Visualising DMA Operations for Improved Parallel Performance

Paul Keir
Codeplay Software Ltd.

MMNet Workshop
Heriot Watt, May 2013

Presentation Outline

- Introduction
- EU Framework 7 Project: LPGPU
- Offload C++ for PS3
- Memory Access and its Visualisation
- Case Study: Interactive IK Animation

Codeplay Software Ltd.

- Based in Edinburgh, Scotland
- Incorporated in 1999
- 25 full-time employees
- Compilers, optimisation and language development
  - GPU and heterogeneous architectures
  - Increasingly mobile and embedded CPU/GPU SOCs
- Commercial partners:
  - Qualcomm, Movidius, AGEIA, Fixstars, Sony

Codeplay Software Ltd.

- Member of two 3-year EU FP7 research projects
  - Peppher and LPGPU

- Sony-licensed PS3 middleware provider
- Contributing member of Khronos group

  - Working towards OpenCL 2.0
  - OpenCL-HLM (High Level Model) Working Group
  - Chaired by Codeplay CEO Andrew Richards

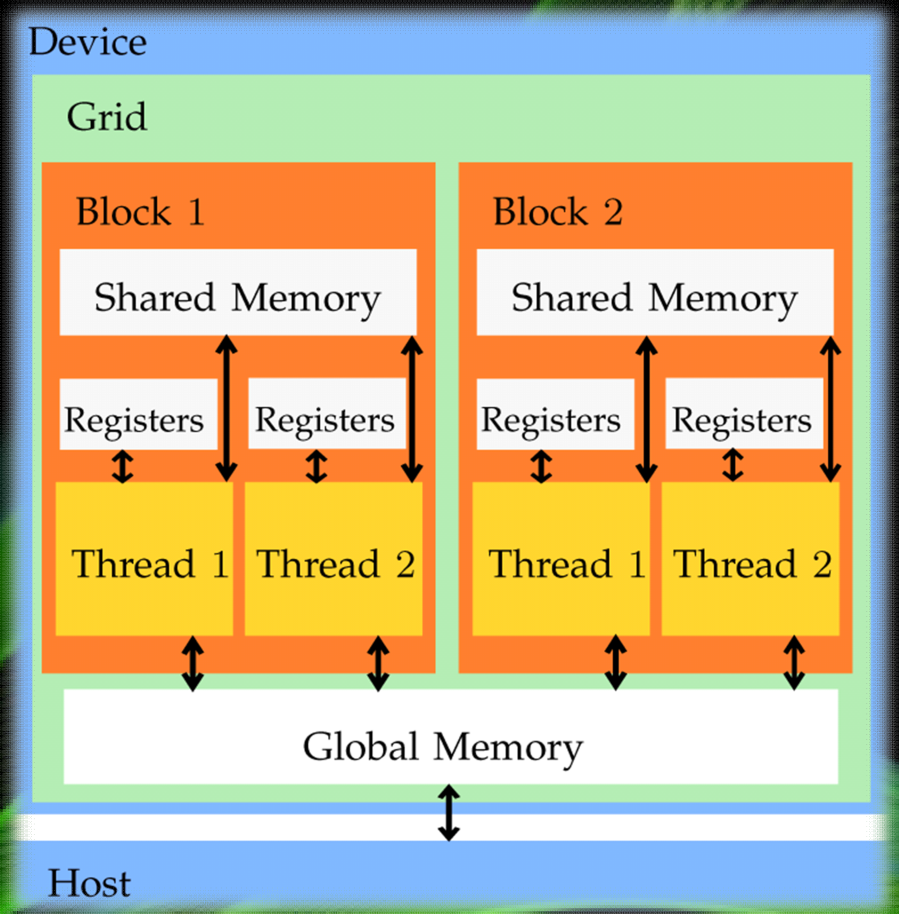
- Member of the HSA Foundation
  - HSA System Runtime Working Group
  - Also chaired by Codeplay's CEO

Low-power GPU (LPGPU)

- EU Framework 7 Project
- Research into low-power and mobile GPU tech.
- Developing hardware, software and tools
- Six Partners
  - Four Companies
    - Codeplay, Geomerics, AiGameDev, Think Silicon
  - Two Universities
    - TU Berlin, Uppsala (Sweden)

Distinct Memory Spaces

- Addressable caches
  - Reduce memory latency
- 4 OpenCL memory spaces
  - private, local, constant, global
- Embedded C
- PGAS Languages
  - Chapel, Fortress, X10



Single-source GPU Programming

- C++ AMP, CUDA, Offload C++, OpenCL-HLM
- Pragma-based:
  - OpenMP 4.0, OpenACC, Open-HMPP, SMPSs
- Facilitates rapid porting of existing software
- Can allow serial and parallel codes to coexist
- May come as part of an integrated package
- Concise: examples fit on one slide!
- n.b. Host merely adds one additional memory space
  - OpenCL C has 4 address spaces (already)

Offload C++ for PS3

```
void process_data(double *data, size_t len) {
  for (int i = 0; i < len; ++len)
    data[i] += 1.0;
}

void f1(double *data, size_t len)
{
  {
    process_data(data,len);
  }

}
```

Offload C++ for PS3

```
void process_data(double *data, size_t len) {
  for (int i = 0; i < len; ++len)
    data[i] += 1.0;
}

void f1(double *data, size_t len)
{
  offload {
    process_data(data,len);
  }

}
```

- Code within an Offload block runs on the accelerator
- Functions automatically compiled for host and accelerator

Offload C++ for PS3

```
void process_data(double *data, size_t len) {
  for (int i = 0; i < len; ++len)
    data[i] += 1.0;
}

void f1(double *data, size_t len)
{
  auto t = offload {
    process_data(data,len);
  };
  offloadThreadJoin(t);
}
```

- Code within an Offload block runs on the accelerator
- Functions automatically compiled for host and accelerator
- Asynchronous calls may also join outwith function scope

Automatic Call Graph Duplication

```
void bar2(double *, size_t) {}
void bar3(double *, size_t) {}

void bar1(double *data, size_t len) {
  bar2(data,len);
  bar3(data,len);
}

void f2(double *data, size_t len) {
  offload {
    bar1(data,len);
  };
}
```

- Entire function call graph is duplicated
- Implicitly rooted by each Offload block

Offload C++ Address Spaces

- Each pointer or address has an implicit *locality*
  - Corresponding to a zero-based index
  - Derived from an enumeration of system memory spaces
- Dereferencing a pointer implicitly moves data
  - Between device memory banks
  - Host ↔ Device transfer (in a dual-space system)
- C++ type system extended for address-locality
  - Pointer assignment between distinct localities prohibited
  - Overloading based on pointer locality
  - Template metaprogramming and type-trait compatible

Implicit Locality

```
void f3(double *data, const size_t len)
{
  offload {
          double *po = data;
          double d   = *po;
          double *pi = &d;
    *pi = *pi + 1.0;
    *po = *pi;
  };
}
```

- Pointer “po” has *implicit* outer attribute
- Pointer “pi” has *implicit* inner attribute
- Static locality-typing catches errors early; e.g. `po = pi;`
- Result: 1st element of “data” array is incremented by one

Explicit Locality

```
void f3(double *data, const size_t len)
{
  offload {
    outer double *po = data;
          double d   = *po;
    inner double *pi = &d;
    *pi = *pi + 1.0;
    *po = *pi;
  };
}
```

- Pointer “po” has *explicit* outer attribute
- Pointer “pi” has *explicit* inner attribute
- Static locality-typing catches errors early; e.g. `po = pi;`
- Result: 1st element of “data” array is incremented by one

Overloading Pointer Locality

```
void reverse_copy(double *curr, double *last, double *res) {
  while (curr != last) {
    *res++ = *curr++;
  };
}

void f4(double *data, const size_t len)
{
  offload {
    double rev_data[len];
    reverse_copy(data,     data+len,     rev_data);

  };
}
```

- Implicit overload for *each* pointer argument's locality
- *pow(nspaces,nargs)* potential overloads for each function
  - Created automatically according to function arguments

Overloading Pointer Locality

```
offload void
reverse_copy(outer double *, outer double *, inner double *);



void f4(double *data, const size_t len)
{
  offload {
    double rev_data[len];
    reverse_copy(data,     data+len,     rev_data);

  };
}
```

- Assume `reverse_copy` provides an optimised implementation
- The `offload` function qualifier permits further overloading

Overloading Pointer Locality

```
offload void
reverse_copy(outer double *, outer double *, inner double *);

offload void
reverse_copy(inner double *, inner double *, outer double *);

void f4(double *data, const size_t len)
{
  offload {
    double rev_data[len];
    reverse_copy(data,     data+len,     rev_data);
    reverse_copy(rev_data, rev_data+len, data);
  };
}
```

- Assume `reverse_copy` provides an optimised implementation
- The `offload` function qualifier permits further overloading

Performance of Ported Code

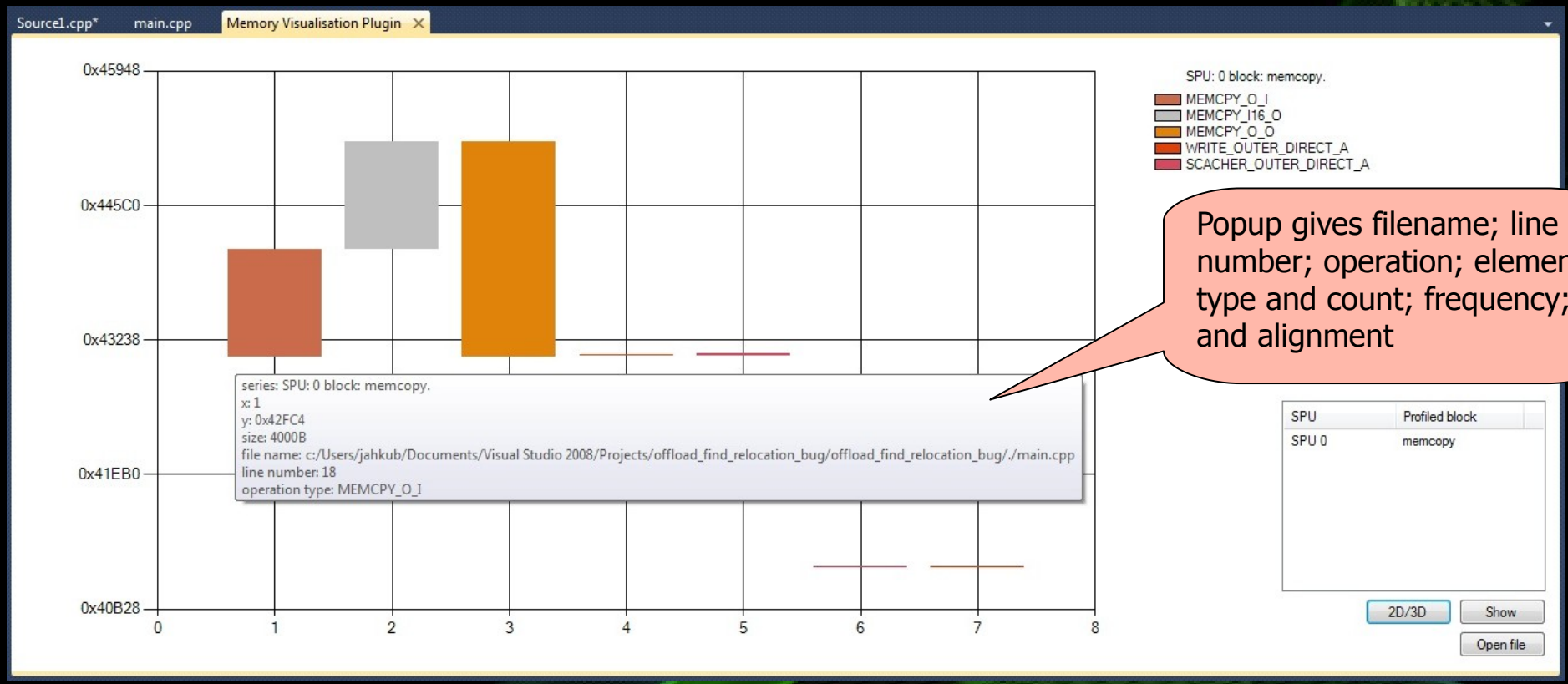
- Rapid porting of code to heterogeneous architectures
- NASCAR The Game 2011 powered by Offload
- Fetching operands costs more (energy) than computing on them
  - Moving a word across die: 10 Fused Multiply-Adds (FMAs)
  - Moving a word off-chip: 20 FMAs
- A need to “lift the hood” on data movement
  - Does the code below involve a DMA transfer?


```
void zod(double *p1, double *p2) {
  *p1 = *p2;
}
```

Logging Memory Operations

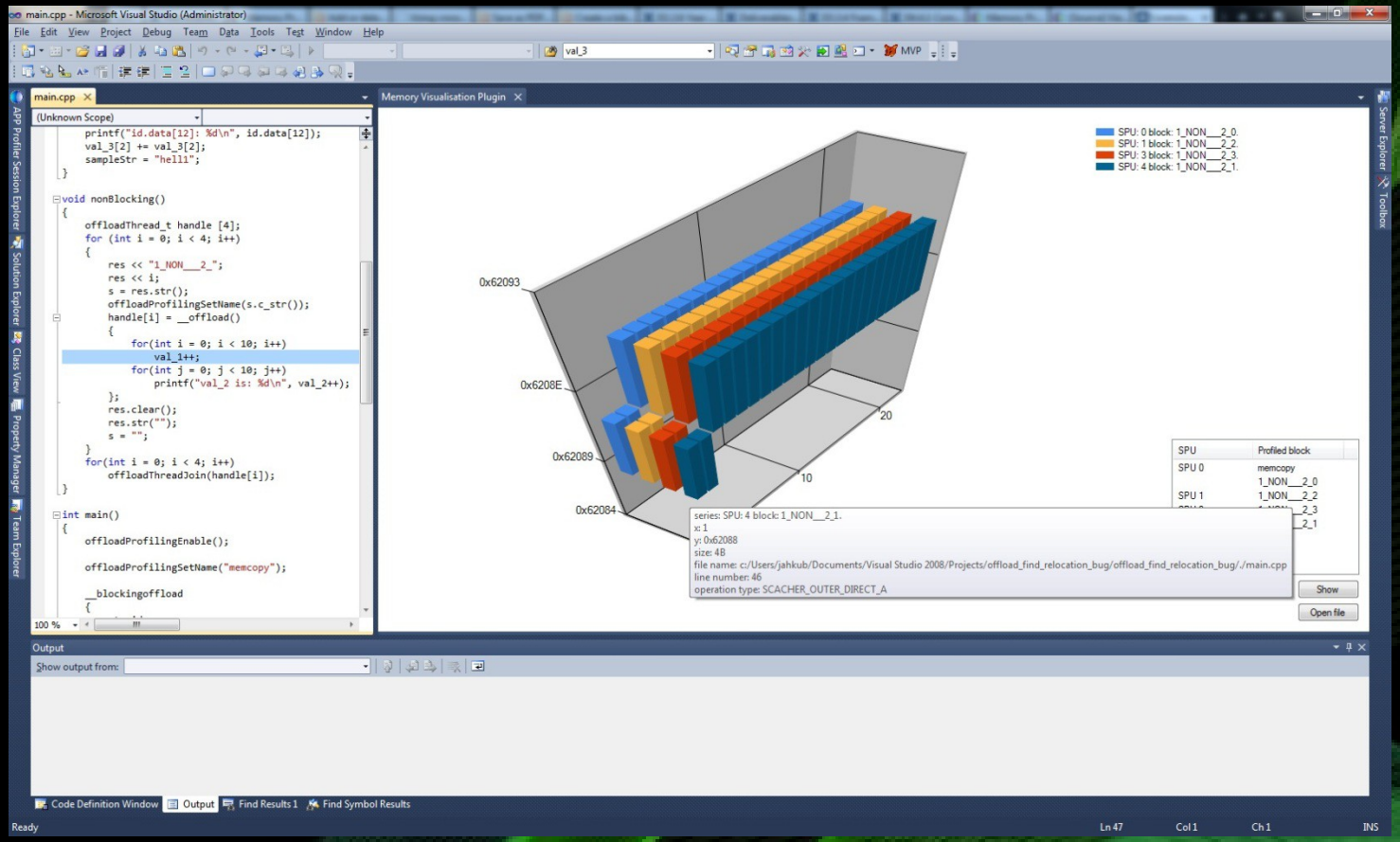
- Instrument code to record memory operations
- Simple C API interface
- Initial development platform: Offload C++ on PS3
- Quantity of data logged is ~10MB per kernel thread
- Logging of all off-chip data accesses will store:
  - Access type (read/write/copy)
  - Element datatype (integer/float/char/pointer)
  - Element count (data size)
  - Location (file name and line number)
  - Frequency (control path)
  - Alignment (for copies only)

MSVS Visualisation Plugin



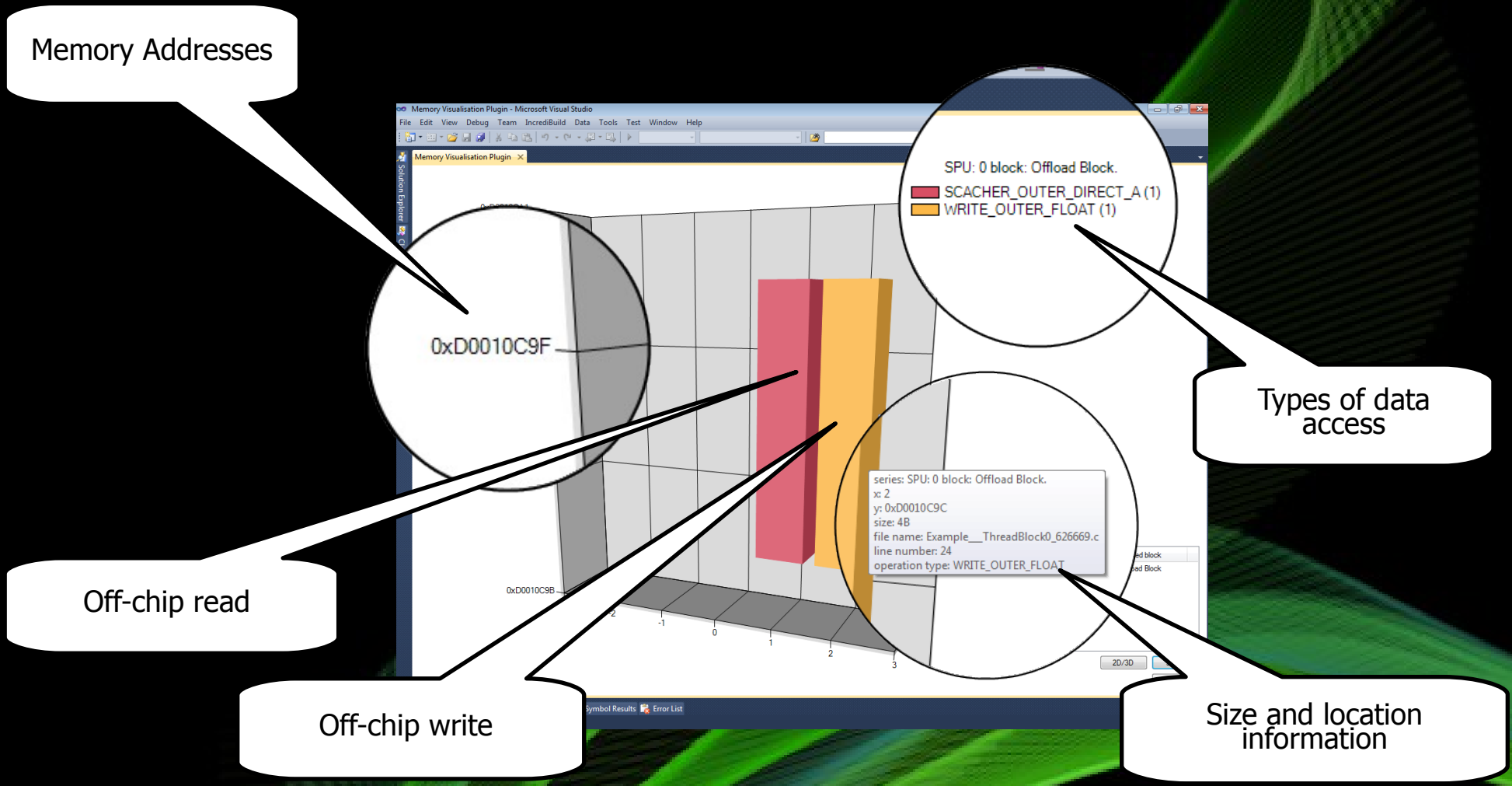
- X-axis: memory operations ordered by time
- Y-axis: memory address range on host
- Z-axis: multiple threads

Integrated Development



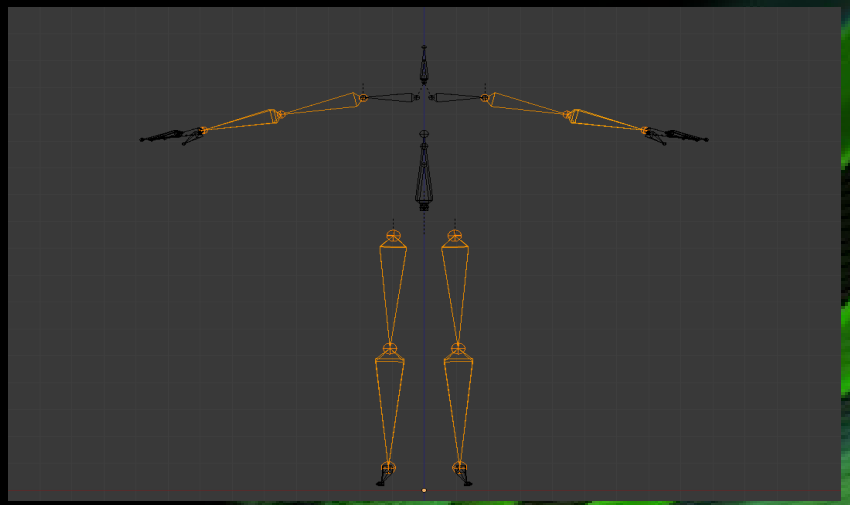
- Clicking a data point on the bar graph
  - Highlights the line of code issuing the memory operation
- 3D plots useful for multiple threads

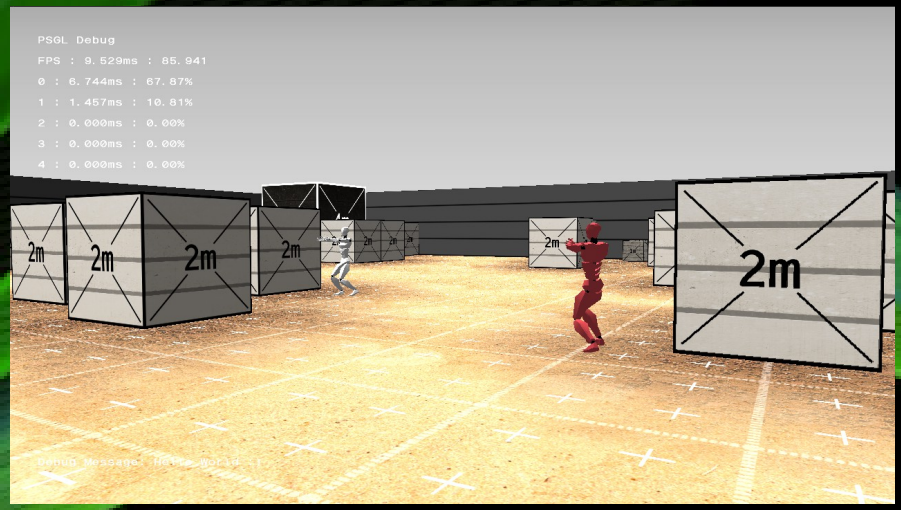
Integrated Development



Project Overview

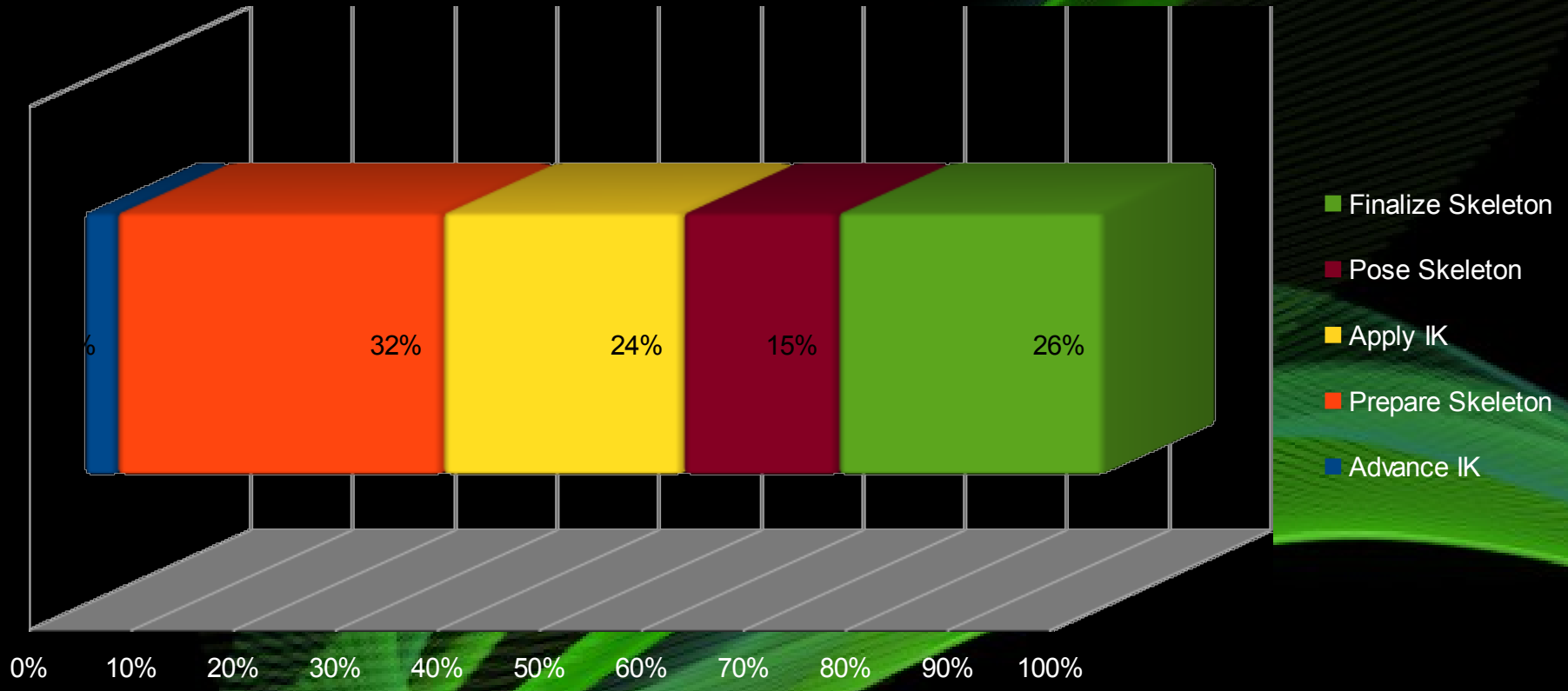
- Accelerate AIGameDev animation system using OffloadPS3
- Run across all available SPUs via Sony GameOS (i.e. 5)
- Analyse using our Memory Access Profiler
- Improve performance and power consumption
- Learn more about data organisation/movement in large app.
- Apply what we learn to future GPU hardware development





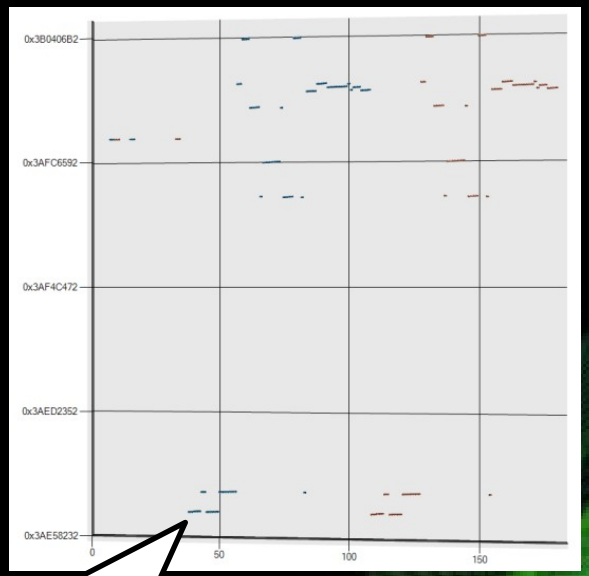
AiGameDev Animation Module

- Closed form two-bone IK algorithm
- Performing leg cycles and hand aiming
- Separate animation component for each actor

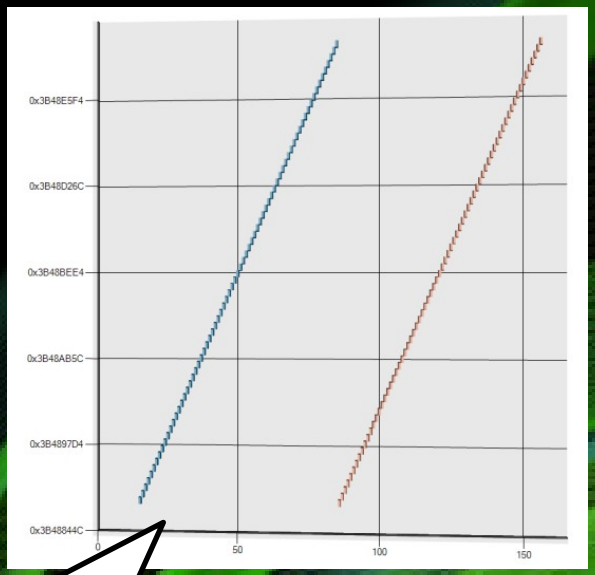


Iterative Performance Improvement

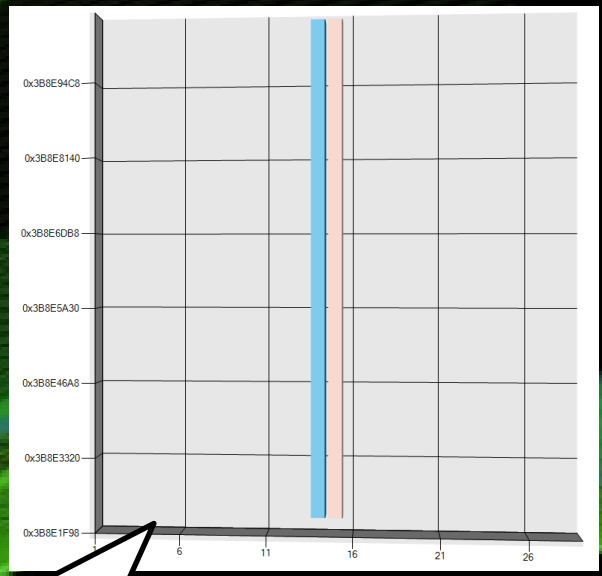
- Refactored to ensure skeleton data structure stored contiguously
- Reduced off-chip DMA transfers
  - From 142 small accesses, to 2 large accesses
- 7.5x Performance Improvement for Animation Component



Multiple accesses to fragmented structure

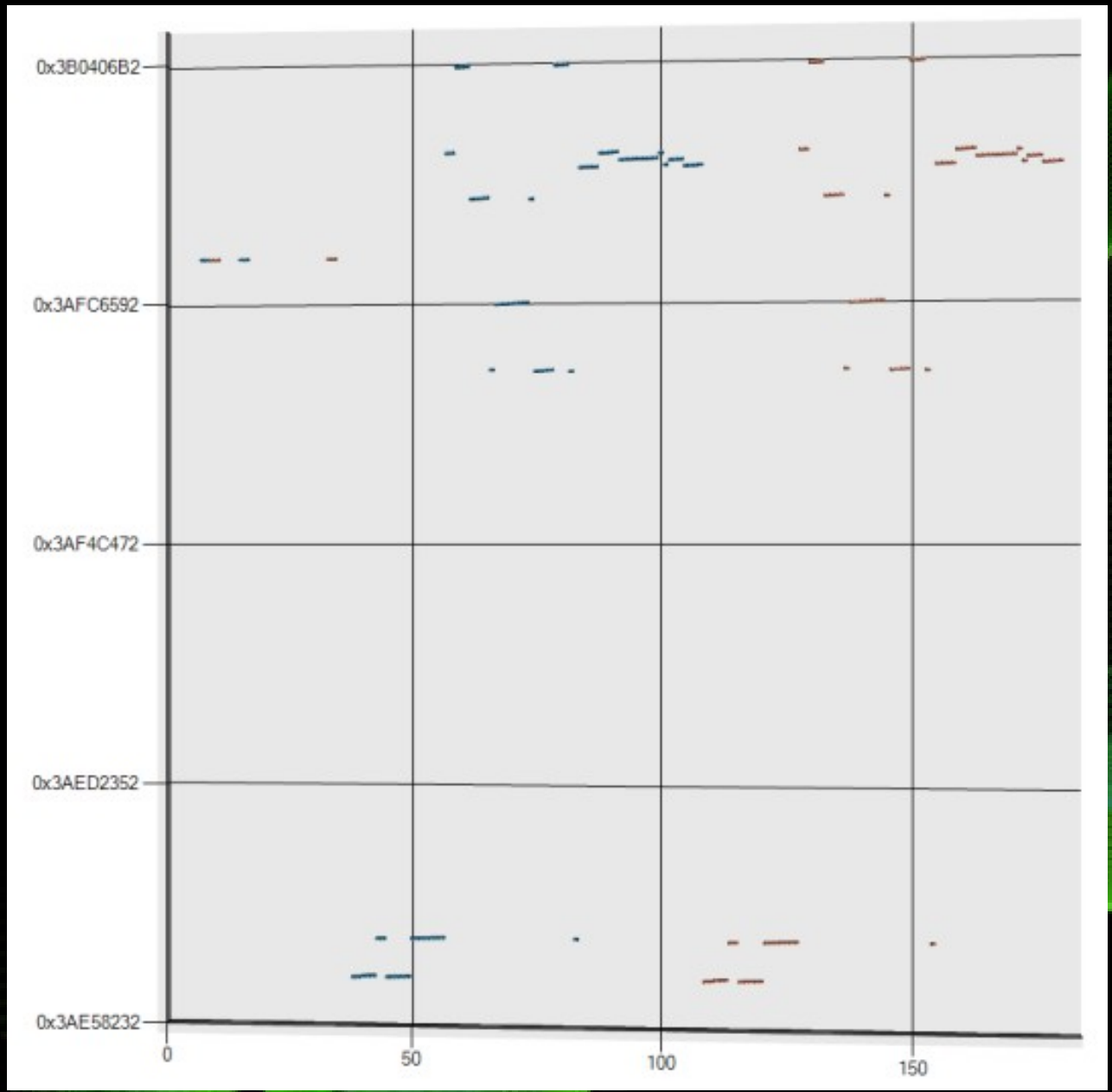


Multiple accesses to contiguous structure

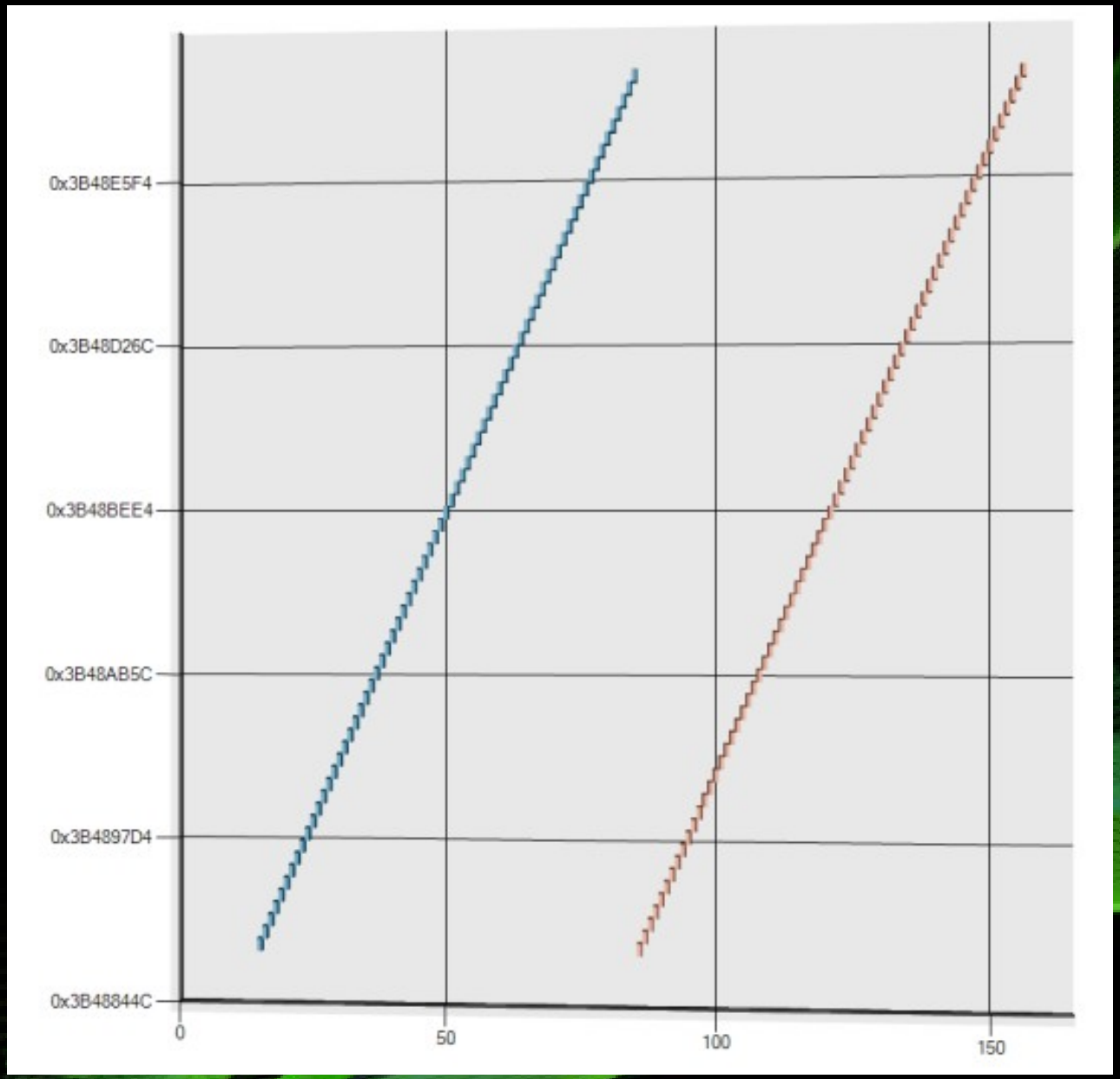


Single access to contiguous structure

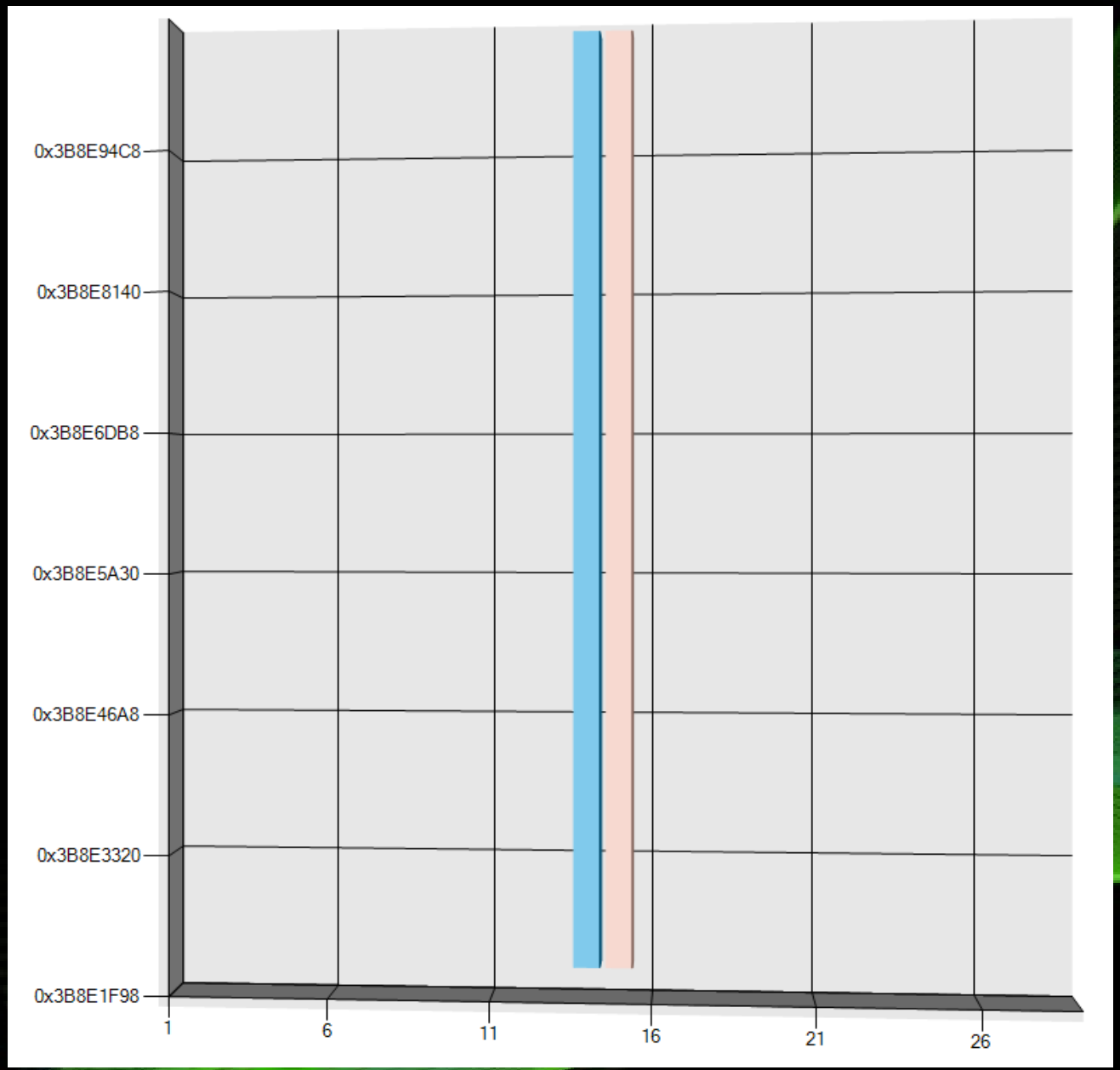
Unmodified Memory Accesses



Multiple Contiguous Accesses



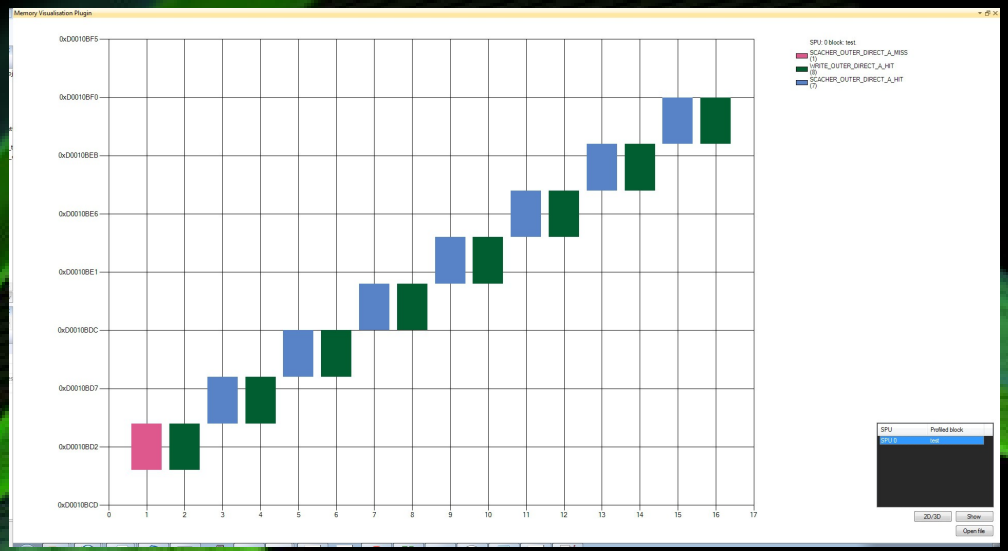
Single Large Access



Visualising Cache Activity

```
int main()
{
  int x[8];

  offloadProfilingEnable();

  offload {
    int i;
    for (i = 0; i < 8; i++)
    {
      int y = x[i];
      x[i] = y;
    }
  }

  offloadProfilingDisable();

  return 0;
}
```

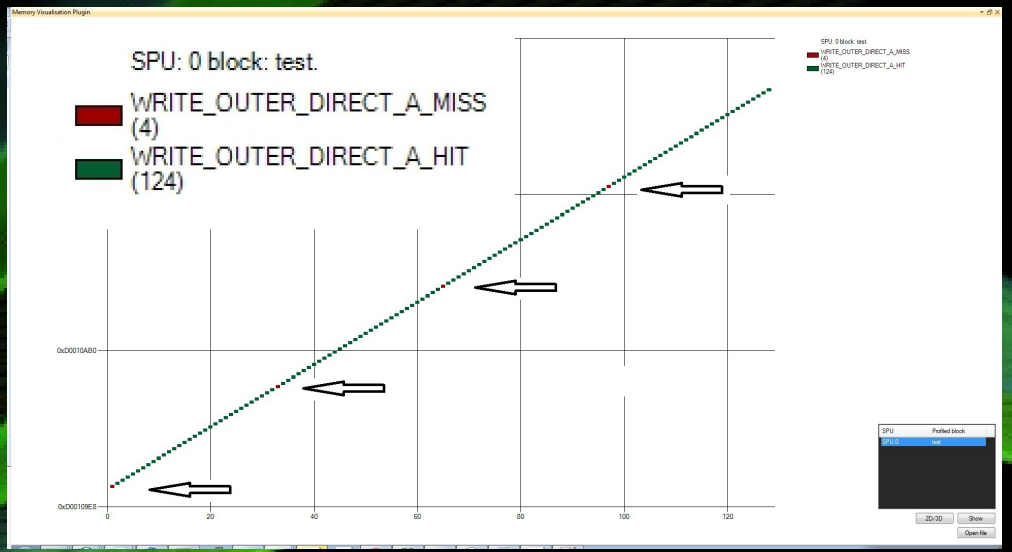
- Sony PS3 software cache
- Visualisation key:
  - Pink bar is cache read miss
  - Green is cache write hit
  - Blue is cache read hit



Revealing Cache-line Size

```
int main()
{
  int x[128];

  offloadProfilingEnable();

  offload {
    int i;
    int y = 0;
    for (i = 0; i < 128; i++)
    {
      x[i] = y;
    }
  }

  offloadProfilingDisable();

  return 0;
}
```

- 128 Byte cache-line
- Arrows at cache read miss
  - Every 32 4-Byte DMA reads



Conclusion

- Presented a memory access profiler for heterogeneous systems
- Ongoing work will target the tool towards GPU architectures
- Expect to see increased use of implicit address spaces
  - Especially in new GPU-oriented languages
  - Shared Virtual Memory (SVM) expected on upcoming GPUs
  - HSA architecture anticipated for Sony PS4