the test results were discussed along chapter 4.2 already this report ends just with a short summary and an outlook.

5.1. Summary

- The `BOOST` circular buffer is - considering our needs - not compatible with the BOOST interprocess library (see chapter 3.1).
- The *HMF Transmitter* code should generally be compiled with `-O3`. And a comparison with `-O0` should be executed to check for unexpected discrepancies.
- The buffer size should be a prime number as this guards for missing errors in test phase (see chapter 4.1.2).
- Volatile variables are sufficient and combined with the structure of the *HMF Transmitter* circular buffer they seem to render memory fences unnecessary (see chapter 2.1).
- The buffer size has no significant effect on bandwidth (see any plot).

  If there is RAM shortage a smaller buffer size should work just as well.
- Multiple pairs do increase the bandwidth but bring along a loss considering the reliability of the transportation time of a single packet (especially see figure 4.5).

  More than 2 pairs will decrease mean time reliability significantly.

I consider the `FOXI` buffer size setting combined with 3 pairs to be the best trade-off between bandwidth and mean time reliability (see figure 4.5). With this setting a bandwidth of about 10 GB per second can be reached (11301591 KiB) and the mean time per 1,000 buffer packets lies around 2.1991311 ms with a standard deviation of 0.5903547 ms. The bandwidth reached should suffice the Hybrid Multiscale Facility's needs [*FACETS M7-5*, 2009; *Brüderle et al.*, 2011].

5.2. Outlook

As with any project there is still some work left. The next steps that should be considered will be outlined here.

5.2.1. Buffer sizes

As we did not see great differences using the three defined buffer sizes these tests should be run again using rather small buffer sizes. Also a buffer size sweep should be useful to find which buffer size works best with a given number of pairs. To support a buffer size sweep the code has to be restructured such that it is able to close down the shamem, cleaning up all remains and then starting up again but with a different buffer size. Therefore the actual structure with a buffer size define will not work. Well another possibility would be to change the *SuperStarter.sh* such that it is able to run make and passing an arbitrary value for the buffer size to the make command.

5.2.2. Buffer packet size

All tests were ran with a buffer packet size of 8 KiB to ensure proper page alignment. But there were actually no tests ran evaluating the effect on total bandwidth when changing this value. There were two reasons why a size of 8 KiB was used:

The page size of the machine where the tests were ran is 4 KiB and 8 KiB is expected to be safe for modern machines. Run `getconf PAGE_SIZE` on console to receive your machines page size.

Secondly the page size has to be fitted to the FPGAs needs such that it can hold a full FPGA Ethernet jumbo frame, which – in our case – is specified as 7 KiB [*FACETS M7-5*, 2009]. Therefore it was decided to use 8 KiB as this is the first multiple of the test machines page size greater than that frame size. Note that the size of a `pulse_packet_t`, being 7 KiB, was also chosen according to the FPGA Ethernet frame size.

5.2.3. Initialisation process

The initialisation process is not optimal. There is the problem with the lazy RAM reservation (see page 11) which cuts down the reliability of the initialisation tests as well as of the first buffer packets filling the buffer within a given time. The initialisation of the buffer is actually using `buffer->assign(BUFFER_SIZE, createInvalidBufferPacket());`. Another approach could be using `buffer->reserve(BUFFER_SIZE)` instead or in combination. Finally one could add to the initialisation the filling of the buffer once with dummy values such that the reservation of the RAM is enforced. This could lead to a more accurate value of the initialisation and to more reliability within the working loop. This of cause will cut down overall performance but if it is decided to initialise the shamem early the initialisation could take place prior to the time critical phase of the project – thereby doing no harm.

5.2.4. Making a fully fledged C++ class

The whole circular buffer code could be packed into a fully fleged C++ class rendering the interprocess circular buffer effortlessly reusable. A design could be creating a *circular buffer management class* which has methods like `initialiseShamem()`, `getWriter()`

and `getReader()`. The get methods should return some type of *circular buffer access class*.

5.2.5. RDMA

Be that as it may, the most important next step will be the adaption and testing of the *HMF Transmitter* using actual RDMA-NICs. At first two computers sharing a common “RDMA-shamem” should suffice.

Following this report there will be a bachelor thesis which will concentrate on using RDMA-NICs.

5.3. Acknowledgments

This document would not look that good without the great latex usage hints and the general revision by *Eric Müller*. Further thanks go to *Alex Bradbury* who helped with some English.

Last but not least I must thank my young daughter *Zora* who motivated with her smiles.

A. Appendix

A.1. Delivered report package

Along with this report a git repository is delivered, which is structured as follows:

```
HMF Transmitter
└── code
    ├── tag ............... codes used for evaluation and their corresponding results
    │   ├── TEST_001_INIT
    │   └── TEST_003_MULTI_PAIR
    └── trunk ............................ final state: HMF Transmitter Version 1
├── doc ............................................... documentation (latex)
└── plot ................................ gnuplot/ .plt files, generating the plots
    └── dat ....................................... data files used for plotting
```

The full package can be found at `https://brainscales-r.kip.uni-heidelberg.de/projects/report-khusmann`.

A.2. Classes and code organization

A.2.1. List of C++ Programming Language files

cActivityPrinter.cpp Shows that something is happening (used in debug mode only)

cMicroCounter.cpp Helps a bit in measuring microseconds

Generator.cpp Main Class: Generates data and puts them into the circular buffer in shamem

Receiver.cpp Main Class: Receives data out of the circular buffer in shamem.

shamemInit.cpp Constructs and initialises the shamem and the circular buffer therein.

Starter.cpp Main Class: calls method in *shamemInit.cpp* to construct the shamem and then starts a Generator-Receiver pair

structure.cpp Helps constructing and asserting diverse structures and `structs`; it is `BOOST` independent!

A.2.2. List of h files

Most .h files are just what they are named after – header files (cActivityPrinter.h, cMicroCounter.h, shamemInit.h). But there is more about these three:

exitDescriptions.h Contains some exit codes

shamemTypes.h Contains all `BOOST` specific shamem `typedef`s

structure.h Apart from being *structure.cpp*'s header file this is the file which exerts most control upon *HMF Transmitter*'s code

As *structure.h* with all its parameters and defines is so important the next appendix section is dedicated to that file in particular.

A.3. structure.h: defines

The file *structure.h* lots of define statements, some are just aliases for clearer code, e.g. `#define MFENCE() __asm__ __volatile__ ("mfence;":::"memory")` or constants to remember universal values `#define PRIME_1131`. These should not be changed and will not be documented any further. They are self declaring and of minor importance only. But some defines are rather switches and changing them will have great influence on the finally compiled code, e.g. `#define USE_SCHED_YIELDS` or `#define BUFFER_PACKET_SZ ( 8 KiB_IEC )`. The following list will give a short description on each of these elements.
Fist note will be the setting used in `TEST_003`.

- Changing active code blocks

  USE_SCHED_YIELDS defined. Enables use of `sched_yield()` within any waiting while loop, should be defined for proper functioning. Disabling it puts CPU usage at 100 percent but with lower efficiency.

  COPY_ACCESS undefined. If defined the receiver will copy the read buffer packet using `memcpy` before comparing it to the expected. If not only a reference to the receivers active buffer packet is kept and its contents are compared.

  COMPLETE_COMPARE undefined. If defined the complete buffer packet is compared upon arrival. Enabling this slows down the *HMF Transmitter* extremely because the complete packet has not only to be compared but the comparator has to be generated as well. If not defined only the expected `flagval` will be checked. Note that the recent code never failed regardless of the settings of `USE_SCHED_YIELDS`, `COPY_ACCESS` and `COMPLETE_COMPARE`.

- Changing test measurements

  FLAG_MAX 1,000. Every `FLAG_MAX` elements measurements are taken, decreasing this number slows down overall performance while it probably increases the measured performance (the time used for the calculation is subtracted from the measured time, but the other process may use that time!). All Tests in `TEST_003` were performed with a value of 1000 this seems to be a reasonable value leading to measurements taken about every millisecond.

TST_PACKAGE_CNT 1,000,000. Number of `PACKAGES` a `FLAG_MAX` buffer packets to be transmitted. Increasing this number will increase the reliability of test results, the time a test takes and the RAM used for measurement (one double per package). Therefore it should be high but in reasonable bounds. One million leads to tests of about 16 minutes and gives a fair reliability of the results.

- Changing RAM usage

BUFFER_SIZE distinct. Number of buffer packets within the circular buffer - raising this number increases used RAM significantly. Should be set according to available RAM and `BUFFER_PACKET_SIZE`. The buffer size is defined through compile environments, code expects to be compiled with either of `-D [EXEC_DOPAMINE|EXEC_FOXI|EXEC_HACKOMAT]` and will set buffer size accordingly to `PRIME_7393`, `PRIME_5133`, `PRIME_2762`.

BUFFER_PACKET_SZ 8 `KiB_IEC`. Specifies the exact size of one buffer packet. Padding/ MaxPulses will be calculated according to this value. When changing it be aware that `BUFFER_PACKET_SZ * BUFFER_SIZE + peanuts` equals the RAM usage of one Generator-Receiver pair. A size of 8 KiB assures clean page alignment and looks like the best value.

PULSE_PACKET_SZ 7 `KiB_IEC`. Specifies the amount of HICANN/ pulse data per buffer packet. Value must be smaller then `BUFFER_PACKET_SZ - BUFFER_PACKET_HEADER`. So at the moment there is 1 KiB free for the `BUFFER_PACKET_HEADER`. That gives room for the `buffer_packet_t` to be changed since it uses only 4 of the 1024 spare bytes. And 7 KiB seem in respect of the expected length of pulse packets (`pulses_count`) to be sufficient.

- Other stuff

BUFFER_IO_HEAD_SZ 1 `KiB_IEC`. Size of `struct buffer_head_t` (position and status of circular buffers g-head/r-head). Value should not be changed.

A.4. Plot data files

Out of the following .dat files the plots are generated (using gnuplot). These files again are derived out of the *.result files which can be found in the tags *TEST_001_INIT* and *TEST_003_MULTI_PAIR*. The .dat files can be found in side-project *plot*.[1]

A.4.1. init_test.dat

```
# GNUplot data file
# HMF Transmitter TEST_INIT
# data derived from tag TEST_001_INIT/test_001.result
#
# Initialisation of Shamem
#
# measuring total init time in seconds, col 8 is per element in usec
# name      nb BUF_SZ min       avg       max       sd        element
# 1         2  3      4         5         6         7         8
# ------------------------------------------------------------------------
Hdebug      1  24989  0.10408   0.10651   0.11767   0.00077   4.2623
Hoptimise   2  24989  0.1037340 0.1061690 0.1242070 0.0007883 4.2486
Fdebug      3  49999  0.1992879 0.2033972 0.2456210 0.0022568 4.0680
Foptimise   4  49999  0.2009740 0.2051974 0.2179220 0.0016418 4.1040
Ddebug      5  74959  0.2960839 0.3003860 0.3140180 0.0020232 4.0073
Doptimise   6  74959  0.2924840 0.3006551 0.3202870 0.0024141 4.0109
```

[1]The structure of the report bundle can be found in appendix A.1

A.4.2. one_pair.dat

```
# GNUplot data file
# HMF Transmitter TEST_ONE_PAIR
# data derived from tag TEST_003_MULTI_PAIR/*1_pairs*.result

# Comparison of HMF Transmitter results using one Gen/Rec pair
#
# An element is one buffer packet
# A measured package contains of 1,000 elements
# 10^6 packages were measured
# total buffer packets transmitted during test therefore 10^9

# description  | el    |                          ms                           | microsec  | KiB/s bandwidth            |
# name      | nb | BUF_SZ | min       | avg       | max       | sd        | element   | buffer      | pulse       |
# 1         | 2  | 3      | 4         | 5         | 6         | 7         | 8         | 9           | 10          |
# ----------|----|--------|-----------|-----------|-----------|-----------|-----------|-------------|-------------|
Hdebug      1    24989    0.8950233   0.9383918   3.7908554   0.0173679   0.9383918   8525223.422   7459570.495
Hoptimise   2    24989    0.8811951   0.9246600   3.6139488   0.0189712   0.9246600   8651829.092   7570350.455
# ----------|----|--------|-----------|-----------|-----------|-----------|-----------|-------------|-------------|
Fdebug      3    49999    0.9031296   0.9527076   3.0291080   0.0192867   0.9527076   8397120.140   7347480.122
Foptimise   4    49999    0.8800030   0.9286324   3.6070347   0.0202740   0.9286324   8614818.652   7537966.320
# ----------|----|--------|-----------|-----------|-----------|-----------|-----------|-------------|-------------|
Ddebug      5    74959    0.8969307   0.9403643   3.1490326   0.0204565   0.9403643   8507341.376   7443923.704
Doptimise   6    74959    0.8819103   0.9251911   2.4299622   0.0207769   0.9251911   8646862.484   7566004.674
# ----------|----|--------|-----------|-----------|-----------|-----------|-----------|-------------|-------------|
```

A.4.3. multi_pair.dat

```
# GNUplot data file
# HMF Transmitter TEST_MULTI_PAIR
# data derived from tag TEST_003_MULTI_PAIR/*optimize*.result

# Comparison of HMF Transmitter results using multiple optimized pairs
#
# An element is one buffer packet
# A measured package contains of 1,000 elements
# 10^6 packages were measured
# total buffer packets transmitted during test therefore 10^9

# descrip.| el       |                       ms                                | microsec  | KiB/s bandwidth      | cnt   |
# name| nb| BUF_SZ   | min        | avg        | max         | sd         | element   | buffer    | puls     | pairs |
# 1   | 2 | 3        | 4          | 5          | 6           | 7          | 8         | 9         | 10       | 11    |
# ----|---|----------|------------|------------|-------------|------------|-----------|-----------|----------|-------/
Hopt1   1  24989     0.8811951   0.9246600     3.6139488   0.0189712   0.9246600    8651829    7570350   1
Fopt1   2  49999     0.8800030   0.9286324     3.6070347   0.0202740   0.9286324    8614819    7537966   1
Dopt1   3  74959     0.8819103   0.9251911     2.4299622   0.0207769   0.9251911    8646862    7566005   1
# ----|---|----------|------------|------------|-------------|------------|-----------|-----------|----------|-------/
Hopt2   4  24989     0.9098053   1.6375465     4.3740273   0.0400216   1.6375465    9770746    8549403   2
Fopt2   5  49999     0.9121895   1.6384369     3.1960011   0.0283882   1.6384369    9765412    8544735   2
Dopt2   6  74959     0.8769035   1.6361372     4.3141842   0.0192134   1.6361372    9779131    8556740   2
# ----|---|----------|------------|------------|-------------|------------|-----------|-----------|----------|-------/
Hopt3   7  24989     0.9109974   2.2066788     4.8341751   0.5896226   2.2066788   11260334    9852793   3
Fopt3   8  49999     0.9119511   2.1991311    10.1079941   0.5903547   2.1991311   11301591    9888892   3
Dopt3   9  74959     0.8890629   2.2348204    10.1699829   0.6450868   2.2348204   10966058    9595300   3
# ----|---|----------|------------|------------|-------------|------------|-----------|-----------|----------|-------/
Hopt4  10  24989     0.9069443   2.8663166    75.7119656   0.8444089   2.8663166   11387325    9963909   4
Fopt4  11  49999     0.8900166   2.8359817    74.7711658   0.8632161   2.8359817   11517232   10077578   4
Dopt4  12  74959     0.9050369   2.8468287    17.8899765   0.8461096   2.8468287   11475476   10041041   4
# ----|---|----------|------------|------------|-------------|------------|-----------|-----------|----------|-------/
?
?
Dopt6  13  74959     1.0950565   4.2344990   154.3040276   2.9379332   4.2344990   11816924   10339808   6
# ----|---|----------|------------|------------|-------------|------------|-----------|-----------|----------|-------/
```

A.5. Acronyms

Vision(s)	Electronic Vision(s) Group
HMF	Hybrid Multiscale Facility
HICANN	High Input Count Analog Neural Network
HMFConvHw	HMF Conventional Hardware
NP	neuromorphic part
C++	C++ Programming Language
RDMA-NIC	RDMA capable network interface card
RDMA	Remote Direct Memory Access
FPGA	Field Programmable Gate Array
`BOOST`	BOOST C++ Libraries
`BOOST::Interprocess`	BOOST interprocess library
shamem	Shared-Memory Area
r-head	Receiver Read-head
g-head	Generator Write-head
sd	standard deviation
KiB	kibibyte: in this document a KiB is defined as $2^{10} = 1024$ bytes

Bibliography

Boost.CircularBuffer, Version 1.46.1 website, `http://www.boost.org/doc/libs/1_46_1/libs/circular_buffer/doc/circular_buffer.html`, 2011.

Boost.Interprocess, Version 1.46.1 website, `http://www.boost.org/doc/libs/1_46_1/doc/html/interprocess.html`, 2011.

BrainScaleS, Research, `http://brainscales.kip.uni-heidelberg.de/public/index.html`, 2011.

Brüderle, D., et al., A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems, *Biological Cybernetics*, *104*, 263–296, 2011.

Ehrlich, M., K. Wendt, L. Zühl, R. Schüffny, D. Brüderle, E. Müller, and B. Vogginger, A software framework for mapping neural networks to a wafer-scale neuromorphic hardware system, in *Proceedings of the Artificial Neural Networks and Intelligent Information Processing Conference (ANNIIP) 2010*, pp. 43–52, 2010.

FACETS M7-5, Verify that the layer-2 communication reaches the bandwidth requirements for a multi-wafer system, including the host communication via GBit-Ethernet, FACETS Milestone M7-5, UHEI and TUD, 2009.

Millner, S., A. Grübl, K. Meier, J. Schemmel, and M.-O. Schwartz, A VLSI implementation of the adaptive exponential integrate-and-fire neuron model, in *Advances in Neural Information Processing Systems 23*, edited by J. Lafferty et al., pp. 1642–1650, 2010.

OpenFabrics, Website, `http://www.openfabrics.org/index.php?option=com_content&view=article&id=3`, 2011.

Wendt, K., M. Ehrlich, and R. Schüffny, GMPath - a path language for navigation, information query and modification of data graphs, in *Proceedings of the Artificial Neural Networks and Intelligent Information Processing Conference (ANNIIP) 2010*, pp. 31–42, 2010.