

# Julia for High Performance Computing

**Carsten Bauer**

Paderborn Center for Parallel Computing (PC2)

20-23 September 2022

# Schedule

	Tuesday	Wednesday	Thursday	Friday
	Foundations	"Serial"	Parallel	Accelerators
9:00- 11:00 (2h)	Introduction	Interop. & Microbench.	Distributed-	Exercises
	Types & Dispatch	Performance		GPU
Short break				
11:15- 12:45 (1,5h)	Exercises	Optimisation	Computing	Programming
	Specialisation &	Exercises	Exercises	Q&A
Lunch break				
14:15- 15:15 (1h)	Generic Prog.	SIMD	Experience Rep.	
	Short break			
15:30 – 16:30 (1h)	Workflow & Pkgs	Profiling	Multithreading	

# Course Material

**[github.com/carstenbauer/JuliaHLRS22](https://github.com/carstenbauer/JuliaHLRS22)**

# Live Survey

# Why Julia for HPC?

# The Julia Programming Language

- **Great for domain science**

- “Python-like” syntax
- Dynamic & interactive
  - powerful REPL
  - Jupyter notebooks ([Julia](#) + [Python](#) + [R](#))
- Powerful linear algebra + ecosystem
- Package manager, environments

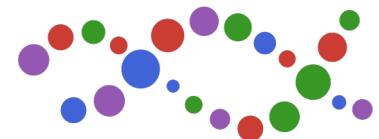
- **Great for HPC**

- Compiled via LLVM (just ahead of time)
- Parallel computing
- Hardware accelerator programming
  - High-level abstractions + low-level kernels
- Interactive HPC

```
julia> function sumup()
           x = 0
           for i in 1:100
               x += i
           end
           return x
       end
sumup (generic function with 1 method)
```

```
julia> @code_llvm debuginfo=:none sumup()
define i64 @julia_sumup_177() #0 {
top:
    ret i64 5050
}
```

# Bridging the Gap



Domain Science



julia



HPC

nature

**”Julia: come for the syntax, stay for the speed”**

[DOI: 10.1038/d41586-019-02310-3](https://doi.org/10.1038/d41586-019-02310-3)

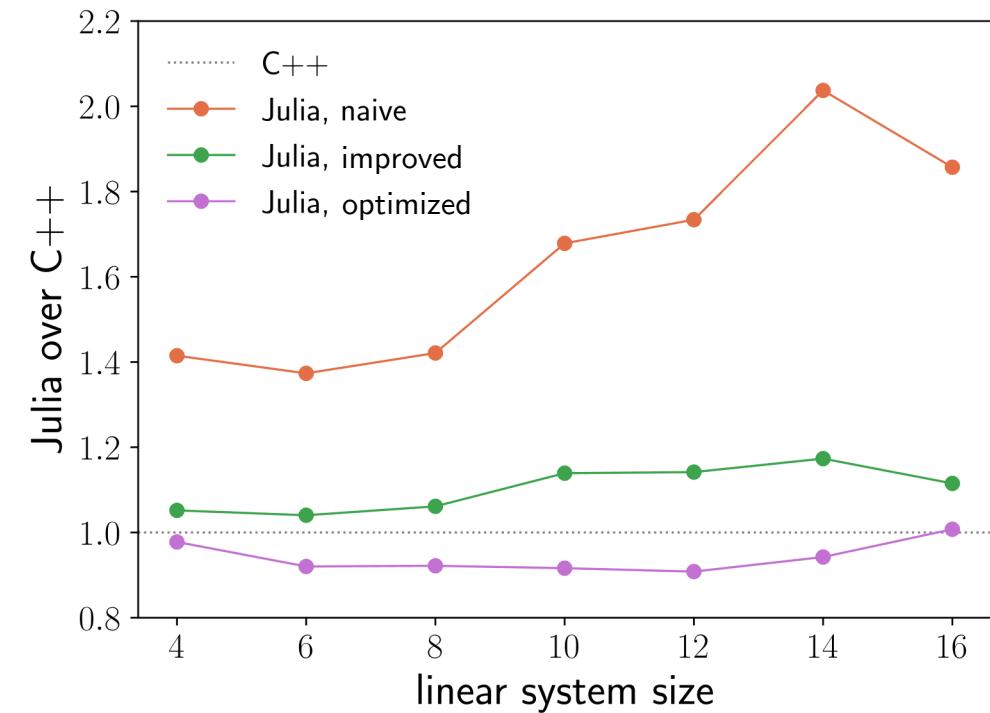
# Bridging the Gap



**Julia covers the spectrum between those two “limits”  
and can often provide the best of both worlds!**

# Personal Experience: Large-Scale Quantum Monte Carlo

- Condensed matter physics application
- Prototypical Julia implementation
  - ~3x faster than Python (not shown)
  - Within a factor of ~2x compared to C++
- Could gradually improve performance!
  - No disruptive language change necessary!
  - A lot of fun!

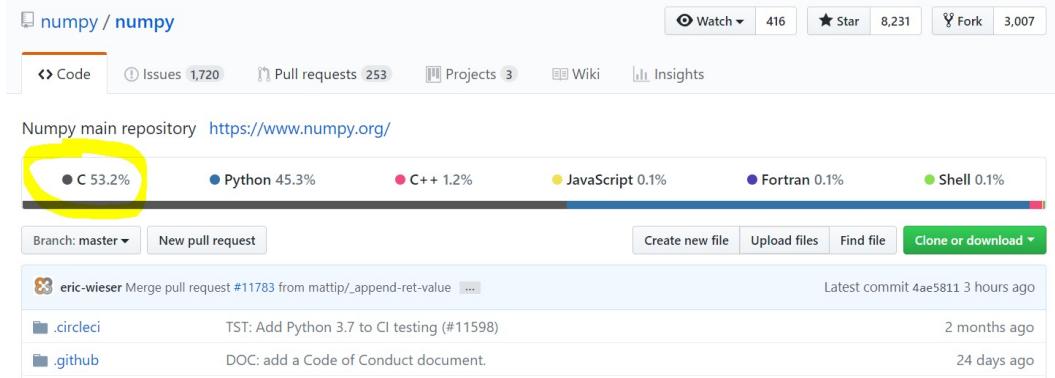


# Bridging the Gap



**Start with HPC on your laptop and transition to clusters!**

# Bridging the Gap



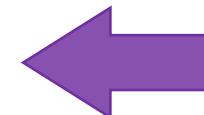
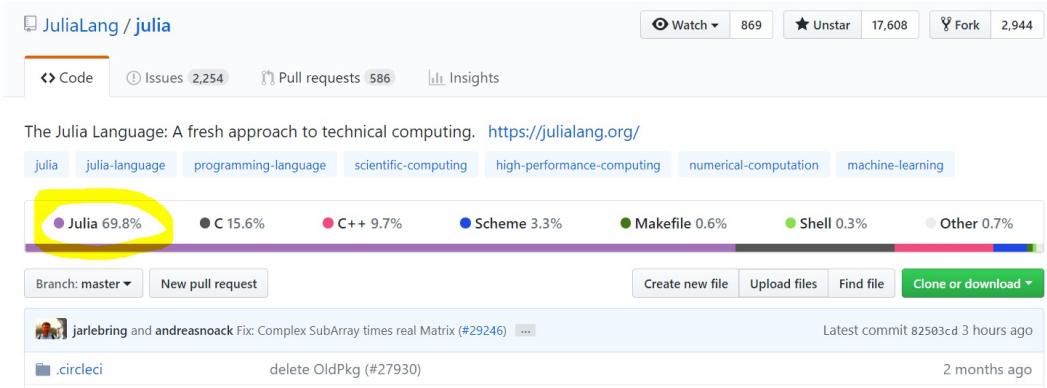
Developer



User



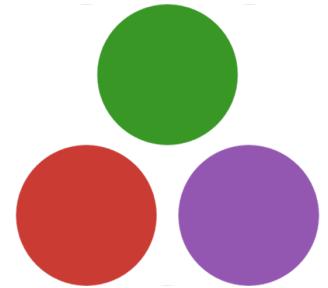
# Bridging the Gap



User



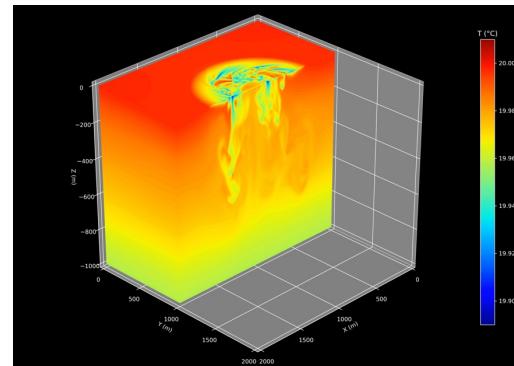
Developer



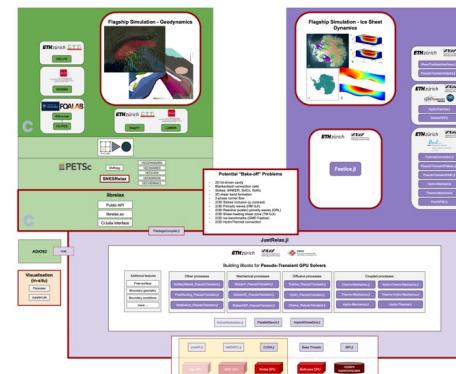
# Julia for HPC

# International Initiatives

- CliMA @ Caltech, MIT
  - Climate Modeling Alliance
- CESMIX @ MIT
  - Center for the exascale simulation of materials in extreme environment
- ExaSGD @ LLNL, ORNL, ...
  - National power grid optimization
- GPU4GEO @ ETH / CSCS
  - Computational earth science
- ...



[Oceananigans.jl](http://Oceananigans.jl)



# Julia HPC Position Paper

(in preparation)

## USA / Canada

- **MIT** Massachusetts Institute of Technology
- **NERSC** National Energy Research Scientific Computing
- **ORNL** Oak Ridge National Laboratory
- **PI** Perimeter Institute for Theoretical Physics

## Europe

- **PC2** Paderborn Center for Parallel Computing
- **HLRS** High-Performance Computing Center Stuttgart
- **UHH** University of Hamburg
- **CSCS** Swiss National Supercomputing Centre
- **ETH** Swiss Federal Institute of Technology Zürich
- **ARC@UCL** University College London

### Bridging HPC Communities through the Julia Programming Language

Journal Title  
XX(X):1-14  
OT To Be Assigned (s) 2022  
Reprints:  
[sagepub.co.uk/journalsPermissions.nav](http://sagepub.co.uk/journalsPermissions.nav)  
DOI: 10.1177/TobBeAssigned  
[www.sagepub.com/](http://www.sagepub.com/)  
**SAGE**

Valentin Churavy<sup>1</sup>, William F Godoy<sup>2</sup>, Carsten Bauer<sup>3</sup>, Hendrik Ranocha<sup>4</sup>, Michael Schlotke-Lakemper<sup>5</sup>, Ludovic Räss<sup>6,7</sup>, Johannes Blaschke<sup>8</sup>, Mosè Giordano<sup>9</sup>, Erik Schnetter<sup>10,11,12</sup>, Samuel Omlin<sup>13</sup>, Jeffrey S. Vetter<sup>2</sup>, Alan Edelman<sup>1</sup>

#### Abstract

The Julia programming language has evolved into a modern alternative to fill existing gaps in the requirements of scientific computing and data science applications. Julia's single-language paradigm, and its proven track record at achieving high-performance without sacrificing user productivity, makes it a viable single-language alternative to the existing composition of high-performance computing (HPC) languages (Fortran, C, C++) and higher-level languages (Python, R, Matlab) suitable for data analysis and simulation alike. Julia's rapid growth in language capabilities, package ecosystem, and community make it a promising new universal language for HPC similar to C++ or Python – an achievable goal if the community is given the necessary resources. This paper presents the views of a multidisciplinary group of researchers in academia, government, and industry advocating for the use of Julia and its ecosystem in HPC centers. We examine the current practice and role of Julia as a common programming model to address major challenges in scientific reproducibility, data-driven artificial intelligence/machine learning (AI/ML), co-design, and in-situ workflows, scalability and performance portability in heterogeneous computing, network, data management, and community education. As a result, we consider necessary the diversification of current investments to fulfill the needs of the upcoming decade as more supercomputing centers prepare for the Exascale era.

#### Keywords

High Performance Computing, Julia, Programming Language

#### 1 Programming Languages and HPC

The development of programming languages for high performance computing (HPC) has a rich and varied history. Early on, the needs of HPC and main-stream computing were still mostly aligned, leading to the development of Fortran as the first high-level language. Today, we see a bifurcation of programming languages designed for HPC, like Chapel ([Chamberlain et al. 2007](#)), and generalist programming languages, like Python and C++, that are being retooled for HPC use cases. Meanwhile, we see many new languages and programming models being deployed to help users manage the increased heterogeneity of contemporary systems: OpenCL, CUDA, HIP, OpenMP, OpenACC, SYCL, Kokkos, and many others.

Specialist languages lead to a smaller community and make it harder for attracting and developing users who were educated in a more generalist language. We identify these smaller communities as a distinct risk for a language to grow to a sustainable size and to keep attracting users and investments. As an example of the three DARPA HPCS languages Chapel, X10, and Fortress only Chapel is still going.

In generalist languages, the needs of the HPC community are often not addressed or not deemed as important as other concerns. This leads to the development of HPC dialects, based around frameworks.

We will present Julia, a modern generalist programming language, that from the beginning was designed for

<sup>1</sup> Massachusetts Institute of Technology, USA

<sup>2</sup> Oak Ridge National Laboratory, USA

<sup>3</sup> Paderborn Center for Parallel Computing, Paderborn University, Germany

<sup>4</sup> Department of Mathematics, University of Hamburg, Germany

<sup>5</sup> High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Germany

<sup>6</sup> Laboratory of Hydraulics, Hydrology and Glaciology (VAW), ETH Zurich, Switzerland

<sup>7</sup> Swiss Federal Institute for Forest, Snow and Landscape Research (WSL), Birmensdorf, Switzerland

<sup>8</sup> National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, USA

<sup>9</sup> Centre for Advanced Research Computing, United Kingdom

<sup>10</sup> Perimeter Institute, 31 Caroline St. N., Waterloo, ON, Canada N2L 2Y5

<sup>11</sup> Department of Physics and Astronomy, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

<sup>12</sup> Center for Computation & Technology, Louisiana State University, Baton Rouge, LA 70803, USA

<sup>13</sup> Swiss National Supercomputing Centre (CSCS), ETH Zurich, Switzerland

Corresponding author:  
Valentin Churavy, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA

Email: vchuravy@mit.edu

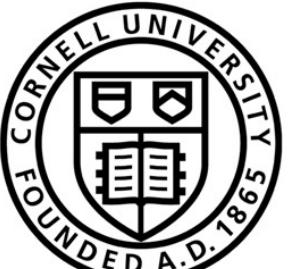
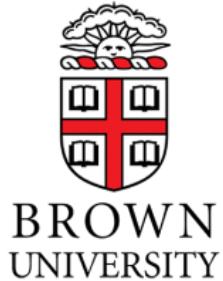
# Julia HPC Community

- Julia Slack
  - Channel: #hpc
  - [julialang.org/slack/](https://julialang.org/slack/)
- Monthly JuliaHPC Call
  - [julialang.org/community/#events](https://julialang.org/community/#events)
- JuliaCon 2022
  - 27 – 29 July
  - Recording: <https://www.youtube.com/watch?v=fog1x9rs71Q>



JULIA FOR HPC  
JuliaCon 2022 Minisymposium





EMORY  
UNIVERSITY



UNIVERSITY  
of  
GLASGOW



UNIVERSITE  
PAUL  
SABATIER



CU THE CITY  
UNIVERSITY  
OF  
NEW YORK

EPFL  
ÉCOLE POLYTECHNIQUE  
FÉDÉRALE DE LAUSANNE



MŰEGYETEM 1782



TOKYO METROPOLITAN UNIVERSITY  
首都大学東京



UCLA

AGH

Let's get started! ☺

[github.com/carstenbauer/JuliaHLRS22](https://github.com/carstenbauer/JuliaHLRS22)

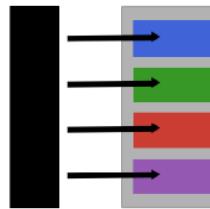
## Attribution / Copyright

- Laptop: [Laptop Vectors by Vecteezy](#)
- Cluster: [Web Vectors by Vecteezy](#)

# HPC Ecosystem

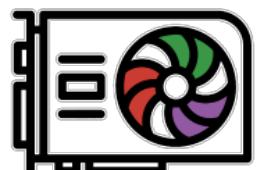
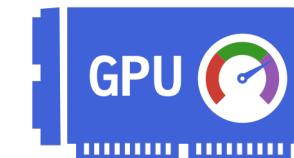
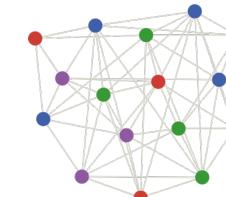
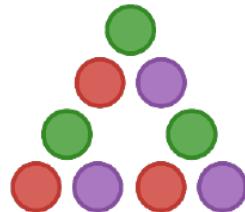
# HPC Ecosystem

- JuliaSIMD
  - LoopVectorization.jl, Polyester.jl
- JuliaParallel
  - Dagger.jl, MPI.jl
- JuliaFolds
  - Transducers.jl, FLoops.jl
- JuliaPerf
  - LIKWID.jl, PProf.jl
- JuliaGPU
  - CUDA.jl, AMDGPU.jl



**LIKWID.jl**

a Julia wrapper for LIKWID

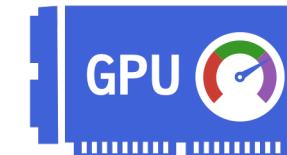
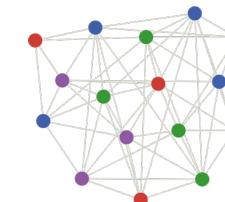


# Julia Packages by PC2

- ThreadPinning.jl
  - Dynamically pin Julia threads to CPU threads
- STREAMBenchmark.jl & BandwidthBenchmark.jl
  - Memory bandwidth measurements
- LIKWID.jl
  - Hardware performance monitoring
- GPUInspector.jl
  - Benchmark max. GPU performance
  - GPU node stresstest: monitor temperature & power



**LIKWID.jl**  
a Julia wrapper for LIKWID



[pc2.de/go/jlpkg](http://pc2.de/go/jlpkg)



# Vandermonde Matrix

## Vandermonde matrix

$$V = \begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \dots & \alpha_2^{n-1} \\ 1 & \alpha_3 & \alpha_3^2 & \dots & \alpha_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_m & \alpha_m^2 & \dots & \alpha_m^{n-1} \end{bmatrix}$$

**vander(x)**

`numpy.vander(x)`

# Python

```
def vander(x, N=None, increasing=False):
    x = asarray(x)
    if x.ndim != 1:
        raise ValueError("x must be a one-dimensional array or sequence.")
    if N is None:
        N = len(x)

    v = empty((len(x), N), dtype=promote_types(x.dtype, int))
    tmp = v[:, ::-1] if not increasing else v

    if N > 0:
        tmp[:, 0] = 1
    if N > 1:
        tmp[:, 1:] = x[:, None]
        multiply.accumulate(tmp[:, 1:], out=tmp[:, 1:], axis=1)

    return v

@TYPE@_kind@(char **args, npy_intp *dimensions, npy_intp *steps, void *NPY_UNUSED(func))
{
    if (IS_BINARY_REDUCE) {
        #if @TYPE@
            @type@ * iop1 = (@type@ *)args[0];
            npy_intp n = dimensions[0];

            *iop1 @OP@= pairwise_sum_@TYPE@(args[1], n, steps[1]);
        #else
            BINARY_REDUCE_LOOP(@type@) {
                io1 @OP@= *(@type@ *)ip2;
            }
            *((@type@ *)iop1) = io1;
        #endif
    }
    else if (!run_binary_simd_@kind@_@TYPE@(args, dimensions, steps)) {
        BINARY_LOOP {
            const @type@ in1 = *(@type@ *)ip1;
            const @type@ in2 = *(@type@ *)ip2;
            *((@type@ *)oip1) = in1 @OP@ in2;
        }
    }
}
```

calls

C code

```
PyFunc_Accumulate(PyFuncObject *ufunc, PyArrayObject *arr, PyArrayObject *out,
                   int axis, int dtype)
{
    PyArrayObject *op[2];
    PyArray_Descr *op_dtypes[2] = {NULL, NULL};
    int op_axes[2] = {op_axes_array[0], op_axes_array[1]};
    npy_uint32 op_flags[2];
    int ndim, ndim, dtype_final;
    int needs_api, need_outer_iterator;

    PyObject *iter = NULL, *iter_inner = NULL;

    /* The selected inner loop */
    PyFuncGenericFunction innerloop = NULL;
    void *innerloopdata = NULL;

    const char *ufunc_name = ufunc_get_name_cstr(ufunc);

    /* These parameters come from extobj or from a TLS global */
    int buffersize = 0, errormask = 0;

    NPY_BEGIN_THREADS_DEF;

    NPY_IF_DBG_PRINT(("Evaluating ufunc %s.accumulate(%s", ufunc_name);

    #if 0
    printf("(Doing %s.accumulate on array with dtype : ", ufunc_name);
    PyObject_Print((PyObject *)PyArray_DESCR(arr), stdout, 0);
    printf("%s)", ufunc_name);
    #endif

    if (_get_buffersize_errormask(NULL, "accumulate", &buffersize, &errormask) < 0) {
        return NULL;
    }

    /* Take a reference to out for later returning */
    Py_XINCREF(out);

    dtype_final = dtype;
    if (get_binary_op_function(ufunc, &otype_final,
                              &innerloop, &innerloopdata) < 0) {
        PyErr_Format(PyExc_ValueError,
                    "could not find a matching type for %s.accumulate.",
                    "requested type has type code '%c'.",
                    ufunc_name, dtype ? dtype->type : '-');

        Py_XDECREF(dtype);
        goto fail;
    }

    if (PyArray_ISALIGNED(arr) || (out && PyArray_ISALIGNED(out))) {
        PyArray_EquivTypes(op_dtypes[0], PyArray_DESCR(arr));
        (out ? &out : &arr) = PyArray_EquivTypes(op_dtypes[0], PyArray_DESCR(out));
    }

    need_outer_iterator = (ndim > 1);

    /* If input and output overlap in memory, use iterator to figure it out */
    else if (out != NULL && solve_my_share_memory(out, arr,
                                                     NPY_MAX_SHARE_MEMORY) < 0) {
        need_outer_iterator = 1;
    }

    if (need_outer_iterator) {
        int ndim_iter = 0;
        npy_uint32 flags = NPY_ITER_ZERO_SIZE_OK |
                           NPY_ITER_REF_OK |
                           NPY_ITER_COPY_IF_OVERLAP;
        PyArray_Descr *op_dtypes_param = NULL;

        /* The way accumulate is set up, we can't do buffering,
         * so make a copy instead when necessary.
         */
        ndim_iter = ndim;
        flags |= NPY_ITER_MULTI_INDEX;
    }

    /* Add some more flags.
     *
     * The accumulation outer loop is 'elementwise' over the array, so turn
     * count_m1 = PyArray_DIM(arr);
     */

    #if NPY_UF_DBG_TRACING
    printf("Found %s.accumulate inner loop with dtype : ", ufunc_name);
    PyObject_Print((PyObject *)op_dtypes[0], stdout, 0);
    printf("\n");
    #endif

    /* Set up the op_axes for the outer loop */
    for (idim = 0; idim < ndim; ++idim) {
        op_axes_array[0][idim] = idim;
        op_axes_array[1][idim] = idim;
    }

    /* The per-operator flags for the outer loop */
    op_flags[0] = NPY_ITER_READONLY | NPY_ITER_NO_BROADCAST |
                  NPY_ITER_ALLOCATE | NPY_ITER_NO_SUBTYPE;
    op_flags[1] = NPY_ITER_READONLY;

    op[0] = out;
    op[1] = arr;

    /* If the reduction axis has size zero, either return the reduction
     * unit for UFUNC_REDUCE, or return the zero-sized output array
     * for UFUNC_ACCUMULATE.
     */
    if (PyArray_SIZE(op[0]) == 0) {
        goto finish;
    }
    else if (PyArray_SIZE(op[1]) == 0) {
        goto finish;
    }

    /* Execute the loop with no iterators */
    npy_intp count = PyArray_DIM(op[0].axis);
    npy_intp stride0 = 0, stride1 = PyArray_STRIDE(op[0].axis);

    NPY_IF_DBG_PRINT("Func: Reduce loop with no
iterators\n");
    if (PyArray_ISDIN(op[0]) != PyArray_ISDIN(op[1])) {
        PyArray_CompareLists(PyArray_DIM(op[0]));
        PyErr_SetString(PyExc_ValueError,
                       "provided out is the wrong size "
                       "for the reduction");
    }
    else if (PyArray_GetSize(iterator) != 0) {
        char *datapr_copy[3];
        npy_intp count_m1, stride, stride;
        stride0 = PyArray_STRIDE(op[0].axis);
        stride1 = PyArray_STRIDE(op[1].axis);
        stride_copy[0] = stride0;
        stride_copy[1] = stride1;
        stride_copy[2] = stride;
    }

    /* Turn the two items into three for the inner loop */
    datapr_copy[0] = PyArray_BYTES(op[0]);
    datapr_copy[1] = PyArray_BYTES(op[1]);
    datapr_copy[2] = PyArray_BYTES(op[0]);

    #if (count_m1 > 0)
        /* Turn the two items into three for the inner
loop */
        datapr_copy[0] += stride1;
        datapr_copy[2] += stride0;
        NPY_UF_DBG_PRINT("Iterator loop count %d\n",
                         (int)count_m1);
    #endif

    if (count_m1 > 0) {
        /* Get the output */
        if (out == NULL) {
            if (iter) {
                op[0] = out = NpyIter_GetOperandArray(iterator);
                Py_INCRREF(out);
            }
            else {
                PyArray_Descr *op_dtypes[0];
                Py_INCRREF(dtype);
                op[0] = out = (PykarrayObject *)Pykarray_NewFromDescr(
                    Pykarray_Type, dtype,
                    ndim, PyArray_DIMS(op[1]), NULL,
                    0, NULL);
            }
        }
        else if (iter == NULL) {
            char *datapr_copy[3];
            npy_intp stride_copy[3];
            int itemsize = op_dtypes[0]->elsize;

            /* Execute the loop with no iterators */
            npy_intp count = PyArray_DIM(op[0].axis);
            npy_intp stride0 = 0, stride1 = PyArray_STRIDE(op[0].axis);
            NPY_IF_DBG_PRINT("Func: Reduce loop with no
iterators\n");
            if (PyArray_ISDIN(op[0]) != PyArray_ISDIN(op[1])) {
                PyArray_CompareLists(PyArray_DIM(op[0]));
                PyErr_SetString(PyExc_ValueError,
                               "provided out is the wrong size "
                               "for the reduction");
            }
            else if (PyArray_GetSize(iterator) != 0) {
                char *datapr_copy[3];
                npy_intp count_m1, stride, stride;
                stride0 = PyArray_STRIDE(op[0].axis);
                stride1 = PyArray_STRIDE(op[1].axis);
                stride_copy[0] = stride0;
                stride_copy[1] = stride1;
                stride_copy[2] = stride;
            }

            /* Turn the two items into three for the inner loop */
            datapr_copy[0] = PyArray_BYTES(op[0]);
            datapr_copy[1] = PyArray_BYTES(op[1]);
            datapr_copy[2] = PyArray_BYTES(op[0]);

            #if (count_m1 > 0)
                /* Turn the two items into three for the inner
loop */
                datapr_copy[0] += stride1;
                datapr_copy[2] += stride0;
                NPY_UF_DBG_PRINT("Iterator loop count %d\n",
                                 (int)count_m1);
            #endif
        }
    }
    else if (iter != NULL) {
        /* Copy the first element to start the reduction.
         *
         * Output (datapr[0]) and input (datapr[1]) may point
         * to the same memory, e.g. np.add.accumulate(a, out).
         */
        if (dtype == NPY_OBJECT) {

```

uses

C template

# Julia

```
function vander(x::AbstractVector{T}) where T
    m = length(x)
    V = Matrix{T}(undef, m, m)
    for j = 1:m
        V[j,1] = one(x[j])
    end
    for i= 2:m
        for j = 1:m
            V[j,i] = x[j] * V[j,i-1]
        end
    end
    return V
end
```