TAMPERE UNIVERSITY OF TECHNOLOGY

Offloading C++17 Parallel STL on System Shared Virtual Memory Platforms

Pekka Jääskeläinen (1&2), John Glossner (4&5),
Martin Jambor (3), Aleksi Tervo (1), and Matti Rintala (1)

(1) Tampere University of Technology, Finland
(2) Parmance(.com), Tampere, Finland,
(3) SUSE, Prague, Czech Republic
(4) University of Science and Technology, Beijing, China
(5) Optimum Semiconductor Technologies, Tarrytown, NY, USA

In OpenSuco 3 Workshop (ISC18)
Frankfurt, June 28 2018

Outline

- C++17 Parallel STL
- System Shared Virtual Memory
- Proof-of-Concept implementation
- Preliminary results

C++17 PARALLEL STL

C++17 ”Parallel STL” (PSTL)

- C++17 standard’s Section 28.4.: “Parallel Algorithms” adds *execution policies* to the *algorithm* library of the Standard Template Library (STL)
- Policies allow the programmer to state execution safety:
  - std::execution::seq – sequenced_policy: only execute serially in the calling thread
  - std::execution::par – parallel_policy: execution may be ”parallelized” (multithread)
  - std::execution::par_unseq – parallel_unsequenced_policy: execution may be vectorized (interleaved within a single thread) and multithreaded

```
sort(std::parallel::par, std::begin(input), std::end(input));
```

The TS provides parallelized versions of the following 69 algorithms from <algorithm>, <numeric> and <memory>:

Standard library algorithms for which parallelized versions are provided [Collapse]

- std::adjacent_difference
- std::adjacent_find
- std::all_of
- std::any_of
- std::copy
- std::copy_if
- std::copy_n
- std::count
- std::count_if
- std::equal
- std::fill
- std::fill_n
- std::find
- std::find_end
- std::find_first_of
- std::find_if
- std::find_if_not
- std::generate
- std::generate_n
- std::includes
- std::inner_product
- std::inplace_merge
- std::is_heap
- std::is_heap_until
- std::is_partitioned
- std::is_sorted
- std::is_sorted_until
- std::lexicographical_compare
- std::max_element
- std::merge
- std::min_element
- std::minmax_element
- std::mismatch
- std::move
- std::none_of
- std::nth_element
- std::partial_sort
- std::partial_sort_copy
- std::partition
- std::partition_copy
- std::remove
- std::remove_copy
- std::remove_copy_if
- std::remove_if
- std::replace
- std::replace_copy
- std::replace_copy_if
- std::replace_if
- std::reverse
- std::reverse_copy
- std::rotate
- std::rotate_copy
- std::search
- std::search_n
- std::set_difference
- std::set_intersection
- std::set_symmetric_difference
- std::set_union
- std::sort
- std::stable_partition
- std::stable_sort
- std::swap_ranges
- std::transform
- std::uninitialized_copy
- std::uninitialized_copy_n
- std::uninitialized_fill
- std::uninitialized_fill_n
- std::unique
- std::unique_copy

Source: http://en.cppreference.com/w/cpp/experimental/parallelism

Element Access Functions

- Some of the algorithms call user-defined functionality via
  - Custom comparison predicates
  - Custom iterators
  - Custom operation functions (algorithm specific), etc.
- These are collectively called ”Element Access Functions” (EAF) in the standard
  - Must be compiled to offload targets' ISA

```
std::transform(std::par, s.begin(), s.end(), s.begin(),
                [](unsigned char c) -> unsigned char {
                    return std::toupper(c);
                });
```

SYSTEM SHARED VIRTUAL MEMORY

Data Transfers in Heterogeneous Platforms

- Until recent years, heterogeneous platform programming required **explicit data transfers**
- Buffers *fully copied* between the host and the accelerator's memory before executing a kernel in the device and results moved back after finishing
- Not compatible with C++ where data is routinely passed around via pointers without size information available in the general case

System Shared Virtual Memory

- Recently, major device vendors and programming standards with ”shared virtual memory” features have appeared
  - Different degrees of support
  - The most powerful ones make the process VM ”look and feel” in terms of data identical to both the CPU and the accelerator
- Benefits:
  - More efficient handling of large sparse data sets
  - Can save useless copying: If only a subset of the transferred buffer is actually accessed by the device
- Drawbacks:
  - Requires hardware and/or operating system support
  - The usual issues with dynamic block/page migration schemes: False sharing, other ping-pong overheads, syscall overheads if OS needed, …

Devices with SSVM-Like Capabilities

- AMD APUs:
  - Cache coherence between the CPU and GPU on the same SoC
- NVIDIA Unified Memory
  - Previously only via a ”managed buffer allocation API”, since Pascal also via malloc() if OS support available
- Convey Computer’s (now of Micron) FPGA card

Major APIs for Heterogeneous SSVM

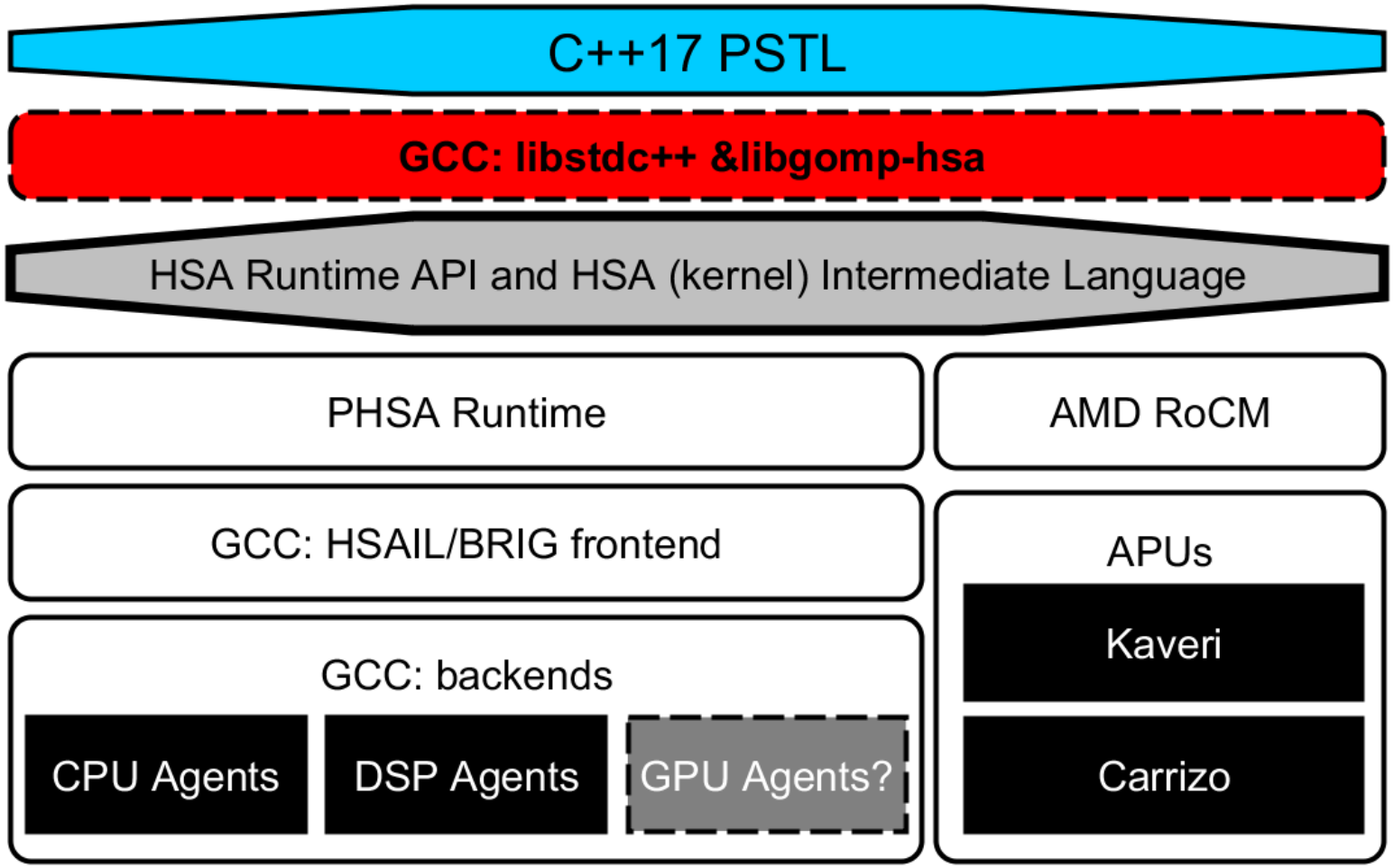
- Khronos OpenCL since 2.0:
  - Allows various granularities for SVM implementations
  - Fine-Grained System SVM allows referring to any host memory without allocation via special allocation functions
- Heterogeneous System Architecture (HSA)
  - ”Base profile” requires special memory management
  - ”Full profile”: similar to OpenCL 2.0 FGSSVM -- no special allocation needed
    - **Used in the presented proof-of-concept**

Heterogeneous Systems Architecture (HSA)

- Standard for shared virtual address space heterogeneous platforms
- Multiple specifications describing HSA-compliant systems and their programming
  - Platform System Architecture: Hardware requirements for HSA compliant platforms
  - Programmer Reference Manual: Kernel intermediate language HSAIL
  - Runtime Specification: Host-side software layer
- Freely available: http://www.hsafoundation.com/standards/

PROOF-OF-CONCEPT IMPLEMENTATION

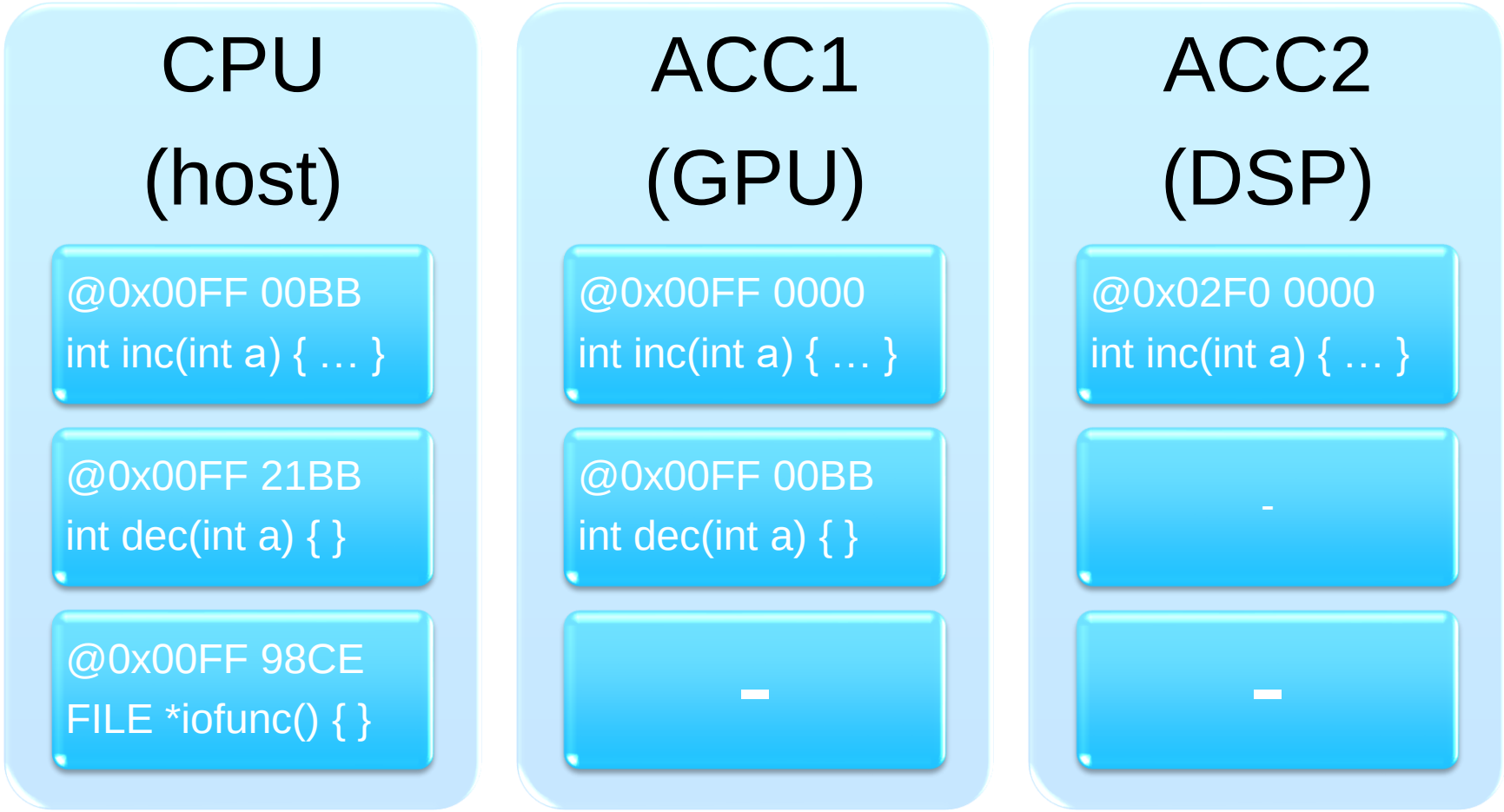
PoC Components



PSTL on Heterogeneous Platforms: Challenges

- For data management for this PoC, we assume SSVM level support is available in the platform
  → Assume data transfers and allocation as defined by C++ ”just work”
- Remaining problems stem from needing to support multiple instruction-set architectures

Multiple ISAs: Multiple Versions of EAFs with Multiple Runtime Addresses



Multiple ISA Challenges:

1) Binary File Portability

- ”Fat binary” enables portability across multiple heterogeneous devices. It includes
  1) Native ISA binary for the whole program,
  2) The **offloadable functions** in an intermediate language (IL) that can be compiled **at runtime** to any supported accelerator available in the platform
- We reused the GCC libgomp’s ELF binary format with custom sections that store HSAIL
- HSA Intermediate Language
  - A RISC-like virtual ISA with vector support and a lot of registers, and a few ”special instructions”
  - Both backend and frontend support in upstream GCC!

Multiple ISA Challenges:

2) What to Compile to IL?

- For offloading PSTL algorithm calls, we need to include
  - The predefined (templated) algorithm implementation
  - The programmer defined **element access functions** and the functions they call (recursively)
- PSTL calls are just templated function calls without any special ”outlining” to **explicitly isolate** the offloaded functionality
  - Recursively statically **analyze the call graphs** starting from detected PSTL calls and include them in IL?
- PoC implementation currently the simplest possible that comes to mind:
  - *Tries* to compile **all functions** to both native and HSAIL when -fpar_offload switch given in the command line
  - Silently drops functions with (currently) unsupported features such as exception handling or syscalls

Multiple ISA Challenges:

3) Function Pointers

- **Function pointers are just data in C/C++**
  - At some point in time function calls can be made through them
  - C++ virtual function calls end up being function pointer calls
- Heterogeneous devices have separate instruction memories
- In C++ offloading, each different ISA in the platform has their own **copy of the potentially offloaded function** in their own instruction address spaces
  - Different versions can have **overlapping addresses**, the copies are thus not identifiable via unique function pointer values
  - What to store as a function pointer value?
- How to implement function pointer calls so correct function address is used when calling inside the accelerator and when calling inside the CPU? **How to make it efficient?**

Function Pointers: Current PoC Implementation

- Insight: At the PSTL **kernel call time**, we know (some of the) function pointer values and the kernel can be recompiled **at run time**
- Convert (marked) function pointer calls to direct calls via **HSAIL specialization**
- No generic runtime function pointer resolution implemented at the moment

  →If there are fpointer references remaining in the offload candidate function, it won't get compiled to HSAIL
- Means that virtual calls are not supported
  - The main shortcoming of the current PoC

```
template<typename ElementType, typename ResultType,
         typename FuncRetType, typename FuncArgType>
static void __attribute__((hsa_kernel))
__transform_array (void *args) {

  size_t i = __builtin_hsa_workitemflatabsid ();

  ResultType *d_first = *((ResultType**)args + 0);
  ElementType *first1 =  *((ElementType**)args + 1);

  d_first[i] =
    _FUNC_PLACEHOLDER <FuncRetType, FuncArgType> (first1[i]);
}
```

Fig. 2. Templated HSA kernel implementation of *transform* with an indirect call specialization placeholder.

par_offload execution policy

- In terms of *correctness*, we could not find any reason why we could **not** (try to) offload all *par_unseq* policy calls on top of HSA Full profile
- However, offloading always incurs additional overheads
- Added a new execution policy **par_offload** to allow programmers to choose when offloading is known safe and potentially beneficial

```
std::transform(std::experimental::execution::par_offload,
               pixel_data.begin(), pixel_data.end(),
               pixel_data.begin(),
               [](char c) -> char {
                  return c * 16;
               });
```

Fig. 1. Example of offloading *transform* with the function to execute for each element defined as a lambda expression.

Current Status

- Source code is available for hacking in http://github.com/parmance/par_offload
- A preliminary proof-of-concept with all the ”puzzle pieces” working and a handful of APIs implemented
- A lot of room for optimization and improvement

PRELIMINARY RESULTS

Benchmark Setup

- Tested the performance of offloading using *par_offload*
  - To get initial proofs that offloading PSTL over SSVM with the PoC can be beneficial
  - **No major optimization / profiling done yet**
- Platform: AMD Carrizo APU
  - SoC with cache coherence between CPU and GPU
- Baseline implementation: Intel's open source PSTL implementation
  - For x86-64/SIMD CPUs (seq, par_unseq tested)

Results

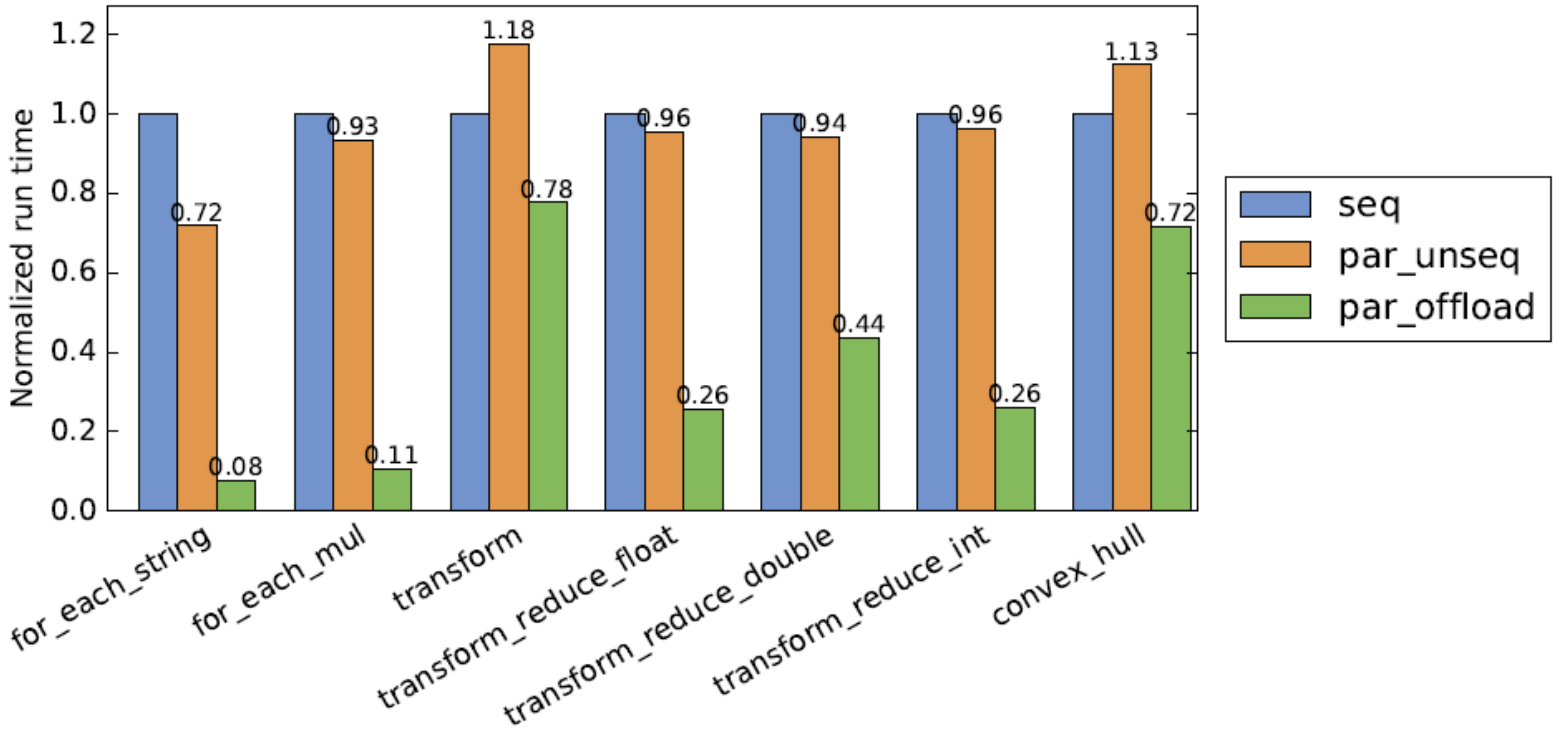


Fig. 3. Wall clock execution times, normalized to Intel PSTL seq execution policy.

Result Insights

- Non-offloading **seq** policy that executes in the caller thread has a major benefit of optimizing the PSTL call with the surrounding code in the call site
  - Improved inlining, pointer alias analysis, constant propagation etc.
  - Can still use autovectorization normally to utilize SIMD extensions in the CPU
- Par_unseq and par_offload isolate the algorithm to thread functions / kernels
  - CPU threads for par&par_unseq and HSA kernel packets for par_offload
- Room for more compiler research here

Summary

- C++17 introduced parallel algorithms with three *execution policies* that the programmer can use to control the degree of parallelization safety of the STL call
- System Shared Virtual Memory platforms present a coherent view of a single virtual address space to the whole heterogeneous platform
- We presented a proof-of-concept implementation of C++17 offloading in SSVM platforms
  - HSA Full profile runtime, GCC and libstdc++
- Key issues still to tackle:
  - More general handing of ”universal multi-ISA function pointers”, e.g., with a more general runtime function pointer specialization
  - Maintaining context information in the ”outlined” offloaded functions to help optimizations

Acknowledgements

The presented work could not have been done without financial support from…

General Processor Technologies

https://www.generalprocessortech.com/

HSA Foundation

http://www.hsafoundation.com/

Academy of Finland

http://aka.fi

Thank you!

Thanks for your interest!

My email:
pekka.jaaskelainen@tut.fi

“par_offload” proof-of-concept sources available:

http://github.com/parmance/par_offload