I. Introduction

II. Template Metaprogramming Approach

A. Background

B. Overview

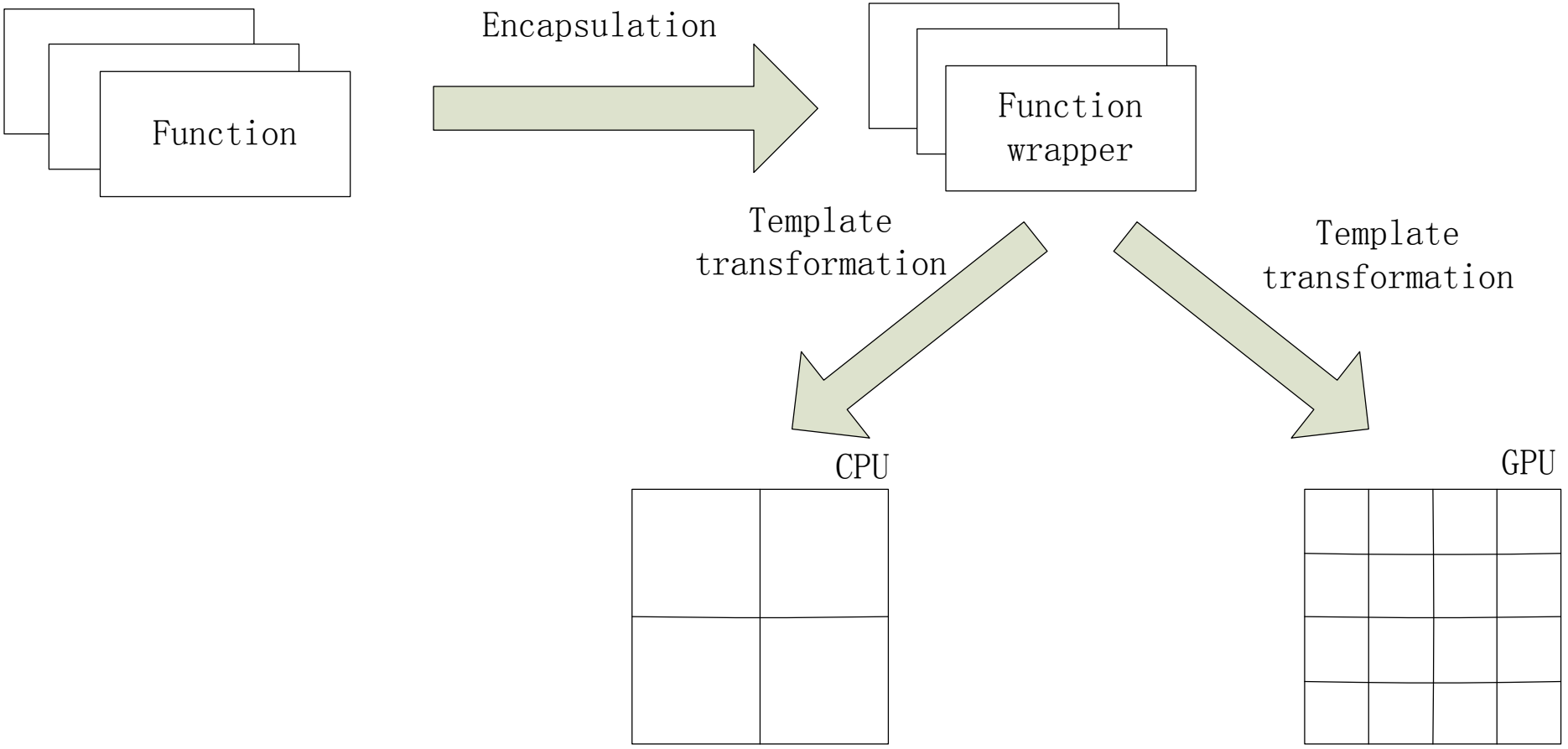

Fig. 1. template transformation

C. Components

1) TF class:

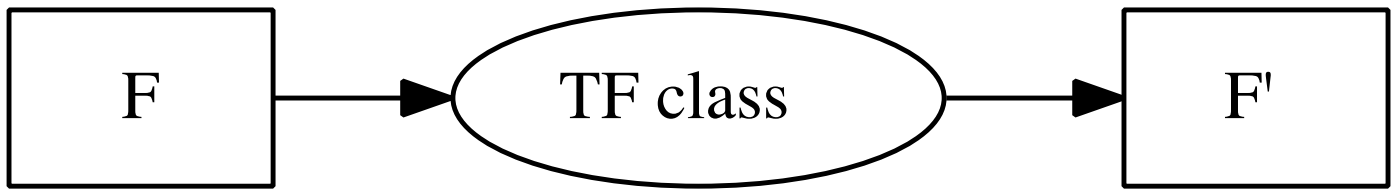

Fig. 2. transform class

2) Function wrapper:

```
/*A function wrapper for vector addition
 */
template<class Result, class Arg0,
         class Arg1>
struct vecAddWrapper {
 //omitted...

 //entry
 static void
 doit(const Arg0& arg0, const Arg1& arg1,
     Result& result)
 {
  vecArithImpl<T, DIM_N>::add(arg0, arg1,
                              result);
 }

 //entry with a barrier to synchronize
 static void
 doit_b(const Arg0& arg0, const Arg1& arg1,
     Result& result, mt::barrier& barrier)
 {
   doit(arg0, arg1, result);
   barrier.wait();
 }

 //...
};
```

Fig. 3. Function Wrapper

3) Predicate:

```
/*determine whether the problem size is
 *less than last level cache.
 */
template <class T, int SIZE_A
     , int SIZE_B, int SIZE_C>
struct p_lt_cache_ll {
 enum {CACHE_LL_SIZE = 4096*1024};
 const static bool value =
     ((SIZE_A * SIZE_B
   + SIZE_A * SIZE_C + SIZE_B * SIZE_C)
   * sizeof(T) ) <= CACHE_LL_SIZE;
};
```

Fig. 4. Predicate

4) Sentinel:

D. Supporting data structures

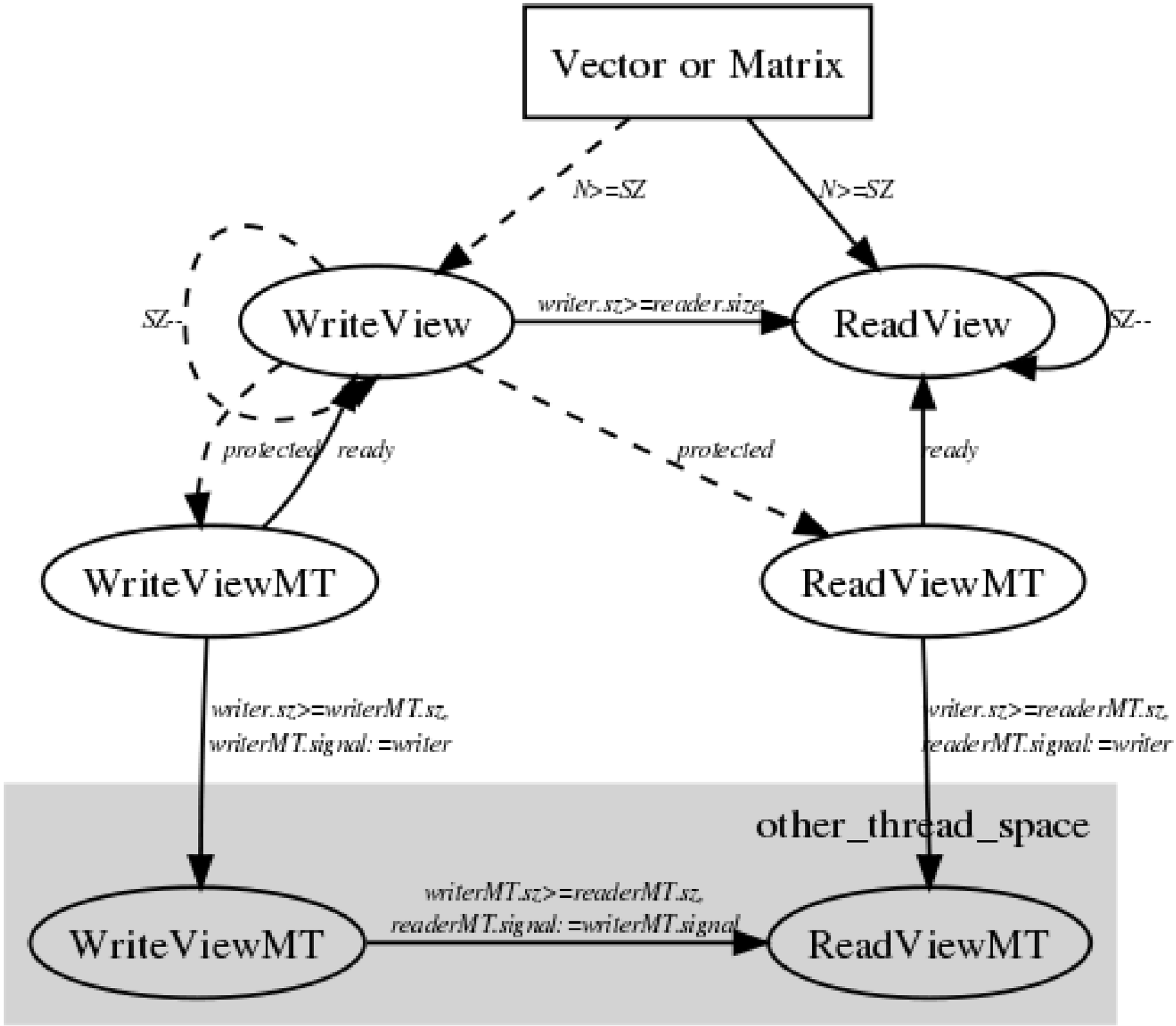

Fig. 5. View classes in libvina

III. RUNTIME SUPPORTS

IV. Source Transformations by Template

A. SPMD

```
/*A TF class for mappar. no dependence exists
 *in subtasks of Instance.
 */
template <class Instance,          /*problem*/
          int _K,       /*number of children*/
          bool _IsMT,      /*multi-threaded?*/
          bool __SENTINEL__ = Instance::_pred
>
struct TF_mappar {
 /*define recursively, evalue __SENTINEL__
  *using subtask's predicate
  */
 typedef TF_mappar<typename Instance::SubTask,
       _K, _IsMT,
       Instance::SubTask::_pred> _Tail;

 //determine whether Arg0 is isomorphic
 typedef typename mpl::or_<mpl::bool_
   <std::tr1::is_arithmetic
     <typename Instance::Arg0>::value>
   ,boost::mpl::bool_<
   std::tr1::is_same<typename Instance::Arg0,
            typename Instance::SubTask::Arg0>
      ::value>
   >::type
 arg0_isomorph;

 //omitted ...

 static void
 doit(const typename Instance::Arg0& arg0,
      const typename Instance::Arg1& arg1,
      typename Instance::Result& result)
 {
   for (int k=0; k < _K; ++k) {
    auto subArg0 = aux::subview<decltype(arg0),
        arg0_dim::value, arg0_isomorph::value>
            ::sub_reader(arg0, k);
    auto subArg1 = ...
    auto subResult = ...

    //execute recursively
    _Tail::doit(*subArg0,*subArg1,*subResult);
   }
 }
};
/*A partial template specialization of
 *TF_mappar.
 */
template <class Instance, int  _K>
struct TF_mappar <Instance, _K
             ,true          /*multithreaded*/
             ,true/*predicate is statisfied*/>
{
 //omitted...
 static void
 doit(const typename Instance::Arg0& arg0,
   const typename Instance::Arg1& arg1,
   typename Instance::Result& result)
 {
   auto compF = Instance::computationMT();

   /*bind function object to a thread
    *and run it.*/
   mt::thread_t leaf(compF, arg0, arg1,
     aux::ref(result, result_arithm()));
 }
};
```

Fig. 6. TF class of *TF_mappar*

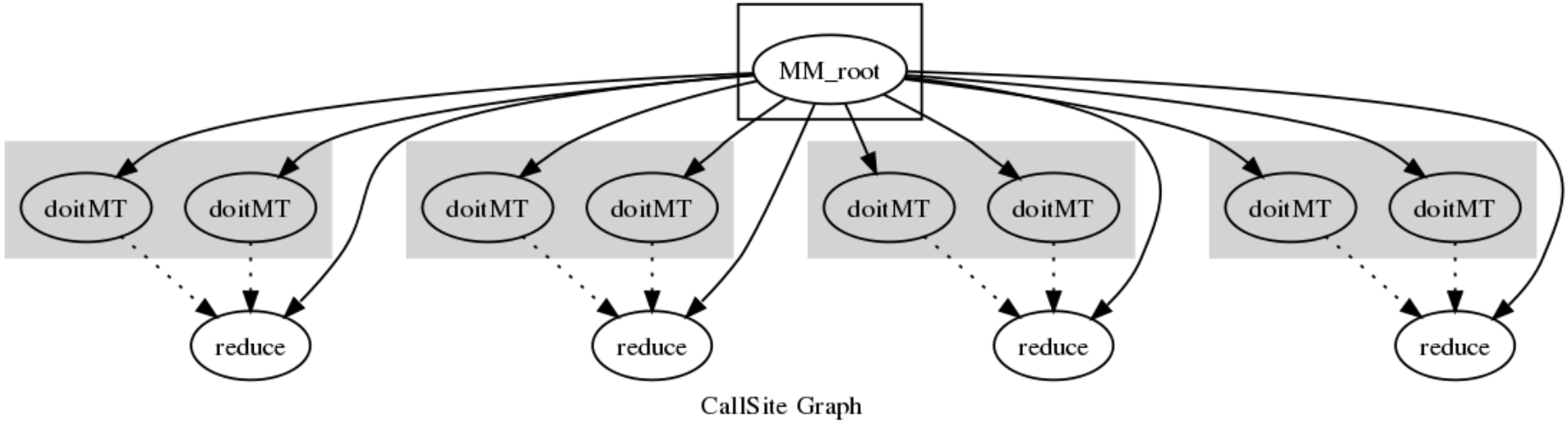

Fig. 7. MM internal call graph

B. Streaming and pipelining

```
/*A TF class for pipeline. User need to define
 *spacialization to handle the output of
 *pipeline.
 */
template <typename... Stages>
struct TF_pipeline;

template <class P, typename... Tail>
struct TF_pipeline<P, Tail...> {
  typedef typename P::input_type in_t;
  typedef typename P::output_type out_t;

 static out_t doit(in_t in)
  {
    //static checker, omitted...
    TF_pipeline<Tail...>::doit( P::doit(in) );
  }
};
```

Fig. 8. TF class of pipeline

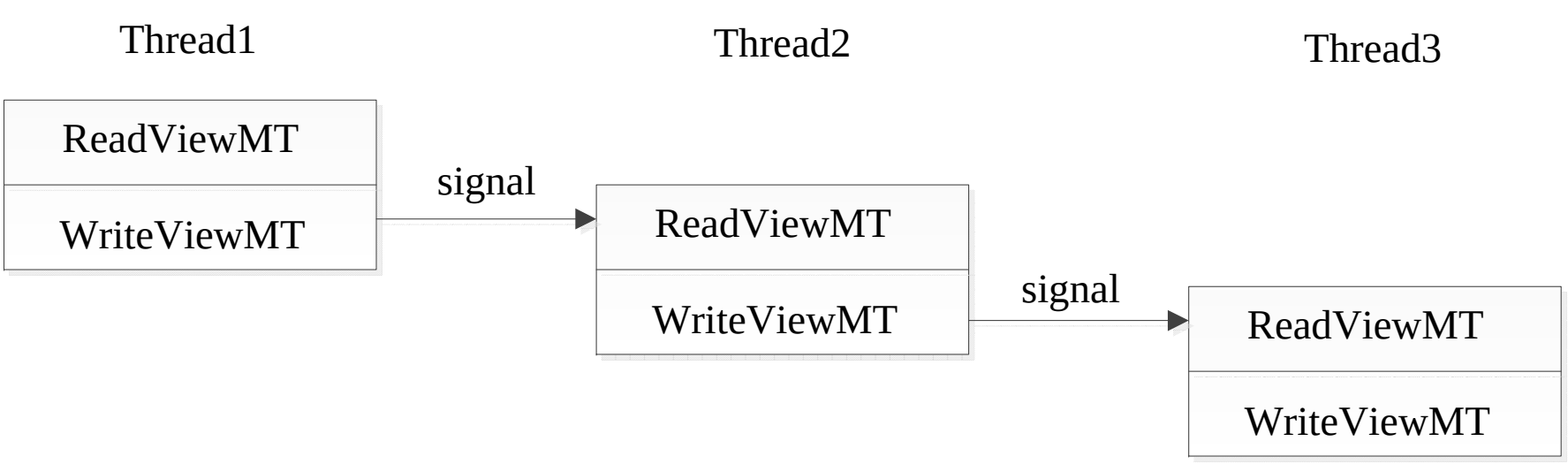

Fig. 9. ViewMT in pipeling

V. Adaption of TF classes

```
/*A adapter class for algorithm saxpy.
 */
template <class RESULT, class T,   class RHS,
    template <typename, typename> class Func,
    template <typename, int>      class Pred,
              int K = 2 , bool IsMT = false>
struct ADAPTER_saxpy{
  //trait classes for type resolution
  typedef view_trait<RHS>    trait1;
  typedef view_trait<RESULT> trait_res;

  //types interfaces used by TF class
  typedef T                                  Arg0
  typedef typename trait1::reader_type       Arg1
  typedef typename trait_res::writer_type
                                           Result
  //define TF class
  typedef TF_mappar<ADAPTER_saxpy/*instance*/
                    ,K, IsMT> Map;

  //define subTask
  typedef ReadView<T, trait1::READER_SIZE / K>
    SubRView1;
  typedef WriteView<T
              ,trait_res::WRITER_SIZE / K>
    SubWView;
  typedef ADAPTER_saxpy<SubWView, T, SubRView1,
                  Func, Pred, K, IsMT>
    SubTask;

  //pre-calculate predicate for TF class
  const static bool _pred =
   Pred<T, trait1::READER_SIZE>::value;

  static void
  saxpy(const T& alpha, const Arg1& lhs,
       Result& rhs)
  {
    Map::doit(alpha, lhs, rhs);
  }

  static std::tr1::function
  <void (const T&, const Arg1&, Result&)>
  computation()
  {
    return &(Func<Result, Arg1>::doit);
  }
};
```

Fig. 10. an adapter class for saxpy

```
//regular usage
vina::saxpy<VEC_TEST_TYPE, VEC_TEST_SIZE_N>
(7, x, result);

//use TF class to perform saxpy
typedef ADAPTER_saxpy<Writer, VEC_TEST_TYPE
            ,TestVector          /*Vector type*/
            ,wrpSaxpy      /*function wrapper*/
            ,p_lt_cache_ll          /*predicate*/
            ,2/*K*/     ,true/*Multi-threaded*/
> TF_MT;
TF_MT::saxpy(7.0f, x, result);
```

Fig. 11. Call function with and without transformation

```
/*template full specialization for langpipe
 */
template<>
struct TF_pipeline<>
{
 static const bool _IsTail = true;
 typedef view_trait<STRING>::reader_type
   READER;

 template <class U>
 static void output(U* in)
 {
   int i;
   /*U may be a ReadViewMT, in which case
    *the assignment is blocking operation.
    */
   READER reader = *(in);
   char output[READER::VIEW_SIZE+1];

   for(i=0; i<READER::VIEW_SIZE; ++i)
     out[i] = reader[i];
   out[i] = '\0';
   std::cout << out << std::endl;
 }

 template <class T>
 static void doit(T * in)
 {
   std::tr1::function<void (T*)>
     func(&(output<T>));

   mt::thread_t thr(func, in);
 }
};

//customize pipeline TF class
typedef TF_pipeline<
   translate<Eng2Frn
            ,true/*Multi-threaded?*/
           >,
   translate<Frn2Spn, true>,
   translate<Spn2Itn, true>,
   translate<Itn2Chn, true>
> MYPIPE;

//function call
MYPIPE::doit(&input);
```

Fig. 12. Call function before and after transformation

VI. EXPERIMENTS AND EVALUTION

A. Methodology

TABLE I
EXPERIMENTAL PLATFORMS

name	**type**	**processors**	**memory**	**OS**
harpertown	SMP server	x86 quad-core 2-way 2.0Ghz	4G	Linux Fedora kernel 2.6.30
macbookpro	laptop	x86 dual-core 2.63Ghz GPU 9400m 1.1Ghz	2G 256M	Mac OSX Snowleopard

B. Evalution

1) Speedup of SPMD transformation on CPU:

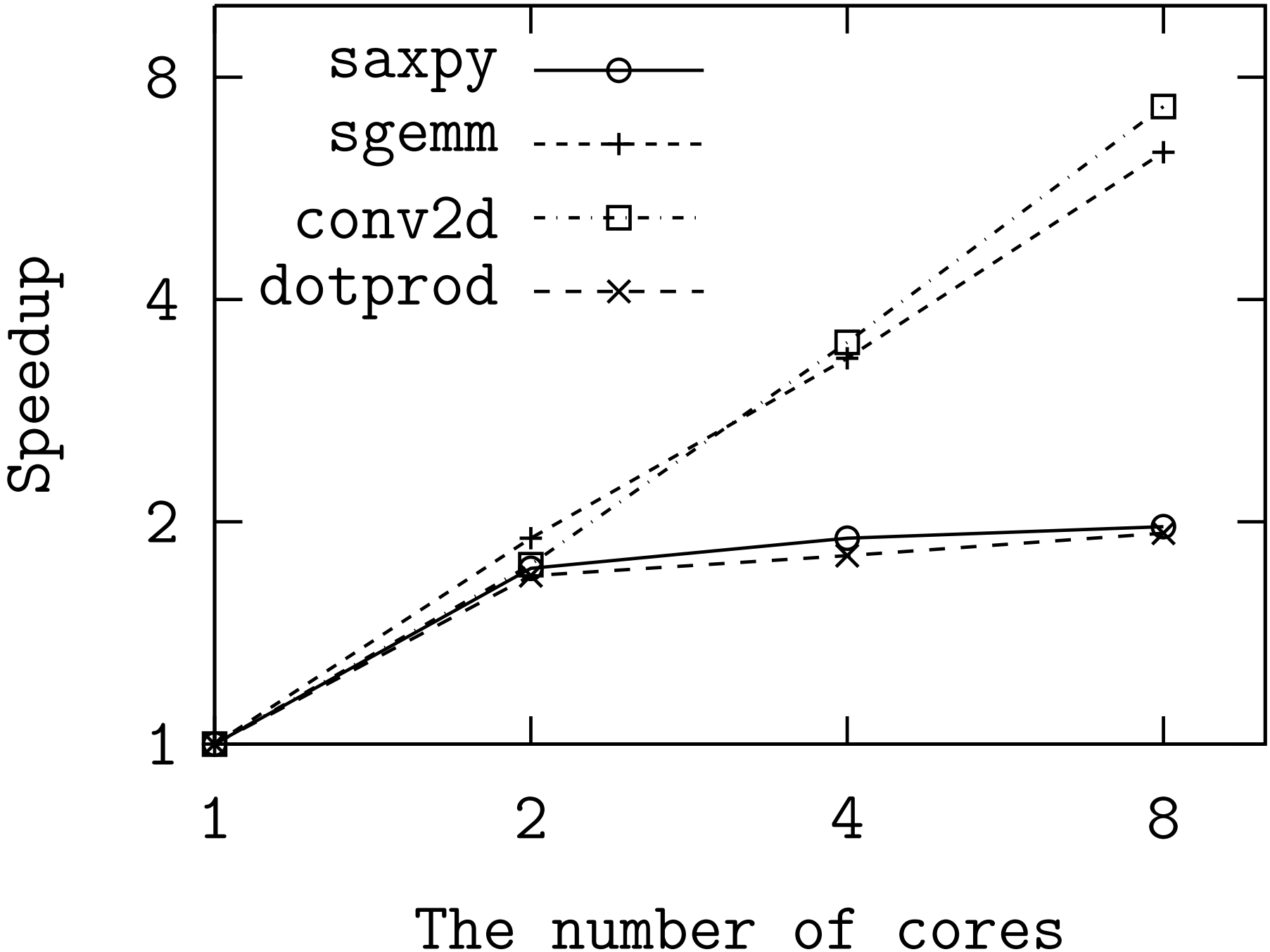

Fig. 13. Speedup on Hapertown

2) Speedup of SPMD transformation on GPU:

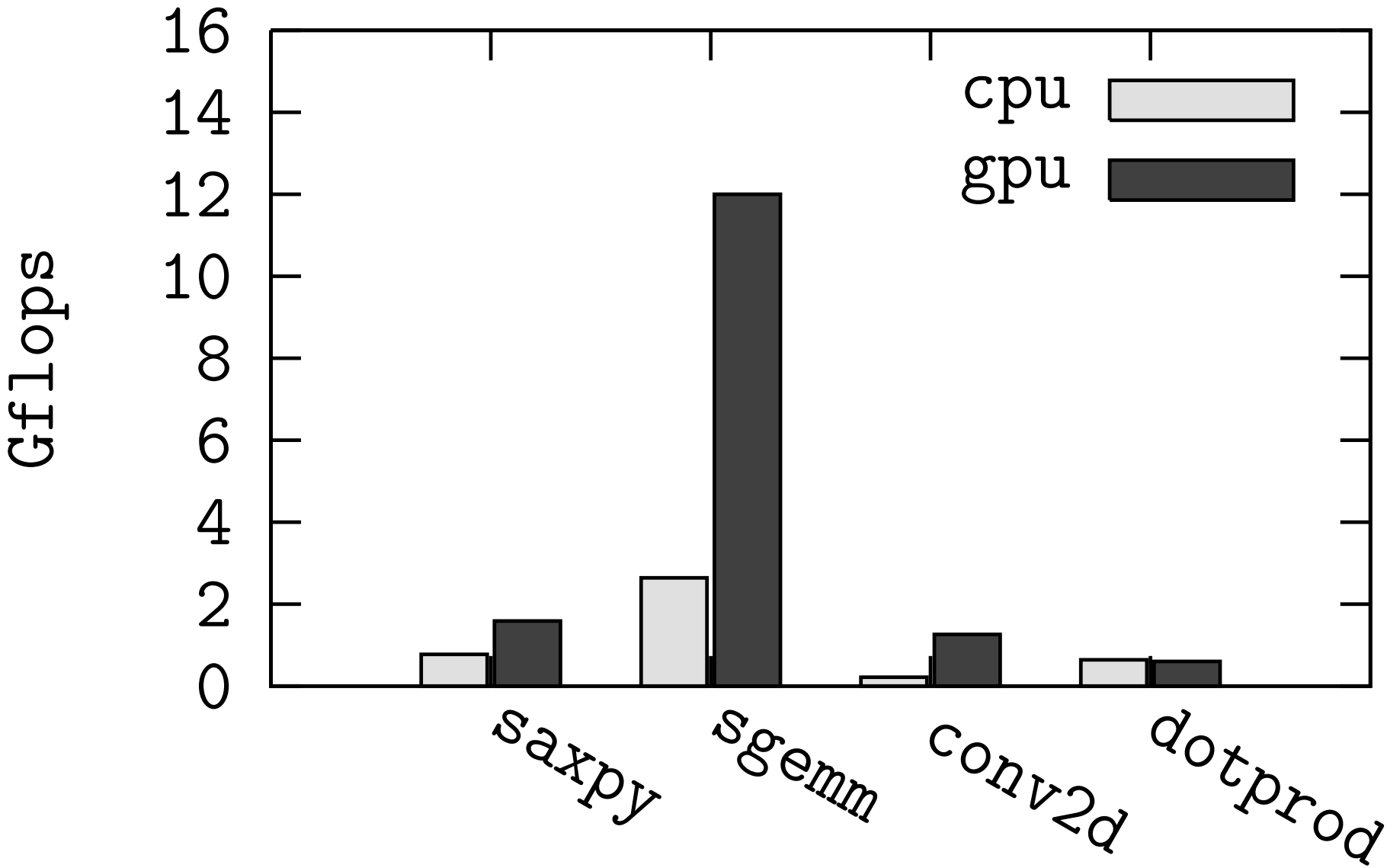

Fig. 14. Speedup Comparing GPU with CPU

1

2

3) Comparison between different multicores:

TABLE II
COMPARISON OF SGEMM ON CPU AND GPU

	baseline	CPU	GPU
cores	1 x86(penryn)	8 x86(harpertown)	2 SMs
Gflops	2.64	95.6	12.0
effectiveness	12.6%	74.9%	68.2%
lines of function	63	unknown	21

3

4) Support streaming computation:

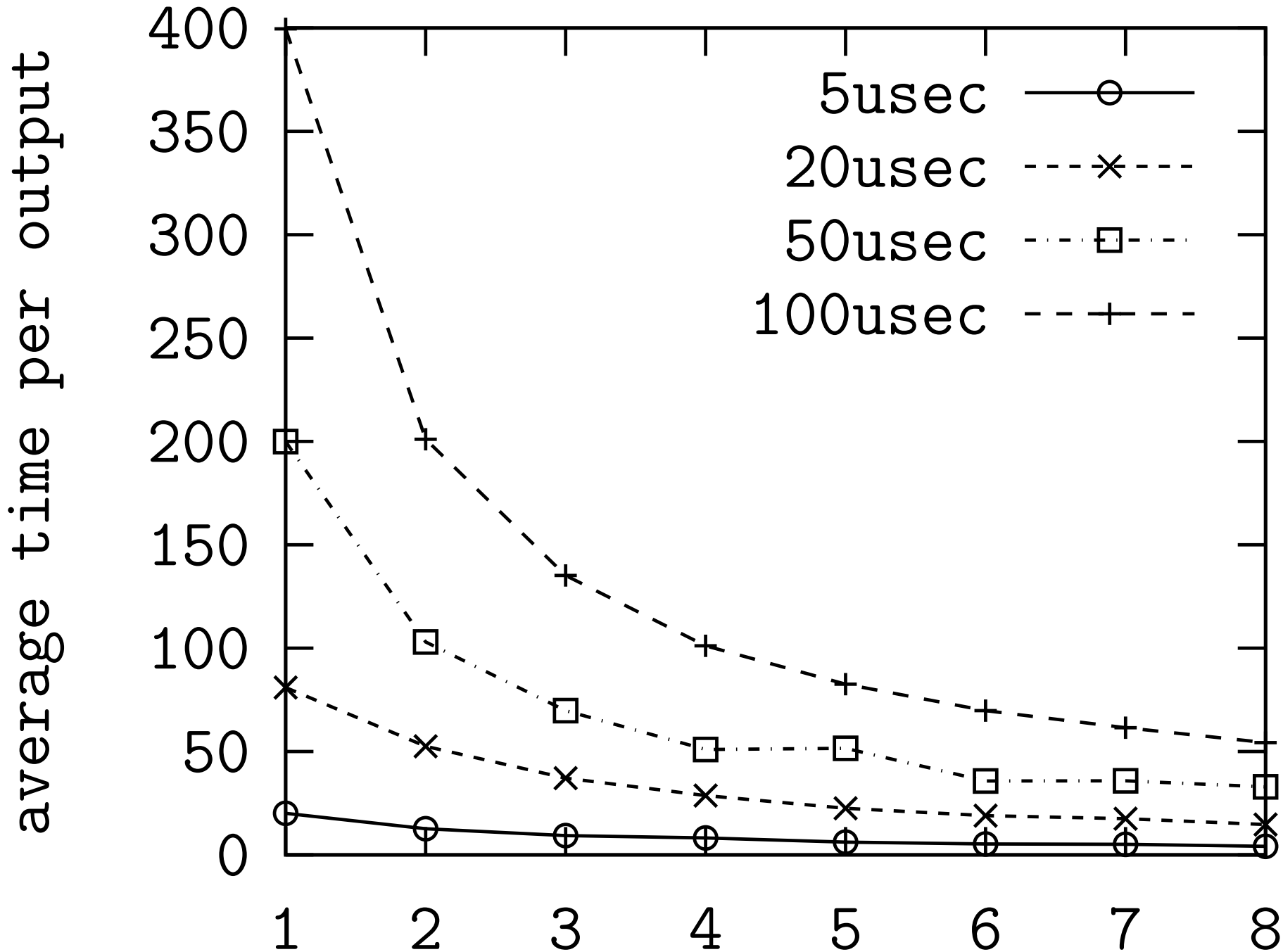

Fig. 15. Pipeline Processing for Psuedo Language Translation

4

t

μs

μs

μs

t

μs

μs

μs

μs

μs

μs

VII. Related work

5

VIII. Discussion and Future work