CUDA Expression Templates

Paul Wiemann
Computer Graphics Lab,
TU Braunschweig, Germany
p.wiemann@tu-bs.de

Stephan Wenger
Computer Graphics Lab,
TU Braunschweig, Germany
wenger@cg.tu-bs.de

Marcus Magnor
Computer Graphics Lab,
TU Braunschweig, Germany
magnor@cg.tu-bs.de

ABSTRACT

Many algorithms require vector algebra operations such as the dot product, vector norms or component-wise manipulations. Especially for large-scale vectors, the efficiency of algorithms depends on an efficient implementation of those calculations. The calculation of vector operations benefits from the continually increasing chip level parallelism on graphics hardware. Very efficient basic linear algebra libraries like CUBLAS make use of the parallelism provided by CUDA-enabled GPUs. However, existing libraries are often not intuitively to use and programmers may shy away from working with cumbersome and error-prone interfaces. In this paper we introduce an approach to simplify the usage of parallel graphics hardware for vector calculus. Our approach is based on expression templates that make it possible to obtain the performance of a hand-coded implementation while providing an intuitive and math-like syntax. We use this technique to automatically generate CUDA kernels for various vector calculations. In several performance tests our implementation shows a superior performance compared to CPU-based libraries and comparable results to a GPU-based library.

Keywords: GPU computing, parallel computing, CUDA, linear algebra

1. INTRODUCTION

In the last years general purpose computation on graphics processing units (GPGPU) has become more and more popular [Deg10, TNA+10, VKS10]. The modern GPU is not only a graphic engine but also a flexible programmable processor that can execute thousands of threads in parallel[TNA+10]. In future parallel computing will most probably get even more important. Microprocessor development will focus on adding cores rather then increasing single thread performance [OHL+08]. Since todays GPUs outclass consumer CPUs in terms of FLOPS, which is a common measure for computing capabilities, it is obvious that one should use this to speed up numerical calculations. Highly parallel linear algebra libraries like CUBLAS make use of computing power on graphics hardware but have a lack in usability. In this paper we take on this problem by introducing a technique allowing us to use a concise and math-like syntax, while utilizing the computing power of the GPU. We achieve our goal by combining CUDA and the expression templates technique.

CUDA [NVI10b] is a general purpose parallel computing architecture developed by NVIDIA. It allows one to run code, written in CUDA C, which is derived from C and has some extensions but also restrictions, on CUDA-enabled GPUs. In general, this is done by writing a so-called kernel, a function that is executed N times in N different threads. The threads are organized in warps and blocks. Each block is individually scheduled on the GPU processor cores. This means, blocks can run on any processor core in any order. Thus, a CUDA kernel scales automatically with the number of cores. A warp is a group of 32 parallel threads within a block. Each warp is scheduled individually and executes one instruction per cycle. Hence, to reach maximum efficiency the threads' execution path within a warp should not diverge, because otherwise each path will be executed separately in serial, multiplying the execution time by the number of execution paths. [NVI10b]

In our approach we use the expression template technique, which was concurrently invented by Todd Veldhuizen and David Vandevoorde in 1995 [VJ03, Vel95, IR09], to generate CUDA kernels. In general expression templates make passing expressions as function arguments in C++ possible. The expression gets inlined into the function body, preventing the overhead of callbacks. Therefore a templated class is defined, which represents an arbitrary expression. At compile time the expression gets parsed and stored as the template parameter [Vel95].

We use this technique to generate CUDA kernels, for arbitrary mathematical expressions, which then can concurrently be evaluated on the GPU.

2. RELATED WORK

Since an efficient implementation of vector algebra operations is crucial to many algorithms, there are many libraries facing this problem. On the one hand, template expression math libraries like uBLAS or blitz++[1] provide a math-like syntax but do not use the new computing capabilities on GPUs [Vel00, WK+10]. But since scientific computing through graphics hardware can be considerably faster then C-code for the CPU [CAN08], we attempt to make use of those capabilities in our approach. On the other hand libraries like CUBLAS [NVI10a] or thrust[2] are based on CUDA. While CUBLAS implements basic linear algebra functionality, thrust is a generic C++ template library for CUDA with a high-level STL-like interface. However, neither of the two provides a convenient mathematical syntax as our implementation does. Additionally, with CUBLAS, only a fixed set of functions is available, which leads to unnecessary calculations as well as to the use of temporary objects in the limited GPU memory [NVI10a]. The thrust library supports user-defined operations but requires manual specification of functor classes, which means plenty lines of code for the SAXPY operation $Y = c * X + Y$, with c as a constant. A simpler solution again requires the allocation of temporary objects. Unlike existing CUDA libraries, our implementation provides a math-like syntax and avoids temporary objects for most operators.

3. IMPLEMENTATION

Like already said we want to make the utilization of CUDA based vector-calculus easier via the expression template technique. This technique allows to pass expressions as function arguments. Those expressions get inlined, thus the code is nearly as fast as handwritten C. It can also be used to overload class operators and have the compiler generate the code to compute the result in a single pass without temporary objects. To achieve this, no subset of an arbitrary expression may be evaluated, until the entire expression is known. At compile time, the complier determines the expression type and stores it as a template parameter. Hence, all operations and operands are determined before the evaluation is evaluated and according to this the expression can be computed in one pass [VJ03, Vel95].

The CUDA compiler (nvcc) does only support a subset of C++ that includes function templates but no general template programming [NVI10b]. Thus, expression templates can not be used directly in CUDA code Instead, we use the expression templates to generate a CUDA kernel for each expression type at runtime. Those kernels are automatically executed once the execution reaches the code line, where the expression occurs.

We achieve our goal by introducing several new classes. Initially we introduce the classical classes of the expression template technique: A base class `Expression` and a vector class `cudaVec`. `Expression` represents any kind of an expression (without assignment), like $v + w$, component_wise_sin(v) or simply v. Since v is also an expression `cudaVec` is derived from `Expression`. `cudaVec` also inherits `thrust::device_vector` from the thrust library to allow for interoperability with thrust's generic interface, e.g. for reductions. Additionally, we develop the class `AssignmentExpression` with represents an assignment of an expression to a `cudaVec` and overload the assignment operator in the `cudaVec` class to instantiate an `AssignmentExpression`. For example $v = w+u$ is represented by an `AssignmentExpression`. A subset of the class structure is shown in Figure 1.

For each element-wise operation, like multiplication of a vector with a scalar or the addition of two vectors, an operator expression class is derived from `Expression`. For example, the multiplication of a vector with a scalar would be implemented in a `VectorMultipliesScalarExpression` class.

In order to allow for a concise and math-like syntax, we overload the arithmetic operators (like +, -, *, /) for expressions such that they invoke the constructor of the appropriate operator expression. An Example of an operator expression class and the associated creator function is shown in Listing 1. In the firest part the class `SumExpression` is defined which represents a plus operation with two arbitrary expressions as operands. The class has two template parameters, each of which stores an operator expression type. The template parameters' types depend on the expression types the constructor is called with. In the second part one can see the associated creator function. It is a templated function, in this case the plus operator, thus any expression can be an argument. The creator function invokes the constructor of its associated class.

In this way, we can directly write vector algebra in application code and the necessary tree of expression classes is automatically instantiated. A simple example is given in Listing 2. Since the plus operator is redefined as an creator function, it returns an instance of the `SumExpression`. Both template parameters are typed as `cudaVec`, because `a`'s and `b`'s type is `cudaVec`. Additionally the overloaded assignment operator in the cudaVec class instantiates an `AssignmentExpression` with `SumExpression<cudaVec,cudaVec>` as template Parameter. The resulting structure of the classes is shown in Figure 2.

[1] http://www.oonumerics.org/blitz/
[2] http://code.google.com/p/thrust

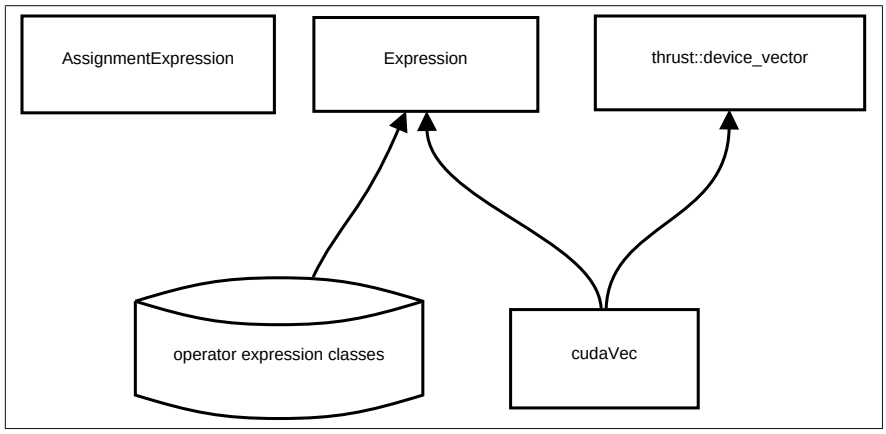

Figure 1: The class structure.

```
template <typename E1, typename E2>
class SumExpression : public Expression<SumExpression<E1, E2> > {
  public:
    const E1 &_l;
    const E2 &_r;
    SumExpression( const Expression<E1> &l, const Expression<E2> &r)
      : _l(l), _r(r) {}
//...
}

template<class E1, class E2>
SumExpression<E1, E2> inline operator +(const Expression<E1> &l,
    const Expression<E2> &r) {
  return SumExpression<E1, E2> (l, r);
}
```

Listing 1: Example for an operator expression class and the associated creator function.

```
cudaVec a(100), b(100), c(100);
//initialize a, b

c = a + b;
```

Listing 2: Sample code which instantiates the classes shown in Figure 2.

Because we want to evaluate the expressions on the GPU, each templated `AssignmentExpression` has to generate a CUDA kernel which performs the calculation. As Listing 3 shows, `AssignmentExpression` contains a static member variable to which the appropriate CUDA kernel is assigned at program startup. This is achieved by the static initializer `AssignmentExpression::init()` which is called for each templated `AssignmentExpression` and which takes care of generating the kernel code, compiling it with nvcc and loading it.

Each kernel is built according to a pattern (Listing 4). Only the parameter list and the evaluation line depend on the type of the `AssignmentExpression`. To create these two strings, we traverse the object hierarchy for the corresponding expression from the root. Each operator expression class writes its CUDA operator or CUDA function into the evaluation line and calls its parameters to do the same. Hereby the tree is gradually traversed. If a parameter is a terminal symbol (a `cudaVec` or a constant), the parameter list is extended by a new parameter and the name of the parameter is put into the evaluation line as an operand.

The example in Listing 2 generates the kernel shown in Listing 5. In this case the tree has a root typed as `SumExpression` with two children of the type `cudaVec`. As the tree is hierarchically traversed, the `SumExpression` writes its cuda operator (a +) into the evaluation line. Since both children are terminal symbols, each extends the parameter list and puts the parameter's name into the evaluation line.

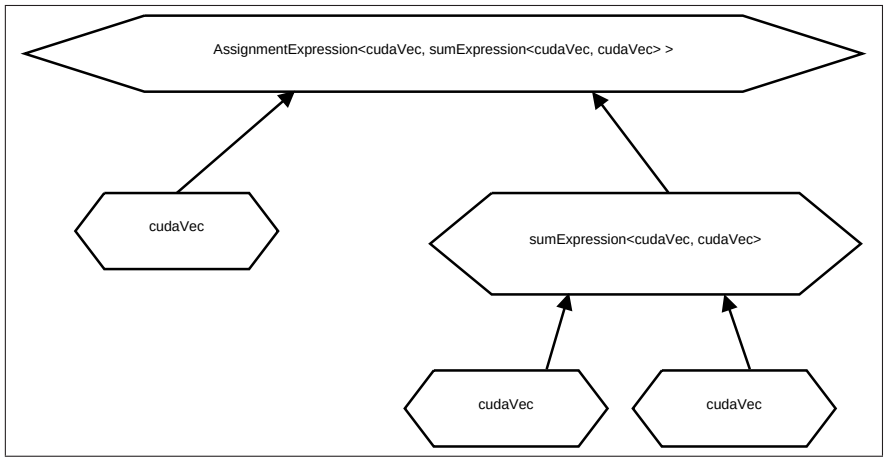

Figure 2: Hierarchy of objects that is created when Listing 1 is compiled.

```
template <class E>
class AssignmentExpression {
    //...
    static int kernelID;
    static int init();
  //...
}
template<class E>
int AssignmentExpression<E>::kernelID =
  AssignmentExpression<E>::init();
template<class E>
int AssignmentExpression<E>::init() {
  // generate, compile and load kernel
}
```

Listing 3: Implementation of the `AssignmentExpression` class.

```
extern "C" __global__ void kernel(float* a,
  /*parameterlist*/, unsigned int size){
  idx = blockDim.x * blockIdx.x + threadIdx.x;
  if (idx < size) {
    a[idx] = /*evaluation line*/;
  }
}
```

Listing 4: The Kernel prototype.

```
extern "C" __global__ void kernel(float* a, float* b,
  float* c, unsigned int size){
  idx = blockDim.x * blockIdx.x + threadIdx.x;
  if (idx < size) {
    a[idx] = b[idx] + c[idx];
  }
}
```

Listing 5: Kernel generated by compiling Listing 2.

The kernel code is compiled into an assembler-like ptx file by the CUDA compiler and then loaded with the CUDA Driver API.

Now, if an `AssignmentExpression` has to be evaluated, the expression template generated tree is traversed and each terminal symbol passes its value (in the case of `cudaVec` a pointer) to the kernel via the CUDA driver API. Thereafter the kernel is executed.

4. EXPERIMENTAL RESULTS

We now examine the experimental performance of our implementation. Zotos and Stephanides have shown that the performance of major numerical CPU-based libraries varies only by a factor of 4 [ZS]. We therefore exemplarily compare our implementation to the expression template library uBLAS, as well as to an implementation based on the thrust GPU programming framework. Our tests were performed on an Intel Core i5 750 CPU, which has four cores running at 2.67 GHz on a 64 bit Linux system with four GB RAM and an NVIDIA GeForce GTS 250 GPU with 1 GB on-board memory.

The GPU time is only reported as execution time and neither includes the time of transferring input data across the PCI express bus to the device nor the time necessary to compile the CUDA kernels. Normally, the data is transferred from the host to the device at the beginning and then all calculations concerning these data are executed. Only after the last calculation, the data is transferred back to the host. Since data transfer, which is limited by 4GB/s, can proceed while a contemporaneous kernel execution is in progress, the PCIe transfer can often be executed in the background, so that no major delay occurs [SHG09]. The kernel compiler is started only once at program startup, therefore the execution time gives us the clearest picture of the overall performance.

We perform different computations with various vector sizes to compare the efficiencies. Our first test is $a = b + c$ starting with a vector size of 1000000 up to a maximum of 50000000. Smaller sizes could not be compared since the processor time of an execution drops below 1ms. Figure 3 shows the result of this test. We gain a 55 to 58 times faster execution time for vector sizes between 1000000 and 33553920 compared to uBLAS and a little bit faster execution time compared to thrust. The execution time of our implementation increases heavily if the size goes from 33553920 to 33553921 due to the fact that we need to modify our kernel. We are forced to do so, because our first kernel computes one element per thread and now there are more elements then the maximum number of threads on the GPU used in our tests. Hence we only get a 35 times speed boost compared to uBLAS and drop slightly below the execution time of thrust. With a further increasing vector size our implementation closes the gap to thrust and increases performance compared to uBLAS as well. The described performance drop was observed in every test we took.

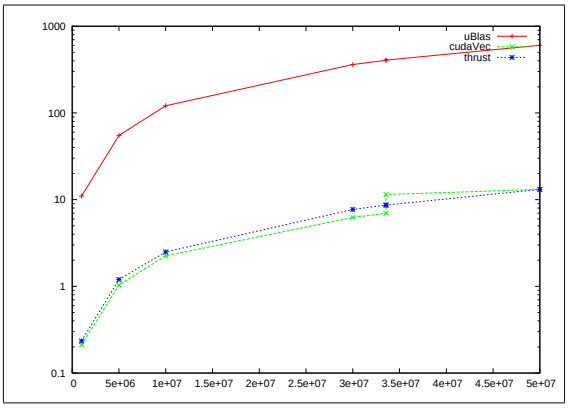

Figure 3: Test 1: a = b + c. The execution time of our CUDA implementation is 35 to 58 times faster compared to uBLAS and performs similar to thrust.

In the second and third test we test a bit more complex calculations. In comparison to uBLAS the results were similar to the first test and are shown in Figure 4 and 5. For thrust we tested both possible methods, one with a user-defined functor class and one with temporary objects. Since the memory on graphics hardware is limited the second method could only be tested for the vector-sizes up to 3355920. It is obvious that the temporary object method is up to five times slower then our implementation and also slower then the functor class approach.

In comparison to the functor class method our implementation performed faster in the second test for vector sizes below 33443920 but fell behind for greater vector sizes. Like in the first test, the gap closes with increasing size. The thrust implementation and ours performed on a similar level in the third test.

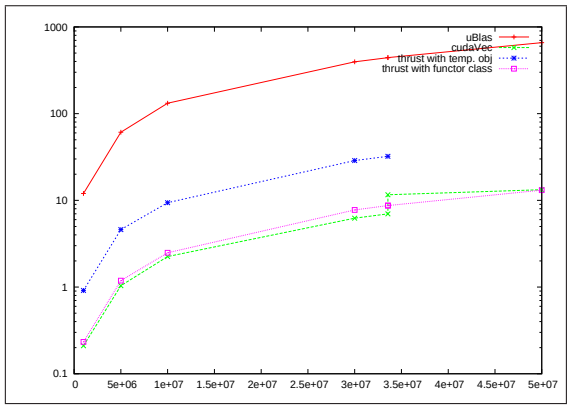

Figure 4: Test 2: a = 0.12*b + 7.54*c. The execution time of our implementation is 38 to 64 times faster compared to uBLAS and about 5 times faster then thrust with temporary objects. Thrust's implementation with a functor class and ours are on the same level.

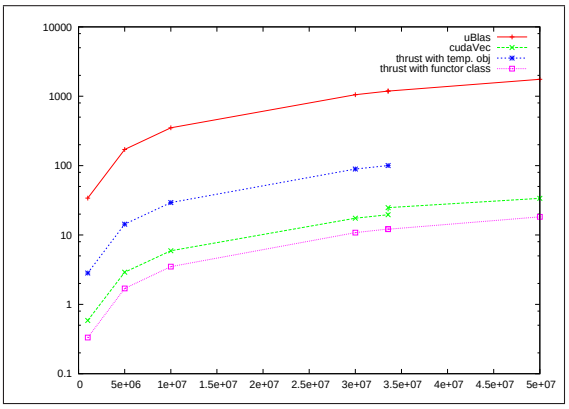

Figure 5: Test 3: $a = (b - (a + 3.75 * c) + c - 0.24 * b)/27.51 + a - 0.25 * b$. The execution time our implementation is 47 to 60 times faster compared to uBLAS and again about 5 times faster then thrust with temporary memory allocation. Our implementation and thrust's functor class approach perform similar.

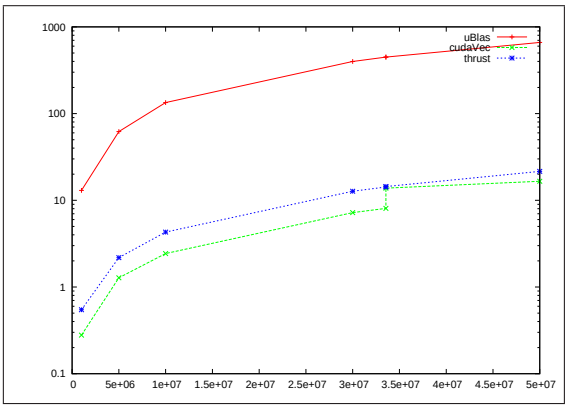

Figure 6: Test 4: $a = a/\|a\|_2$. The execution time of our CUDA implementation is 34 to 56 times faster compared to uBLAS. Thrust shows comparable results.

In a fourth test we normalized a vector. Therefor we had to compute the euclidean norm and multiply the inverse with the vector. Thrust did not provide the functionality of calculating the Euclidean norm, thus a user-defined functor class is needed. The computation of the Euclidean norm we used in our implementation is part of the CUBLAS library from NVIDIA and gets evaluated first, before the expression template generated kernel is executed. Even though we need two kernel executions there is a considerable performance gain compared to uBLAS as shown in Figure 6. The speed of our implementation and thrust's are comparable.

We also want to examine the different syntaxes of our implementation to thrust and uBLAS (Listing 6). Our implementation and uBLAS have the same concise and math-like syntax whereas thrust's syntax is more cumbersome and requires way more code.

5. CONCLUSION AND FUTURE WORK

We presented a technique, which leads to a library for vector algebra operations utilizing the capabilities of graphics hardware and still providing a math-like and concise syntax. Our implementation combines expression templates with CUDA, so that we benefit from the strengths of both.

Our experimental results show a superior performance compared to the non-GPU library uBLAS, while keeping the same brief syntax. In comparison to the thrust library our implementation performed similar to the user-defined functor class method in all tests. Compared to the thrust method with temporary objects our approach was considerably faster.

We want to point out that kernel generation occurs at run time and of course slows down the execution time of the program. But for programs with a long execution time this investment quickly pays off. There are possibilities conceivable that one could save time in this step. By caching earlier compiled kernels, recompilation can be avoided. Possible further extensions of our library include copy-free implementation of matrix algebra as well as further optimization of the kernel call parameters such as the number of threads per block.

REFERENCES

[CAN08] G. Cummins, R. Adams, and T. Newell. Scientific computation through a GPU. In *Southeastcon, 2008. IEEE*, pages 244–246. IEEE, 2008.

[Deg10] M. Degirmenci. Complex Geometric Primitive Extraction on Graphics Processing Unit. *Journal of WSCG*, pages 129–134, 2010.

[IR09] K. Iglberger and U. Rüde. The Math Library of the pe Physics Engine – Combining Smart Expression Templates and BLAS Efficiency. Technical report, Institut für Informatik, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2009.

[NVI10a] NVIDIA Corporation. CUBLAS Library. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUBLAS_Library.pdf`, 2010.

[NVI10b] NVIDIA Corporation. NVIDIA CUDA C Programming Guide. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf`, 2010.

[OHL+08] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

```
/*our implementation*/
//initialize vectors a,b,c
c = 0.12*b + 7.54*b;

/*uBLAS implementation*/
//initialize vectors a,b,c
c = 0.12*b + 7.54*b;

/*thrust implementation*/
//define functor
struct functor {
  const float f1;
  const float f2;

  functor(float _f1, float _f2)
    : f1(_f1), f2(_f2) {}

  __host__ __device__
  float operator()(const float& x, const float& y) const {
    return f1*x + f2*y;
  }
};

//initialize vectors a,b,c
thrust::transform(b.begin(), b.end(),
  c.begin(), a.begin(), functor(0.12f, 7.54f)
);
```

Listing 6: A comparison of the different syntaxes from thrust, uBLAS and our implementation.

[SHG09] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[TNA+10] A. Tasora, D. Negrut, M. Anitescu, H. Mazhar, and T.D. Heyn. Simulation of Massive Multibody Systems using GPU Parallel Computation. In *WSCG 2010 Full Papers Proceedings*, pages 57–64, 2010.

[Vel95] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[Vel00] T. Veldhuizen. *Advances in Software tools for scientific computing*, volume 10, chapter 2: Blitz++: The library that thinks it is a compiler, pages 57–87. Springer, 2000.

[VJ03] D. Vandevoorde and N.M. Josuttis. *C++ Templates: The Complete Guide*, chapter 18: Expression Templates. Addison-Wesley, 2003.

[VKS10] J.S.M. Vergeest, A. Kooijman, and Y. Song. Partial 3D Shape Matching Using Large Fat Tetrahedrons. *Journal of WSCG*, pages 41–48, 2010.

[WK+10] J. Walter, M. Koch, et al. uBLAS, Boost C++ software library. `http://www.boost.org/doc/libs/1_44_0/libs/numeric/ublas/doc/index.htm`, August 2010.

[ZS] K. Zotos and G. Stephanides. Analysis of Object-Oriented Numerical Libraries. Technical report, Deptartment of Applied Informatics, University of Macedonia.