VectorPU: A Generic and Efficient Data-container and Component Model for Transparent Data Transfer on GPU-based Heterogeneous Systems

Lu Li, Christoph Kessler

Linköping University, Sweden

lu.li@liu.se, christoph.kessler@liu.se

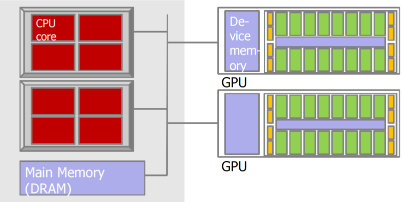
Outline

Motivation

- GPU-based systems, CUDA
- ☹ 82% of code [Jablin et al.]: data allocation and coherence

- Explicit data movement is outdated
  - ☺ Nvidia's unified memory feature already make it unnecessary

- ☹ However it is not efficient with page-based data transfer.



(a) source: compu poster

(b) source: gigaom.com

Motivation: Previous Work

- Compiler-only approach
  - ☺ may not need annotations
  - ☹ can not capture runtime information
  - ☹ pure compiler analysis can lead to redundant transfer [Pai et al.]
  - e.g. OpenMPC, eager C2G transfer [Pai et al.]
- Run-time approach
  - ☺ capture runtime information
  - ☹ require programmers' annotations or low level support
  - ADSM, DyManD and Nvidia's unified memory: page-based
  - SemCache and SemCache++: matrix abstraction, less generic
  - SkePU's smart container and StarPU: only for their host software package.

  - VectorPU: generic, efficient vector abstraction.

Motivation

- Hybrid compiler/runtime approach
  - ☺ combine compiler and runtime
  - state-of-art: Ishizaki et al.
  - ☹ source code may not be available: cuFFT, cuBlas, cuSparse etc
- VectorPU: work nicely with binary libraries
  - no need for programming of data transfer considering performance and programmability.
  - only require annotations.

Flow Signature

- Annotations for access properties (read, write, or readwrite) can help to decide data movement

```
GR,GW,GRW,NA
```

- DSEL
- Flow signature:
  - a tuple of annotations for every parameter (or argument) in a function signature

```
#define bar_flow (GR)(GW)(GRW)(NA)
__global__
void bar(const float *x, float *y, float *z, int size){
  ...
}
```

Annotation DSEL

- An annotation for one argument or parameter:

[where] + access property + [end position] + [iterator]

Access Property	Host	Device
Read pointer	R	GR
Write pointer	W	GW
Read and write pointer	RW	GRW
Read iterator	RI	GRI
Read end iterator	REI	GREI
Write end iterator	WEI	GWEI
Read and write iterator	RWI	GRWI
Read and write end iterator	RWEI	GRWEI
Not Apply	NA	NA

Table 1: VectorPU's annotation for a parameter

Flow signature

- α signature: at call site, simplest, no reuse

```
bar(GR(x), GW(y), GRW(z), size);
```

- β signature: at declaration site, allow reuse, allow multiple flow signatures defined

```
//——VectorPU Component Definition———
#define bar_flow (GR)(GW)(GRW)(NA)
__global__
void bar(const float *x, float *y, float *z, int size){
  ...
}
```

- γ signature: at declaration site, allow reuse, clean interface, require our `vpucc` compiler

```
__global__
void bar(const float *x[[GR]], float *y[[GW]], float *z[[GRW]],
    int size){
  ...
}
```

VectorPU component

- Any function with its call or definition annotated by flow signature (α, β, or γ) is called *VectorPU component*
- Call VectorPU component with α signatures

```
bar<<<32,256>>>(GR(x), GW(y), GRW(z), N)
```

- Call VectorPU component with β or γ signatures

```
CALL( (bar)((<<<32,256>>>))((x,y,z,N)) );
CALLC( (bar)((<<<32,256>>>))((x,y,z,N)) (bar_flow_non_default) );
```

VectorPU vector

- *VectorPU vector*: represent data at host and device side simultaneously.
- Manage a array, can be dynamically resized

```
template <class T, class Index_Type=std::size_t>
struct vector : public std::vector<T>, public thrust::device_vector<T>;
```

- Motivating Example:

```
#define bar_flow (GR)(GW)(GRW)(NA)
__global__
void bar(const float *r, float *s, float *t,
         const int size) { ... }

int main(){
  vectorpu::vector x(10), y(10), z(10);
  CALL( (bar) ((<<<32,256>>>)) ((x,y,z,10)) );
  }
```

Coherence Management

- We use a simplified MSI coherence mechanism
- Only two states: invalid (not most recent copy) or valid (most recent copy)
- Shared state not needed, because VectorPU vector start from shared state and keep shared.
- Common in the context of heterogeneous computing, leads to simpler coherent algorithm

Coherence Management

- Coherence algorithm (device), constant time complexity:

```
//GR:
if(coherent_flag_GPU != valid)
{
  copy_CPU_to_GPU();
  coherent_flag_GPU = valid;
}
return pointer;

//GW:
coherent_flag_GPU = valid,
coherent_flag_CPU = invalid;
return pointer;

//GRW
//the same as GR
coherent_flag_CPU = invalid;
return pointer;
```

Expressiveness of VectorPU

- Basic and Customized Data Type
- Smart Iterator
- Lambda Functions
- VectorPU Algorithms
- Overloaded Functions
- Template Functions
- Express Skeletons in Skeleton Programming
- Flow Signature Switch

Basic and Customized Data Type

```
vectorpu::vector<int> x;
vectorpu::vector<MyType> x; //array of structs
```

Listing 1: VectorPU with Different Data Type

Smart Iterator

```
vectorpu::vector<My_Type> x(N);
std::generate(WI(x), WEI(x), RandomNumber);
thrust::sort(GRWI(x), GRWEI(x));
std::copy(RI(x), REI(x), std::ostream_iterator<My_Type>(std::cout, ""))
    ;
```

Listing 2: Heterogeneous programming by using VectorPU to glue STL and Thrust

```
vectorpu::vector<My_Type> cpu(N), gpu(N);
std::generate(WI(cpu), WEI(cpu), RandomNumber);
std::generate(WI(gpu), WEI(gpu), RandomNumber);
userspace::sort<<<32,256>>>(GRWI(gpu),GRWEI(gpu));
std::sort(GRWI(cpu), GRWI(cpu)+N);
cudaDeviceSynchronize();
```

Listing 3: Hybrid computation

VectorPU Lambda Functions

- Code snippet as coherence granularity

```
//alpha signature
for(std::size_t i=0; i<N; ++i)
{
  W(z)[i]=3;
}

//beta signature
#define lambda(z) \
        for(std::size_t i=0; i<N; ++i)   \
        { \
             z[i]=3; \
        }
#define VARGS ARGS(int, z, W)
VECTORPU_LAMBDA_GEN
```

Listing 4: VectorPU Lambda Functions

VectorPU Algorithms

- Extend STL algorithm to heterogeneous world
- Write the whole VectorPU algorithm library by β signatures

```
vectorpu::copy<int>(RI(x), REI(x), GWI(y) );
```

Listing 5: VectorPU Algorithms

Overloaded and Templated Functions

- Define separate flow signatures, and call with the right one.
- If they share flow signatures, define only one and reuse it.
  - e.g. template expansion usually does not alter flow signature

Express Skeletons in Skeleton Programming

- background: skeleton programming terms
  - skeleton: memory manipulation pattern, e.g. `map`
  - user function: memory manipulator operator, e.g. `negate()`
- different user function changes flow signature!
- α signatures help out: flow signature switch.

```
struct my_set{
  template <class T>
  __host__ __device__
    void operator() (T &x) { x+=101; }
};

int main() {
  vectorpu::vector<int> x(N);
  vectorpu::for_each<int>(GRWI(x), GRWEI(x), my_set() );
  vectorpu::for_each<int>(GWI(x), GWEI(x),
                          [] __device__ (int &x){x=10;} );
  vectorpu::for_each<int>(RI(x), REI(x),
                          [](int const &x) {cout<<x<<",";} );
}
```

Listing 6: Skeleton Programming by VectorPU

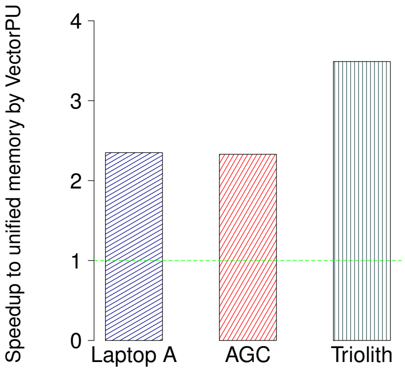
Performance Evaluation Platform

Table 2: Machine configuration

Machine	CPU	GPU	CUDA
Laptop A	Intel(R) Core(TM) i7-4710MQ @ 2.50GHz	1 K2100M (Kepler)	7.5
AGC (workstation)	Intel(R) Xeon(R) E5-1620 v3 @ 3.50GHz	1 K620 (Maxwell)	7.5
Triolith n1598 (supercomputer)	Intel(R) Xeon(R) E5-2660 0 @ 2.20GHz	1 K20Xm (Kepler)	7.5

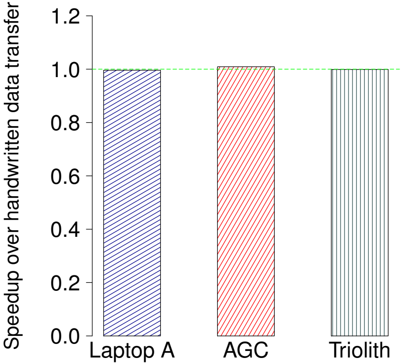
Comparison with Nvidia UM

- Conjugate gradient, popular numerical method
- Originally written with Nvidia's UM in CUDA SDK
- Rewritten with VectorPU
  - β signature
  - 8 VectorPU vectors
  - 6 VectorPU components
  - 12 component call places (8 in loops)
  - 3 VectorPU lambda functions
- Similar speedup:
  - A typical GPU parallel reduction: $1.40\times$ to $8.66\times$
  - Thrust sort() on 1M element: $5.58\times$ to $13.29\times$



Comparison With Expert Code (Manual Data Transfer)

- FFT
- Originally written with `cudaMemcpy()` in CUDA SDK
- Rewritten with VectorPU
  - α and β signature
  - Compound data type
  - 4 VectorPU vectors
  - 4 VectorPU components
  - 9 component call places (none in loops)



Programmability Improvement

- Responsibility removed:
  - allocating memory on both host and device
  - explicit moving of data with the calculation of the memory size
  - carefulness to avoid unnecessary movements
  - freeing those memory copies and freeing them only once
  - ...
- Simple example vectorAdd from the CUDA SDK
  - drops from 75 to 24 LOC after rewriting with VectorPU
- Parallel reduction by unified memory
  - drops from 21 to 17 LOC after rewriting with VectorPU
  - do allocation and initialization by one line, no deallocation required
- Iterator for productivity, raw pointers for tunability

Conclusion

- Observation: UM offers nice programmability, but efficiency is poor

- VectorPU makes unified memory practical,
  both performance-wise and programmability-wise,
  even no source available
- Wide expressiveness of VectorPU component and vector
- Much more efficient than UM,
  no noticeable slowdown compared to manual code.
- Using OpenMP-like pragmas: cleaner interface
  with better static support e.g. for spotting errors,
  but using macros: no extra tool support required

- Questions?
- Contact:
  - lu.li@liu.se
  - http://www.ida.liu.se/ lilu09/